

PL / SQL



```
#####  
#           ==> Amélioration Continue <==           #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à cette adresse: #  
#           bit.ly/eni_ac                             #  
#####
```

PL / SQL

Module 1 – Présentation de l'écosystème ORACLE



Dans ce module vous allez comprendre

- Comment Oracle définit sa notion de base de données.
- Vous allez en savoir plus sur le leader mondial des systèmes de gestion de bases de données relationnelles.
- Vous allez aussi découvrir l'environnement de travail et les différents outils offerts pour effectuer vos développements.

Objectifs

- Découvrir Oracle
- Découvrir l'environnement logiciel
- Savoir créer un utilisateur Oracle fonctionnel
- Découvrir la notion de base de données Oracle



```
#####  
#  
#          ==> Amélioration Continue <==  
#          #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à cette adresse:  
#          #  
#          bit.ly/eni_ac  
#          #  
#####  
#
```

Les points que nous allons aborder :

- Découvrir la société Oracle
- Découvrir l'environnement logiciel d'Oracle, ou plutôt les outils que vous allez utiliser
- Savoir créer un utilisateur Oracle fonctionnel
- Et un dernier point très important vous allez découvrir la notion de base de données d'Oracle

Présentation de l'écosystème ORACLE

L'entreprise



[INTRO]

Saviez-vous que parmi les 10 hommes les plus riches du monde se trouve Larry Ellison. Vous ne le connaissez pas ?

Et pourtant il est un des cofondateurs d'Oracle .

[DIAPO]

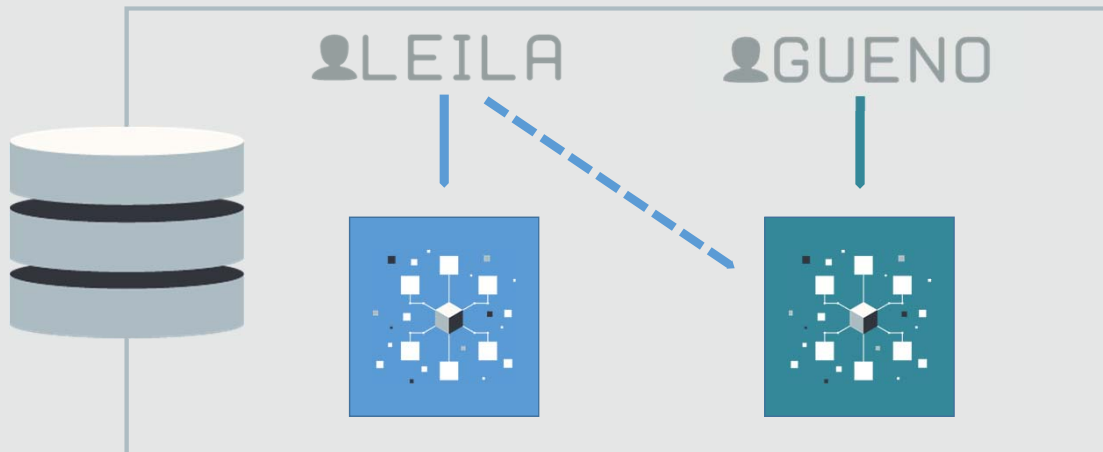
Et oui, et c'est là où l'on se rend compte de l'importance qu'ont les bases de données dans notre société. Vous vous douterez qu'Oracle fait partie des grands leaders dans le domaine des bases de données ou plus précisément dans le domaine des systèmes de gestion de base de données relationnelle!!

Cette entreprise a été précurseur dans le domaine des SGBD , elle a toujours brillé grâce à son serveur de base de données qui offre de puissants outils d'administration et de développement. Elle est aussi connue pour son rachat de SUN Microsystems qui a créé le célèbre langage Java...

Aujourd'hui le serveur d'Oracle est à la version 12c et la vous vous demandez « mais pourquoi met-on une lettre derrière le numéro de version ». Et bien le c signifie Cloud, en fait ils veulent dire que cette version est prête pour Cloud, avant nous avions la version 11g ou le g signifie grid architecture soit prêt pour les Architectures en réseau distribué et avant on avait 9i avec un i pour internet prêt pour internet...

Présentation de l'écosystème ORACLE

Notion de base de données chez Oracle



[INTRO]

Nous allons aborder la notion de base de données chez Oracle.

Chaque serveur de base de données Oracle contient au moins une instance Oracle.

L'**instance** Oracle est l'ensemble des processus et des zones mémoire qui permet de faire fonctionner le système de gestion de base de données relationnel. Je ne vais pas m'étaler sur ce sujet qui appartient plus aux administrateurs qu'au Dev. A l'intérieur de l'instance on retrouve une notion très importante chez oracle qui est le Schéma.

Le schéma est la zone dans laquelle se trouvent les tables et les différents objets d'un utilisateur.

Je parle d'un Utilisateur car chaque Utilisateur possède son propre Schéma, défini par le nom de celui-ci.

L'Utilisateur est propriétaire de ses tables. Un schéma appartient donc à un seul et uniquement un seul Utilisateur.

[DIAPO]

Leila et Gueno sont tous deux des utilisateurs.

Ils ont chacun leur schéma qui contient plusieurs tables.

On a donné à Leïla des droits sur le schéma de Gueno. Ses droits sont définis en fonctions des privilèges qu'on lui accorde.

Voilà ce qu'il faut retenir lorsque l'on parle de schéma dans un environnement Oracle. Un schéma est égal à un utilisateur. Le schéma est la zone dans laquelle se trouve les tables et les différents objets d'un utilisateur.

Création d'un utilisateur avec des droits

```
create role eni;
grant connect, resource to eni;
grant create procedure to eni;
grant create trigger to eni;

create user u1
identified by x
default tablespace users
temporary tablespace temp
quota unlimited on users;

grant eni to u1;
```



```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
# cette adresse:               #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

[INTRO]

Je vais vous présenter la commande permettant de créer un nouvel Utilisateur.
Nous allons d'abord créer un rôle associé à plusieurs privilèges.
Ensuite nous créons un nouvel utilisateur.
Puis enfin nous associerons ce nouvel utilisateur à notre rôle.

[DIAPO]

Voici le script :

Ligne 1 : Création du rôle nommé eni

Ligne 2 : On associe à ce rôle les privilèges Connect et Resource qui nous donne la possibilité de créer des objets SQL et PL/SQL.

Ligne 3 : On va lier des privilèges plus spécifiques à notre rôle tels que la possibilité de créer des procédures.

Ligne 4 : Mais aussi la possibilité de créer des triggers.

Nous avons terminé la création de notre rôle. Nous allons passer à la création de notre utilisateur

Ligne 6 : Création d'un utilisateur nommé u1

Ligne 7 : Le mot de passe de l'utilisateur sera x

Ligne 8 : On définit le tablespace de données de l'utilisateur. C'est-à-dire la zone dans laquelle l'utilisateur va pouvoir stocker ses données.

Ligne 9 : On attribue à l'utilisateur un tablespace temporaire destiné aux opérations qui ont besoin d'espace comme des opérations d'analyse.

Ligne 10 : On définit le quota d'espace attribué à l'utilisateur à l'intérieur de ces tablespace. On lui donne un droit illimité.

Ces opérations sont plutôt des opérations d'administration Oracle. On ne s'étalera pas plus sur le sujet.

Présentation de l'écosystème ORACLE L'environnement logiciel



Voici l'environnement logiciel que nous allons utiliser :

- Le premier est SQL Plus qui est un interpréteur de commande.
- Le second est SQL Developer qui est un IDE avec une interface graphique.

Démonstration



Démonstration SQL Plus:

Exécuter / cmd

SQL plus

Saisir le nom : u

Saisir le mode de passe : x

CONNECT u/x as SYSDBA ;

Maintenant que vous êtes connecté vous pouvez manipuler votre BDD.

Nous allons créer une simple table nommée prénoms

Create table prénoms (prenom varchar2(50)) ;

Insert into prénoms ('Anthony') ;

SELECT * FROM prénoms ;

Nous allons ensuite créer un nouvel utilisateur

-- Création d'un rôle avec des privilèges particuliers

create demo_role eni;

grant connect, resource to demo_role;

grant create procedure to demo_role;

grant create trigger to demo_role;

--Création de l'utilisateur

create user roger

identified by x

default tablespace users

temporary tablespace temp

```
quota unlimited on users;  
-- Assigner un rôle à l'utilisateur  
grant demo_role to roger;
```

Voilà voilà, je n’irai pas plus loin dans cette démonstration, vous l’avez compris SQL Plus est un interpréteur de commande pour manipuler votre serveur de BDD.

Démonstration



Démonstrations SQL Développeur :

Lancer SQL Developer.

Se connecter. (**Attention ! Si vous utiliser « Local » en type de connexion**)

Développer l'arborescence du côté gauche et présenter les différents dossiers de ressources.

Créer une nouvelle feuille de calcul

Créer une table

Faites une requête sur cette table.

Montrer exécuter un script.

Montrer exécuter une commande.

Terminé.

Conclusion

- Vous connaissez l'entreprise Oracle
- Vous avez compris le concept de schéma
- Vous avez découvert SQL Plus
- Vous savez créer un nouvel utilisateur
- Vous avez découvert SQL Developer



C'est la fin du module.

Maintenant :

- Vous connaissez l'entreprise Oracle
- Vous avez compris le concept de schéma
- Vous avez découvert SQL Plus
- Vous savez créer un nouvel utilisateur
- Vous avez découvert SQL Developer

```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse: #  
#                bit.ly/eni_ac  
#  
#####  
#####
```

PL / SQL

Module de rappel – La gestion des tables sous Oracle



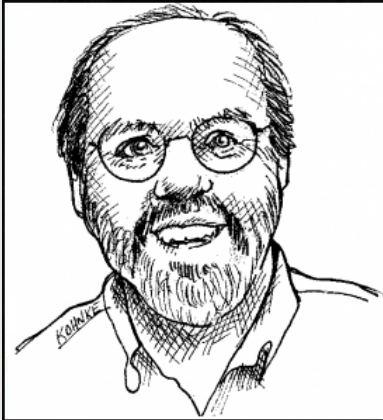
Objectifs

- Rappel des instructions du DDL
- Découverte des particularités Oracle



La gestion des tables sous Oracle

La dette technique



Inventeur du concept de wiki et
a permis l'élaboration de
Wikipédia

-Ward Cunningham-



Ward Cunningham est un acteur du logiciel libre et aussi à l'origine du concept de wiki. Expert en développement logiciel, il a aussi inventé un concept nommé « Dette technique ».

La gestion des tables sous Oracle

La dette technique

Quand on code au plus vite et de manière non optimale, on contracte une dette technique que l'on rembourse tout au long de la vie du projet sous forme de temps de développement de plus en plus long et de bugs de plus en plus fréquents.



A répéter plusieurs fois afin que cela soit bien compris.

La gestion des tables sous Oracle

Les conventions de nommage

- Mots clés SQL en majuscules
- Nom des tables au pluriel et en snake_case
- Nom des colonnes explicites et en snake_case



https://docs.oracle.com/cd/A97630_01/appdev.920/a96624/02_funds.htm

Voici quelques clés pour réduire sa dette technique.

Les commentaires de code

```
/* Je suis  
    un commentaire sur  
    plusieurs lignes */  
  
--Je suis un commentaire sur une ligne
```



Les principaux types caractères

`CHAR (Size)`

`VARCHAR (Size)`

`LONG`



Le principal type numérique

`NUMBER (precision, scale)`

Valeur réelle	Spécification de la colonne	Valeur stockée
123,89	NUMBER	123,89
123,89	NUMBER(3)	124
123,89	NUMBER(3,2)	Impossible
123,89	NUMBER(6,-2)	100
0,000127	NUMBER(4,5)	0,00013



La gestion des tables sous Oracle

Les principaux types dates

DATE

TIMESTAMP



La gestion des tables sous Oracle

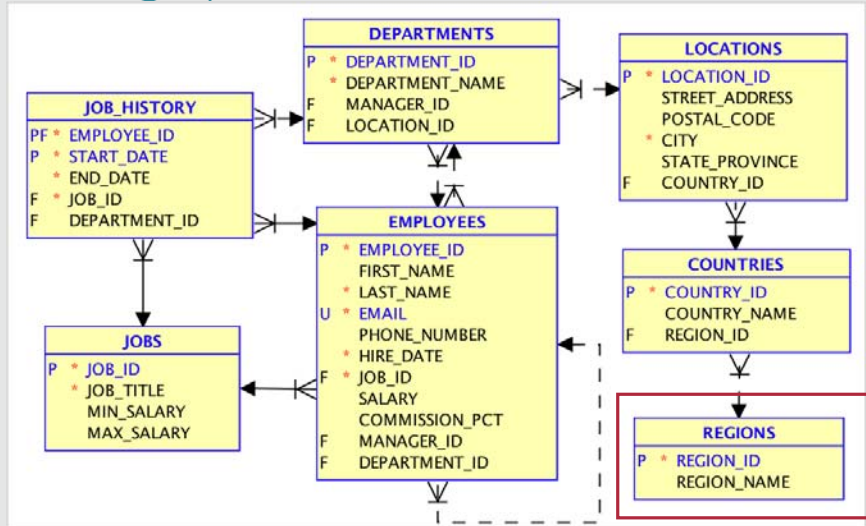
Le principal type multimédia

BLOB



La gestion des tables sous Oracle

Le schéma logique de base de données



La gestion des tables sous Oracle

La création de la table REGIONS

```
CREATE TABLE regions
(
  region_id      NUMBER PRIMARY KEY,
  region_name    VARCHAR2(25) DEFAULT 'XXX'
);
```



La gestion des tables sous Oracle

Les commentaires posés sur les objets

```
COMMENT ON TABLE regions
IS 'Regions table that contains region numbers and names. Contains 4 rows; references with the Countries table.'

COMMENT ON COLUMN regions.region_id
IS 'Primary key of regions table.'

COMMENT ON COLUMN regions.region_name
IS 'Names of regions. Locations are in the countries of these regions.'
```



La consultation des commentaires

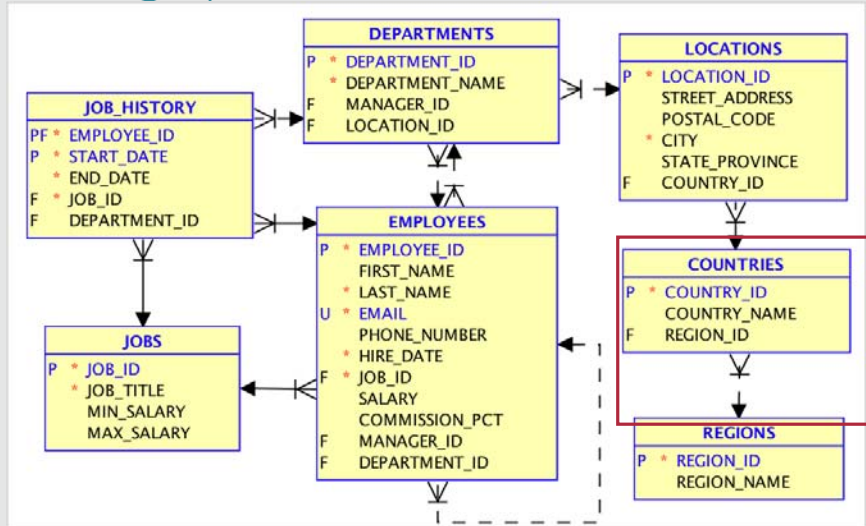
```
SELECT * FROM user_tab_comments;
```

```
SELECT * FROM user_col_comments;
```



La gestion des tables sous Oracle

Le schéma logique de base de données



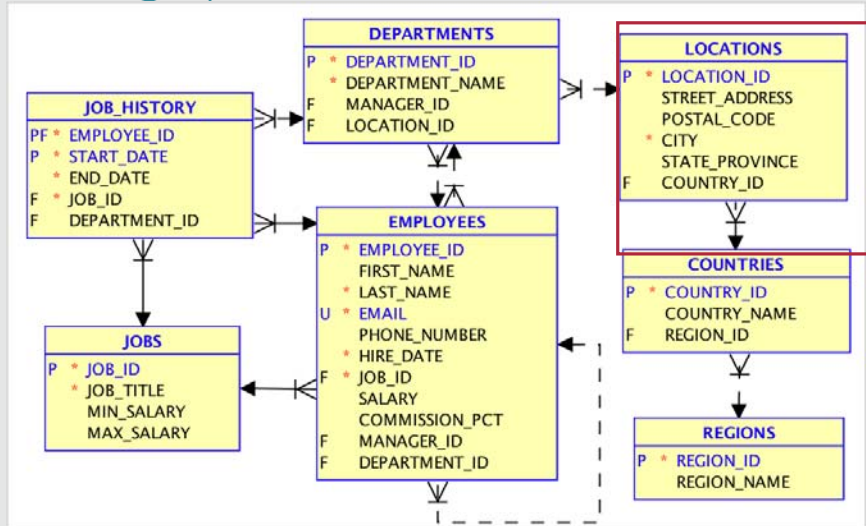
La création de la table COUNTRIES

```
CREATE TABLE countries
(
    country_id CHAR(2) CONSTRAINT country_id_nn NOT NULL,
    country_name VARCHAR2(40) ,
    region_id   NUMBER ,
    CONSTRAINT country_c_id_pk PRIMARY KEY (country_id)
);

ALTER TABLE countries
ADD (
    CONSTRAINT countr_reg_fk
    FOREIGN KEY (region_id)
    REFERENCES regions(region_id)
);
```



Le schéma logique de base de données



La création de la table LOCATIONS

```
CREATE TABLE locations
(
    location_id    NUMBER(4) CONSTRAINT loc_id_pk PRIMARY KEY,
    street_address VARCHAR2(40),
    postal_code    VARCHAR2(12),
    city           VARCHAR2(30) CONSTRAINT loc_city_nn NOT NULL,
    state_province VARCHAR2(25),
    country_id     CHAR(2) CONSTRAINT loc_c_id_fk REFERENCES countries(country_id)
);
```



La création d'une séquence



La gestion des tables sous Oracle

La création d'une séquence

```
CREATE SEQUENCE locations_seq  
  START WITH      3300  
  INCREMENT BY    100  
  MAXVALUE        9900  
  NOCACHE  
  NOCYCLE;
```



La gestion des tables sous Oracle

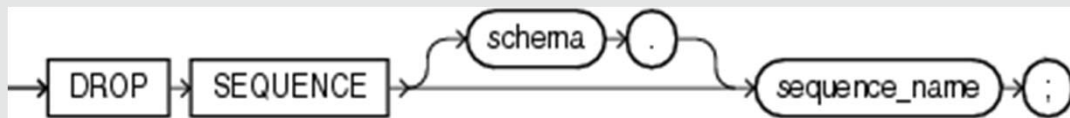
L'utilisation d'une séquence

```
locations_seq.NEXTVAL
```

```
locations_seq.CURRVAL
```



La suppression d'une séquence

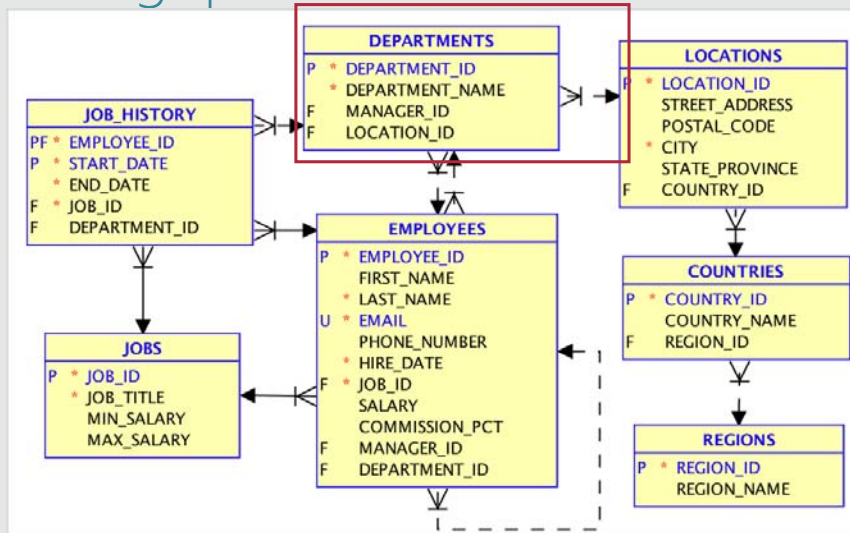


```
DROP SEQUENCE departments_seq;  
DROP SEQUENCE employees_seq;  
DROP SEQUENCE locations_seq;
```



La gestion des tables sous Oracle

Le schéma logique de base de données



La création de la table DEPARTMENTS

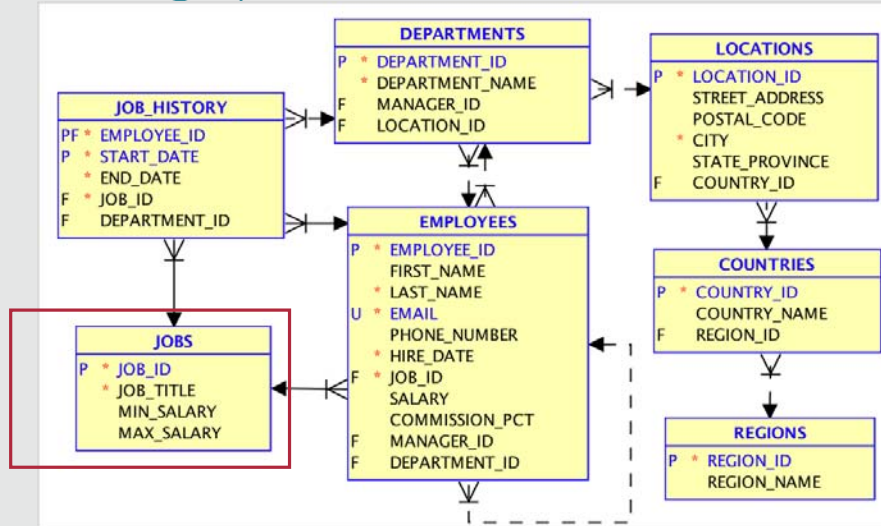
```
CREATE TABLE departments
(
    department_id    NUMBER(4),
    department_name  VARCHAR2(30) CONSTRAINT dept_name_nn NOT NULL,
    manager_id       NUMBER(6),
    location_id      NUMBER(4)
);

ALTER TABLE departments
ADD (
    CONSTRAINT dept_id_pk PRIMARY KEY (department_id),
    CONSTRAINT dept_loc_fk FOREIGN KEY (location_id) REFERENCES locations (location_id)
);

CREATE SEQUENCE departments_seq
START WITH          280
INCREMENT BY        10
MAXVALUE            9990
NOCACHE
NOCYCLE;
```



Le schéma logique de base de données



La création de la table JOBS

```
CREATE TABLE jobs
(
  job_id          VARCHAR2(10),
  job_title       VARCHAR2(35)
  CONSTRAINT      job_title_nn  NOT NULL,
  min_salary      NUMBER(6),
  max_salary      NUMBER(6)
);

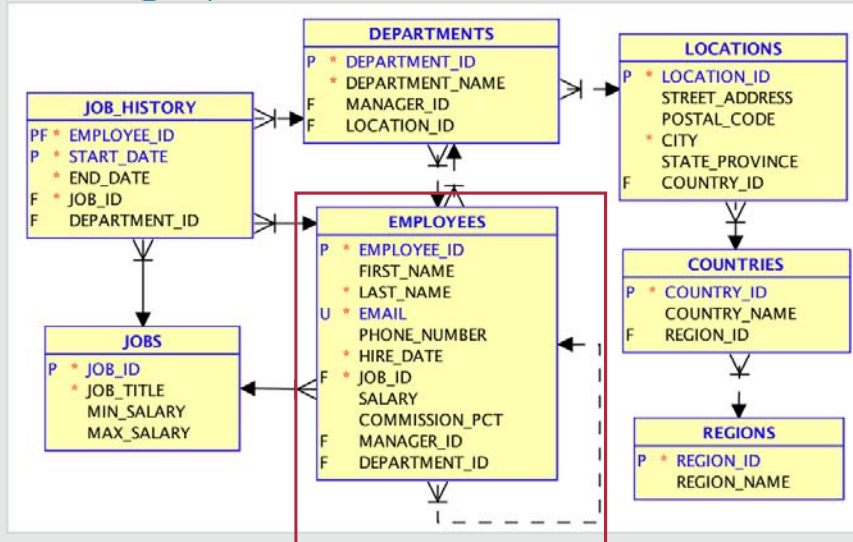
CREATE UNIQUE INDEX job_id_pk ON jobs (job_id) ;

ALTER TABLE jobs
ADD (
  CONSTRAINT job_id_pk PRIMARY KEY(job_id)
) ;
```



La gestion des tables sous Oracle

Le schéma logique de base de données



La création de la table EMPLOYEES

```
CREATE TABLE employees
(
  employee_id      NUMBER(6),
  first_name       VARCHAR2(20),
  last_name        VARCHAR2(25) CONSTRAINT emp_last_name_nn NOT NULL,
  email            VARCHAR2(25) CONSTRAINT emp_email_nn NOT NULL,
  phone_number     VARCHAR2(20),
  hire_date        DATE CONSTRAINT emp_hire_date_nn NOT NULL,
  job_id           VARCHAR2(10) CONSTRAINT emp_job_nn NOT NULL,
  salary           NUMBER(8,2),
  commission_pct   NUMBER(2,2),
  manager_id       NUMBER(6),
  department_id    NUMBER(4),
  ,CONSTRAINT emp_salary_min CHECK (salary > 0)
  ,CONSTRAINT emp_email_uk UNIQUE (email)
);
```



Mise en évidence des contraintes CHECK et UNIQUE

L'intégration de la table EMPLOYEES

```
ALTER TABLE employees
ADD (
    CONSTRAINT emp_emp_id_pk PRIMARY KEY (employee_id),
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id) REFERENCES departments,
    CONSTRAINT emp_job_fk FOREIGN KEY (job_id) REFERENCES jobs (job_id),
    CONSTRAINT emp_manager_fk FOREIGN KEY (manager_id) REFERENCES employees
) ;

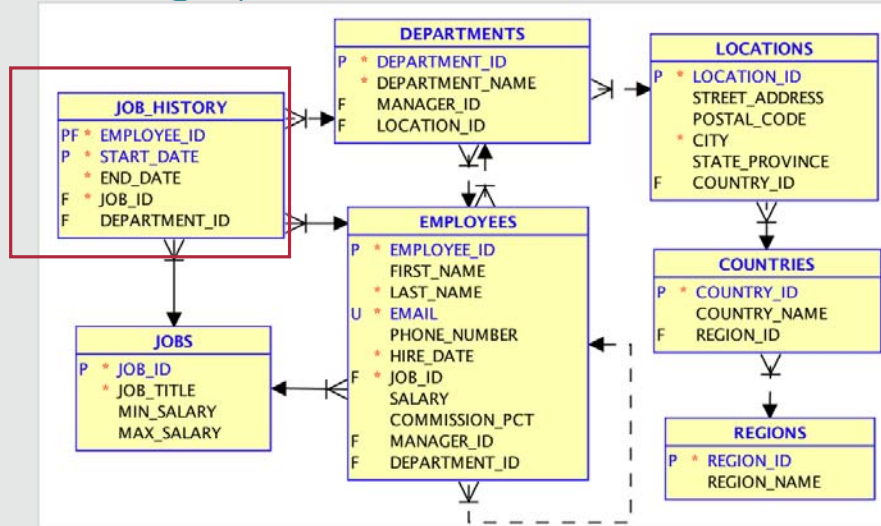
ALTER TABLE departments
ADD (
    CONSTRAINT dept_mgr_fk FOREIGN KEY (manager_id) REFERENCES employees (employee_id)
) ;

CREATE SEQUENCE employees_seq
START WITH 207
INCREMENT BY 1
NOCACHE
NOCYCLE;
```



La gestion des tables sous Oracle

Le schéma logique de base de données



Les contraintes portant sur plusieurs colonnes

```
CREATE TABLE job_history
(
  employee_id  NUMBER(6) CONSTRAINT jhist_employee_nn NOT NULL,
  start_date   DATE CONSTRAINT jhist_start_date_nn NOT NULL,
  end_date     DATE CONSTRAINT jhist_end_date_nn NOT NULL,
  job_id       VARCHAR2(10) CONSTRAINT jhist_job_nn NOT NULL,
  department_id NUMBER(4),
  CONSTRAINT jhist_date_interval CHECK (end_date > start_date)
);

CREATE UNIQUE INDEX jhist_emp_id_st_date_pk ON job_history (employee_id, start_date) ;

ALTER TABLE job_history
ADD (
  CONSTRAINT jhist_emp_id_st_date_pk PRIMARY KEY (employee_id, start_date),
  CONSTRAINT jhist_job_fk FOREIGN KEY (job_id) REFERENCES jobs,
  CONSTRAINT jhist_emp_fk FOREIGN KEY (employee_id) REFERENCES employees,
  CONSTRAINT jhist_dept_fk FOREIGN KEY (department_id) REFERENCES departments
) ;
```



La gestion des tables sous Oracle

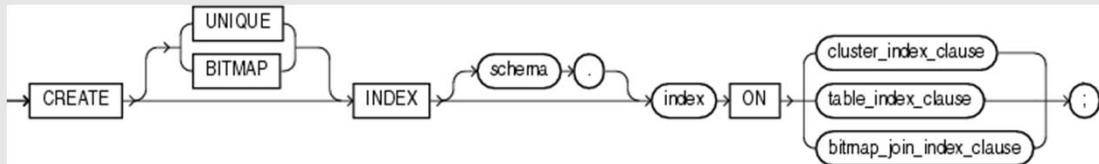
Le vidage d'une table

```
TRUNCATE TABLE informations;
```



La gestion des tables sous Oracle

La syntaxe de création d'un index



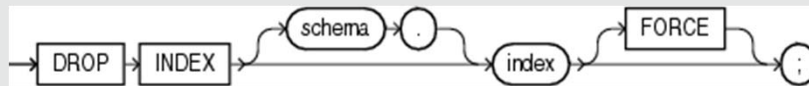
Exemple de création d'un index

```
CREATE INDEX emp_department_ix ON employees (department_id);  
  
CREATE INDEX emp_job_ix ON employees (job_id);  
  
CREATE INDEX emp_manager_ix ON employees (manager_id);  
  
CREATE INDEX emp_name_ix ON employees (last_name, first_name);  
  
CREATE INDEX dept_location_ix ON departments (location_id);  
  
CREATE INDEX jhist_job_ix ON job_history (job_id);  
  
CREATE INDEX jhist_employee_ix ON job_history (employee_id);  
  
CREATE INDEX jhist_department_ix ON job_history (department_id);  
  
CREATE INDEX loc_city_ix ON locations (city);  
  
CREATE INDEX loc_state_province_ix ON locations (state_province);  
  
CREATE INDEX loc_country_ix ON locations (country_id);
```



La gestion des tables sous Oracle

La suppression d'un index



```
DROP INDEX dept_location_ix
```



La gestion des tables sous Oracle

La suppression d'une table



```
DROP TABLE regions      CASCADE CONSTRAINTS;  
DROP TABLE departments  CASCADE CONSTRAINTS;  
DROP TABLE locations    CASCADE CONSTRAINTS;  
DROP TABLE jobs         CASCADE CONSTRAINTS;  
DROP TABLE job_history  CASCADE CONSTRAINTS;  
DROP TABLE employees    CASCADE CONSTRAINTS;  
DROP TABLE countries    CASCADE CONSTRAINTS;
```



La gestion des tables sous Oracle

La suppression d'une contrainte

```
ALTER TABLE locations DROP CONSTRAINT loc_city_nn;
```



La gestion des tables sous Oracle

L'ajout ou suppression d'une colonne

```
ALTER TABLE departments ADD dn VARCHAR2(300);  
  
ALTER TABLE departments DROP COLUMN dn;
```



La gestion des tables sous Oracle

Le changement de nom d'une colonne

```
ALTER TABLE regions  
RENAME COLUMN region_name TO region_true_name;
```



La gestion des tables sous Oracle

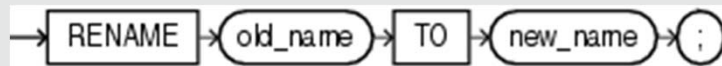
La modification d'une colonne

```
ALTER TABLE  
    departments  
MODIFY  
    department_name VARCHAR2 (50) ;
```



La gestion des tables sous Oracle

Le changement de nom d'une table



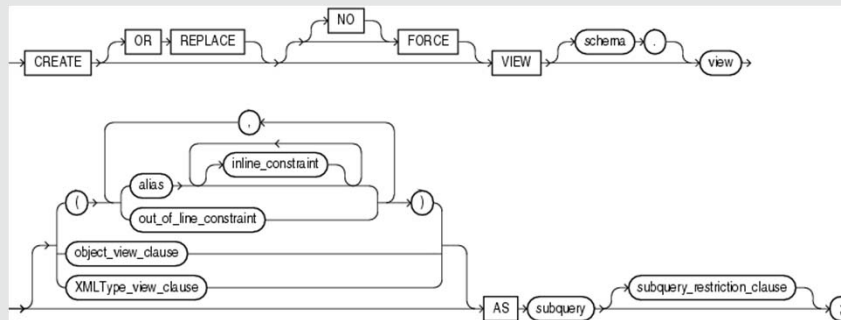
La définition d'une colonne identité

```
CREATE TABLE informations
(
    informations_id NUMBER GENERATED AS IDENTITY,
    informations_text VARCHAR(500)
)
```



La gestion des tables sous Oracle

La syntaxe de création d'une vue



Exemple de création d'une vue

```
CREATE OR REPLACE FORCE VIEW regions_countries_view
AS
SELECT
    r.region_name,
    c.country_name
FROM
    regions r
JOIN countries c ON r.region_id = c.region_id;
```



La gestion des tables sous Oracle

La suppression d'une vue



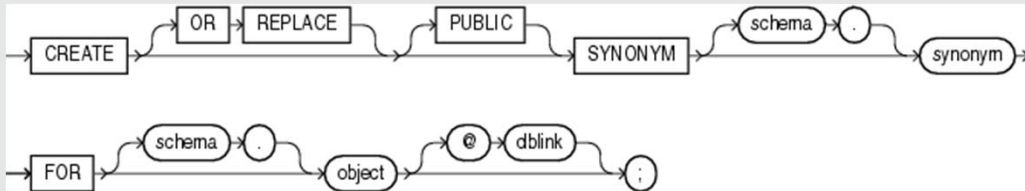
La gestion des tables sous Oracle

Exemple de suppression d'une vue

```
DROP VIEW regions_countries_view;
```



La syntaxe de création d'un synonyme



```
CREATE SYNONYM history FOR job_history;
```



La gestion des tables sous Oracle

La suppression d'un synonyme

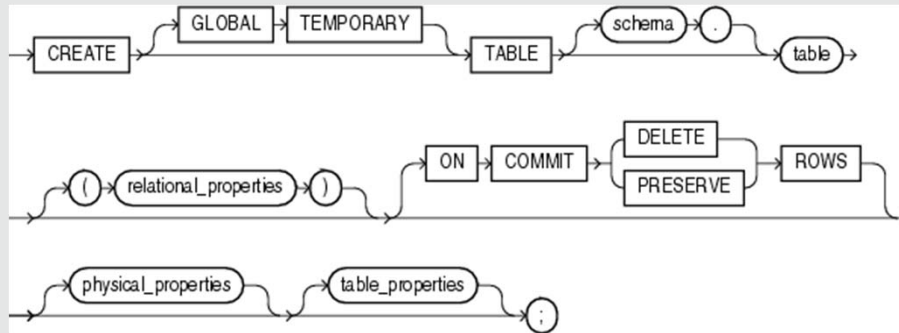


```
DROP SYNONYM history;
```



La gestion des tables sous Oracle

La syntaxe de création d'une table temporaire



Exemple de création d'une table temporaire

```
CREATE GLOBAL TEMPORARY TABLE employees_sel  
(  
    id VARCHAR2(256),  
    end_date DATE  
) ON COMMIT PRESERVE ROWS ;
```



Conclusion

- Vous savez mettre en place une base de données sous Oracle



PL / SQL

Module de rappel – La gestion des données sous Oracle



Objectifs

- Rappel du DML
- Découverte des particularités Oracle



La gestion des données sous Oracle

L'insertion d'enregistrement



`INSERT INTO Table VALUES (valeur_1, valeur_2, valeur_3)`

`INSERT INTO Table (colonne_1, colonne_3) VALUES (valeur_1, valeur_3)`



Il faut que tous les attributs NOT NULL soient définis lors d'un INSERT.

Usages possibles :

1. Insertion de données sans spécifier les attributs :

```
INSERT INTO Table  
VALUES (valeur_1, valeur_2, valeur_3)
```

2. Insertion de données avec spécification des attributs :

```
INSERT INTO Table (colonne_1, colonne_3)  
VALUES (valeur_1, valeur_3)
```

Exemples d'insertions d'enregistrements

```
INSERT INTO regions(region_id,region_name) VALUES (1, 'Europe' );  
INSERT INTO regions VALUES (2,'Americas');  
INSERT INTO regions VALUES ( 3, 'Asia');  
INSERT INTO regions VALUES ( 4, 'Middle East and Africa');
```



La gestion des données sous Oracle

La mise à jour d'enregistrement(s)

UPDATE

UPDATE Table

SET colonne = valeur

SET

UPDATE Table

SET colonne_1 = valeur

WHERE

WHERE colonne_2 = condition



Usages possibles:

1. Modification de toutes les lignes d'une table :

UPDATE Table

SET colonne = valeur

2. Modification des lignes d'une table selon condition(s) :

UPDATE Table

SET colonne_1 = valeur

WHERE colonne_2 = condition

La gestion des données sous Oracle

Exemples de mises à jour d'enregistrements

```
UPDATE regions SET region_name = UPPER(region_name)
```

```
UPDATE regions SET region_name = 'America' WHERE region_name = 'Americas'
```



La gestion des données sous Oracle

La suppression d'enregistrement(s)

DELETE

DELETE FROM Table

FROM

DELETE FROM Table WHERE colonne = condition

WHERE



Usages possibles :

1. Suppression de toutes les lignes d'une table :

DELETE FROM Table

1. Suppression des lignes d'une table selon condition(s) :

DELETE FROM Table
WHERE colonne = condition

Exemples de suppressions d'enregistrements

```
DELETE FROM regions WHERE region_id = 1;  
  
DELETE regions;
```



La gestion des données sous Oracle

La validation des modifications

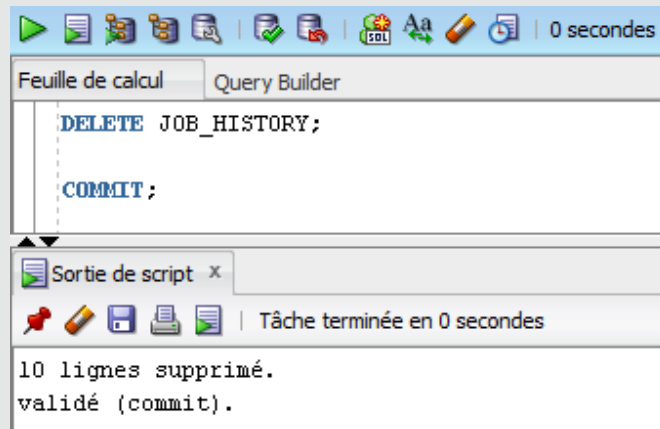
Enregistre en base toutes les insertions, modifications et suppressions réalisées depuis le début de la transaction.

Tant qu'il n'y a pas eu COMMIT, **seule la connexion courante** voit ses mises à jour.



La gestion des données sous Oracle

La validation des modifications



La gestion des données sous Oracle

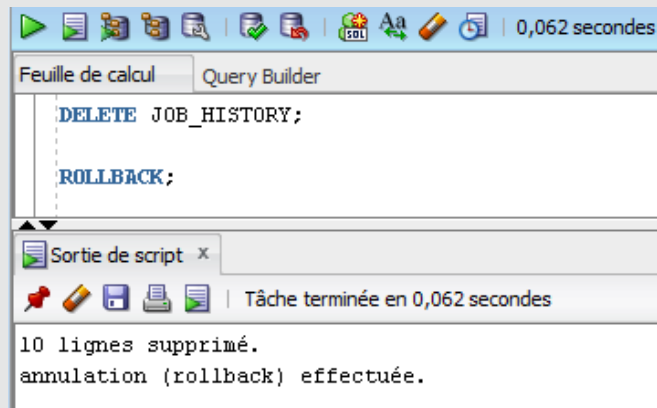
L'invalidation des modifications

Annule toutes les insertions, modifications et suppressions réalisées depuis le début de la transaction.



La gestion des données sous Oracle

L'invalidation des modifications



Conclusion

- Vous savez modifier des données sous Oracle
- Vous savez valider vos modifications
- Vous savez annuler vos modifications



PL / SQL

Module 2 – Le langage PL / SQL



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

Prérequis obligatoires :

-Les stagiaires **doivent savoir utiliser** le langage SQL.

Objectifs

- Savoir expliquer ce qu'est le PL / SQL
- Comprendre l'intérêt du PL / SQL

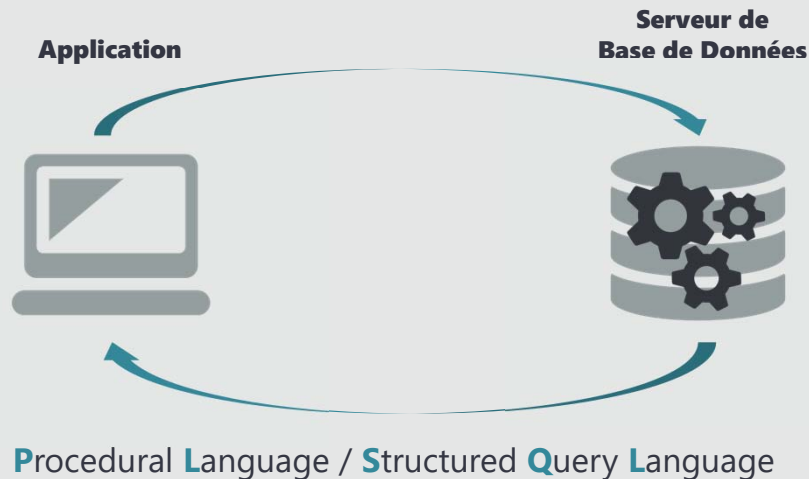


```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

Nous allons aborder les points suivants :

- Savoir expliquer ce qu'est le PL / SQL
- Comprendre l'intérêt du PL / SQL

Le langage PL / SQL C'est quoi ?



[INTRO]

Le PL/SQL c'est quoi ?

On pourrait déjà deviner grâce au sigle PL/SQL.

PL signifie Procedural Language et SQL pour le langage SQL qui lui-même signifie Structured Query Language.

Le PL/SQL est un langage qui combine des requêtes SQL avec des variables et des instructions procédurales telles que la boucle for, les structures de contrôle if etc. etc.

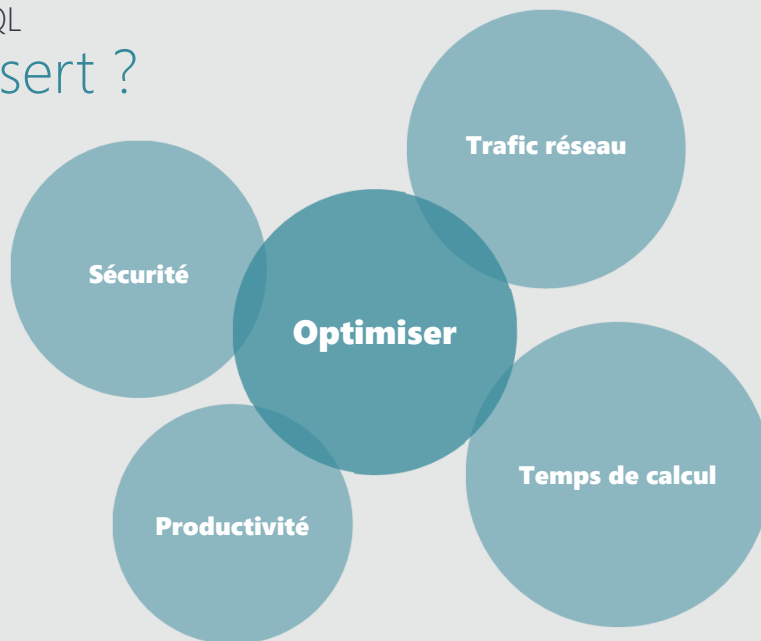
[DIAPO]

Les applications font donc appel aux programmes PL/SQL qui s'exécute sur le serveur de base de données.

Ce langage a été créé par Oracle qui en est propriétaire. Il permet **de créer des programmes qui s'exécutent sur les environnements Oracle donc sur notre serveur de base de données Oracle.**

Le langage PL / SQL

À quoi ça sert ?



```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à cette
adresse:      #
#                               bit.ly/eni_ac
#                               #
```

```
#####
#####
```

[DIAPO]

La sécurité

Le trafic Réseau

Les temps de calcul

La productivité

La sécurité

En permettant par exemple aux utilisateurs d'accéder à une table en lecture ou en écriture uniquement via des fonctions PL/SQL.

Ainsi :

Ces fonctions peuvent empêcher des actions dangereuses lors des modifications.

Ces fonctions peuvent contrôler les données à fournir à l'utilisateur (et permettre par exemple de cacher un compte en banque) et cela sans donner d'informations sur la structure de la base de données.

[DIAPO]

Le Traffic réseau

Les traitements logiques de données étant stockées dans des fonctions sur le serveur de base de données cela permet de ne pas envahir le trafic réseau entre le client et le serveur de base de données avec les requêtes nécessaires au traitement.

meilleure gestion du code.

De plus, on peut utiliser un environnement dédié au PL/SQL qui offre un confort au développeur pour la conception , le débogage des requêtes et des fonctionnalités d'accès aux données.

Sur le serveur de base de données nous n'avons qu'une seule copie des fonctionnalités plutôt qu'une copie des fonctionnalités pour chaque client. On peut donc faire plus simplement des mises à jour sans affecter les différentes applications clientes.

Pour finir, on peut faire plus simplement des modifications du schéma de la base de données sans impacter toutes les applications clientes.

Point négatif

Néanmoins l'utilisation du PL/SQL peut-être très problématique si un jour on décide de changer de SGBD. Mais cela n'est pas fréquent.

Conclusion

- Vous savez que le PL/SQL est un langage procédural
- Vous savez que le PL/SQL s'exécute sur le serveur
- Vous savez que le PL/SQL permet d'accéder aux données de manière optimisée



[INTRO]

Le PL/SQL c'est un langage procédural s'exécutant sur le serveur de base de données permettant d'accéder aux données de manière optimisée.
C'est la fin du module. Maintenant

[DIAPO]

-Vous savez que le PL/SQL est un langage procédural
-Vous savez que le PL/SQL s'exécute sur un serveur de base de données
-Vous savez que le PL/SQL permet d'accéder aux données de manière optimisée

```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:      #  
#                bit.ly/eni_ac
```

#

#####

PL / SQL

Module 3 – Les blocs PL/SQL



```
#####
#####
#           ==> Amélioration Continue <==
#
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:      #
#                   bit.ly/eni_ac
#                   #
#####
#####
```

Objectifs

- Sensibilisation aux conventions de nommage
- Comprendre ce qu'est un bloc PL/SQL
- Connaître les différents types de blocs PL/SQL
- Découvrir le plus petit bloc PL/SQL du monde
- Connaître les sections possibles d'un bloc PL/SQL



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

[INTRO]

Voici les points que nous allons aborder :

[DIAPO]

- Tout d'abord nous allons faire un point sur quelques bonnes pratiques à propos des conventions de nommage.
- Ensuite vous comprendrez ce qu'est un bloc PL/SQL
- De plus, vous verrez les différents types de blocs en PL/SQL
- Puis vous découvrirez le plus petit bloc PL/SQL du monde

- Et enfin pour terminer vous allez voir quelles structures peut avoir un bloc PL/SQL.

Une fois ces objectifs atteint le PL/SQL vous paraîtra plus clair.

La dette technique

Quand on code au plus vite et de manière non optimale, on contracte une dette technique que l'on rembourse tout au long de la vie du projet sous forme de temps de développement de plus en plus long et de bugs de plus en plus fréquents.



[INTRO]

Vous connaissez Ward Cunningham ?

J'imagine que non et pourtant je suis sûr que vous utilisez au moins une fois par semaine un certain outil qui existe grâce à lui.

Vous ne voyez toujours pas ? Et bien je parle de Wikipédia .

En effet, Ward Cunningham est la personne qui a inventé le concept de wiki et qui a permis notamment l'élaboration de Wikipédia.

Alors pourquoi je vous parle de Ward Cunningham ? Et bien parce qu'il est à l'origine d'un autre concept nommé Dette technique.

Accrochez-vous, Le concept de dette technique c'est de dire que lorsqu'on code au plus vite et de manière non optimale, on contracte une dette technique que l'on rembourse tout au long de la vie du projet sous forme de temps de développement de plus en plus long et de bugs de plus en plus fréquents.

[DIAPO]

X2

Quand on code au plus vite et de manière non optimale, on contracte une dette technique que l'on rembourse tout au long de la vie du projet sous forme de temps de développement de plus en plus long et de bugs de plus en plus fréquents.

Par exemple je suis en train de mettre en place la base données de mon projet et pour gagner du temps je décide de ne pas mettre de clé étrangère. Erreur fatale !! Ici, nous sommes tous d'accord, je suis en train de développer une bdd au plus vite et surtout de manière non optimale. Certes sur le coup j'ai peut-être gagné 10 minutes... Mais lorsque je serai en production et que je m'apercevrai que mes données ne sont plus cohérentes et que cela engendre des bugs je devrai remettre à niveau toute mes données. Et là je peux vous dire que dans le meilleur des cas, j'en aurai au moins pour 1 journée et je risque de perdre des informations.

Développez votre code de la meilleure manière et dès la première écriture.

Donc, une des clés pour diminuer sa dette technique c'est respecter des conventions de nommage.

Toute équipe de dev doit respecter des conventions de nommage, soit elle respecte des conventions maison ou sinon elle peut reprendre la convention de nommage officielle du langage utilisé.

Alors, je ne vais pas vous faire la liste de toute les conventions Oracle, je vais seulement vous en donner deux (avec les doigts)

Les clés pour réduire la dette technique

```
nombre_adherents INT;
```

```
UPDATE  
    benevoles  
SET  
    actif = 1;
```



https://docs.oracle.com/cd/A97630_01/appdev.920/a96624/02_funds.htm

[DIAPO]

Bien qu'Oracle ne soit pas sensible à la casse et qu'il met automatiquement tous les mots en majuscules. Une des clés pour diminuer sa dette technique c'est respecter des conventions de nommage.

Toute équipe de dev doit respecter des conventions de nommage, soit elle respecte des conventions maison ou sinon elle peut reprendre la convention de nommage officielle du langage utilisé.

Présentation des deux exemples :

- La première convention c'est décrire tous les mots clés SQL et PL/SQL en majuscules.
- La seconde convention c'est d'écrire les noms de variables de manière explicite et en snake_case c'est-à-dire que chaque mots-clés de la variable doivent être séparé par des Under score .

Il est très important d'avoir des noms de variables explicites car cela permet d'autocommenter le code !

Et comme le disait le grand Albert Camus « Mal nommer les choses c'est ajouter

aux malheurs du monde »

```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
cette adresse:  #  
#                bit.ly/eni_ac  
#  
#####  
#####
```

Les blocs PL / SQL

Définition d'un bloc PL/SQL



[INTRO]

Je vous le rappelle, un bloc PL/SQL est un programme qui s'exécute sur le serveur de base de données.

Je représenterai ça comme ceci :

[DIAPO]

L'engrenage représente l'exécution du bloc PL/SQL sur le serveur.

Le bloc interne

Envoi une requête avec le nom du bloc à exécuter



Le bloc est stocké et exécuté sur le serveur



[DIAPO]

Un bloc PL/SQL interne c'est un bloc qui est associé à un nom et qui est stocké sur le serveur de base de données.

Ainsi lorsque l'on veut exécuter notre bloc interne, l'utilisateur va faire appel à son nom et celui-ci va ensuite s'exécuter.

Les blocs internes sont dits stockés on les appelle « fonction stockée » ou « procédure stockée » ou « trigger », cela dépend de leur utilité.

Les blocs PL / SQL

Le bloc externe

Envoi une requête avec le contenu du bloc à exécuter



Le bloc est exécuté sur le serveur



[DIAPO]

Un bloc PL/SQL externe est un bloc que l'on appelle plus communément un bloc anonyme. Celui-ci n'est pas stocké sur le serveur de base de données.

Ils sont plutôt utilisés pour des opérations de maintenance, de tests et phase de développement. On développe le bloc sur un IDE côté client puis on envoie le bloc au serveur de base de données qui l'exécutera.

En général, lorsqu'on veut commencer à découvrir les blocs PL/SQL on commence par développer des blocs anonymes.

Le plus petit bloc du monde

```
BEGIN
    NULL; --Traitement
END;
/
```



```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
# cette adresse:               #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

[INTRO]

Le plus petit bloc PL/SQL du monde ! Je vous sens déjà impatient de le découvrir !

Alors sans plus tarder je vous le présente !

[DIAPO]

Alors ici on a à faire à un bloc externe, donc anonyme car il n'est pas associé à un nom.

La première ligne définit le début du traitement procédural. On dit plus

communément que le mot clé BEGIN définit la section de traitement procédural. La seconde ligne définit notre traitement, vous pouvez vous en douter ce traitement ne fait absolument rien.

Ici on pourrait y mettre des instructions SQL de type SELECT INSERT UPDATE DELETE

La troisième ligne définit la fin du traitement procédural. On dit plus communément que le mot clé END définit la fin de la section de traitement procédural.

Et en 4 eme ligne vous pouvez apercevoir un slash qui indique à l'IDE d'exécuter le code du dessus.

Voilà c'est tout, maintenant vous pouvez vous faire une idée de ce à quoi ressemble un bloc PL/SQL. Je vous rassure les nôtres seront beaucoup plus garnis !

Il existe en tout et pour tout trois sections... Et si vous voulez en savoir plus je vous invite à découvrir les vidéos suivantes.

La section DECLARE

```
DECLARE
    --Declaration des variables
BEGIN
    NULL; --Traitement
END;
/
```



C'est dans cette SECTION que vous pouvez déclarer les variables et constantes qui seront utilisé dans les autres sections.

Cette section se place toujours au-dessus de la section BEGIN.

Bravo, vous venez de découvrir la seconde SECTION parmi les trois existantes, la dernière section possible et facultative est la section permettant de gérer les erreurs, cette section se nomme EXCEPTION.

La section EXCEPTION

```
DECLARE
    --Déclaration des variables
BEGIN
    NULL; --Traitement
EXCEPTION
    --Section de gestion des erreurs
END;
/
```



Bravo, vous venez de découvrir la seconde SECTION parmi les trois existantes, la dernière section possible et facultative est la section permettant de gérer les erreurs, cette section se nomme EXCEPTION.

Voici comment celle-ci se définit :

*C'est dans cette SECTION que vous allez pouvoir gérer toutes vos erreurs.
Cette section se place toujours après le traitement.*

Conclusion

- Vous savez qu'un bloc PL/SQL est un programme
- Vous savez que les blocs PL/SQL sont exécutés sur le serveur de BDD
- Vous savez qu'un bloc associé à un nom est un bloc interne
- Vous savez qu'un bloc sans nom est un bloc externe ou anonyme
- Vous connaissez les sections DECLARE, BEGIN et EXCEPTION
- Vous savez que les sections DECLARE et EXCEPTION sont facultatives



[INTRO]

Vous voici arrivé au moment de faire le point sur ce module de formation.

[DIAPO]

Maintenant :

- Vous savez qu'un bloc PL/SQL est un programme.
- Vous savez qu'il est exécuté sur le serveur de base de données.
- Vous savez que si un bloc PL/SQL est associé à un nom il est interne sinon externe ou anonyme.
- Vous savez aussi qu'un bloc PL/SQL peut contenir jusqu'à 3

SECTIONS DECLARE BEGIN EXCEPTION en sachant que les sections DECLARE et EXCEPTION sont facultatives.

Et pour finir, vous avez aussi intégré le fait qu'il faut écrire les mots clés SQL et PL/SQL en majuscule.

```
#####  
#####  
# ==> Amélioration Continue <==
```


En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse: #
bit.ly/eni_ac

#####

PL / SQL

Module 4 – La section DECLARE



```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse: #  
#                bit.ly/eni_ac  
#  
#####  
#####
```

Objectifs

- Connaître les types simples de variables
- Savoir déclarer une variable



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

[DIAPO]

Nos objectifs sont :

- Connaître les types simples de variables
- Savoir déclarer une variable

La section DECLARE

Les types de données

Tous les types SQL

BOOLEAN

PLS_INTEGER

Sous-types



[INTRO]

Quels types de variables peut-on déclarer dans un bloc PL/SQL ?

[DIAPO]

C'est très simple, on retrouve déjà tous les types du SQL.

Attention, il y a quelques différences quand même , par exemple la taille maximum de certaines variables est largement supérieure en PL/SQL. Ainsi, un VARCHAR en PL/SQL peut stocker jusqu'à 32767 octets contre 2000 octets pour un VARCHAR en SQL. Pour plus d'informations sur les différences de tailles je vous laisse parcourir la documentation officielle Oracle 12c.

En revanche nous avons de nouveaux types :

- Le type **BOOLEAN**, absent du SQL, est présent en PL/SQL.

La variable de type BOOLEAN peut prendre trois valeurs différentes : TRUE, FALSE ou NULL.

- Le type **PLS_INTEGER** qui est aussi utilisé sous le nom de **BINARY_INTEGER**, il permet de stocker des entiers.

Il est vivement conseillé d'utiliser ce type au lieu de NUMBER.

En effet PLS_INTEGER est un type plus performant en termes de stockage et au niveau des calculs arithmétiques.

Sachez aussi que chez Oracle, chaque type de données est associé à des sous types prédéfinis.

C'est sous types contiennent la même capacité que leur type associé mais avec des contraintes supplémentaires.

Par exemple, PLS_INTEGER à un sous type nommé POSITIVE qui permet de stocker une valeur obligatoirement positive...

Une fois de plus, je vous invite à parcourir la documentions officielle Oracle si vous voulez plus d'informations au sujet des sous types prédéfinis.

Et pour finir je voudrais rajouter une chose concernant les sous types.

Il existe aussi des sous-types utilisateurs, et comme vous pouvez le deviner, ce sont des sous types définis par l'utilisateur.

La section DECLARE

La déclaration d'une variable

```
nom_de_la_variable [CONSTANT] TYPE [NOT NULL] [:= expression];
```



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:                #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

[INTRO]

Comment déclarer une variable ?

Et bien c'est très simple :

[DIAPO]

nom_de_la_variable [CONSTANT] TYPE [NOT NULL] [:= expression];

Sur cette diapo vous pouvez découvrir la syntaxe de déclaration d'une variable.

Tout d'abord on doit écrire le nom de la variable en respectant bien entendu les conventions de nommage c'est-à-dire nommé la variable avec un nom explicite et en snake_case.

Ensuite vous pouvez voir le mot clé `CONSTANT` en majuscule, celui-ci se trouve entre crochet, cela indique qu'il est optionnel.

Donc si vous voulez que votre variable soit une constante vous le mettez sinon si vous ne voulez pas que ce soit une constante vous ne le mettez pas.

Puis on doit définir le `TYPE` de la variable. (`CHAR INT` etc.)

Viens ensuite le mot clé `NOT NULL`. `NOT NULL` est à utiliser si l'on veut interdire la valeur null à la variable, ce mot clé est lui aussi optionnel car comme vous pouvez le voir il est entre crochet.

Et pour finir nous avons la possibilité de définir une valeur par default a la variable grâce au `:=` suivi de la valeur voulu.

Voilà, je vous laisse quelques secondes pour digérer cette syntaxe...

Maintenant nous allons voir quelques exemples de déclarations

La section DECLARE

Des exemples de déclarations

```
DECLARE
    compteur PLS_INTEGER;
    maximum CONSTANT PLS_INTEGER := 500;
    resultat BOOLEAN NOT NULL := TRUE;
    prenom VARCHAR2(30) := 'Anthony';
BEGIN
    NULL; --Traitement
END;
/
```



[DIAPO]

La première variable se nomme compteur et elle est de type PLS_INTEGER.

compteur PLS_INTEGER;

La seconde est une constante nommée maximum de type PLS_INTEGER et qui est égale à 500 ;

maximum CONSTANT PLS_INTEGER := 500;

La troisième se nomme résultat, elle est de type boolean, elle ne peut pas avoir la valeur null et est égale à true.

resultat BOOLEAN NOT NULL := TRUE;

Et pour finir la quatrième se nomme prenom, elle est de type VARCHAR2 et est égale à 'Negan'.

prenom VARCHAR2(30) := 'Negan';

Conclusion

- Vous connaissez les types simples de variables
- Vous savez déclarer une variable
- Vous avez découvert les types BOOLEAN et PLS_INTEGER
- Vous savez qu'il existe des sous-types
- Vous savez que vous pouvez créer vos propres sous types



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse: #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

[DIAPO]

Vous voici arrivée à la fin de ce module et pour conclure on peut maintenant dire que :

- Vous connaissez les types simples de variables
- Vous savez déclarer une variable
- Vous avez découvert les types BOOLEAN et PLS_INTEGER
- Vous savez qu'il existe des sous-types
- Vous savez que vous pouvez créer vos propres sous types

PL / SQL

Module 5 – La section BEGIN



```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:                 #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

Objectifs

- Savoir utiliser des variables
- Savoir utiliser des structures de contrôle
- Savoir utiliser des types complexes
- Savoir afficher des messages pour déboguer



[DIAPO]

Les objectifs de ce module sont les suivants :

- Savoir utiliser des variables
- Savoir utiliser des structures de contrôle
- Savoir utiliser des types complexes
- Savoir afficher des messages pour déboguer

La section BEGIN

Les commentaires de code

```
/* Je suis  
    un commentaire sur  
    plusieurs lignes */  
  
--Je suis un commentaire sur une ligne
```



[DIAPO]

C'est une syntaxe courante. Slash Etoile pour commenter plusieurs lignes et tiret tiret pour commenter une seule ligne.

Félicitation vous savez comment commenter du code.

Mais ce n'est pas une raison pour en mettre à outrance, un code trop commenté reflète souvent un problème de complexité.

Normalement un code avec des noms de variables explicites est censé s'autocommenter.

Autre chose, ne laisser pas du code mort en commentaire.

Le code qui ne sert plus à rien on le supprime ! Alors quand vous faites de la maintenance veuillez laisser le code aussi propre que vous l'avez trouvé.

Merci.

La section BEGIN

L'affectation

:=
INTO

```
nombre_de_mots := 14540;  
  
SELECT count(*) INTO total FROM regions;
```



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

[DIAPO]

Concept très simple dans cette vidéo, l'affectation.
Il existe deux manières de faire :

Manière numéro 1 :

Pour affecter une valeur bien spécifique dans une variable on va utiliser le := comme vous pouvez le voir dans l'exemple.

Manière numéro 2 :

Pour mettre la valeur d'une requête dans une variable on va utiliser le INTO. Dans l'exemple vous pouvez voir que ma requête ne retourne qu'un seul résultat de type NUMBER donc je peux le mettre dans ma variable nommée total.

Félicitation vous savez comment affecter une valeur à une variable scalaire. Une variable scalaire est une variable qui enregistre une valeur simple telles qu'une chaîne de caractères, un entier etc...

La section BEGIN

Le package DBMS_OUTPUT

DBMS_OUTPUT pour déboguer

```
DBMS_OUTPUT.PUT_LINE('Valeur de la variable nommée code_agence : ' || code_agence );
```

La commande **SET SERVEROUTPUT ON** active les fonctions du package DBMS_OUTPUT.



Le package DBMS_OUTPUT– Vidéo 4

Concept super simple dans cette vidéo, l’affichage de messages.

Un seul nom à connaître DBMS_OUTPUT, c’est le nom du package qui nous permet d’afficher des messages en sortie de script.

Ce package est surtout utiliser pour déboguer votre application et analyser le dérouler de votre code.

Comme vous pouvez le voir dans cet exemple on doit utiliser le nom du package avec la fonction PUT_LINE.

A l’intérieur du PUT_LINE vous pouvez mettre votre texte mais aussi la valeur de vos variables.

Ici j’affiche « valeur de la variable nommée code_agence puis la valeur de la variable »

Vous pouvez voir que la concaténation se fait avec une double pipe.

Attention !! Pour que l’affichage fonctionne Vous devez activer les fonctionnalités du package DBMS_OUTPUT grâce à la commande SET SERVEROUTPUT ON ; cette commande est à mettre au-dessus de votre bloc anonyme.

Avant de finir je voudrai vous alerter sur quelques bonnes pratiques :

L’écriture d’un message à un coût non négligeable.

Lors d’une erreur, le log doit fournir toute les informations nécessaires concernant l’erreur.

Mais n'hésitez surtout pas à utiliser ces fonctionnalités lors de vos dev cela va énormément vous aider !!

La section BEGIN

Le traitement conditionnel IF ... THEN ... ELSE

```
DECLARE
    couleur_drapeau VARCHAR(30) := 'VERT';
BEGIN
    IF couleur_drapeau = 'ROUGE' THEN
        DBMS_OUTPUT.PUT_LINE('Baignade interdite');
    ELSIF couleur_drapeau = 'ORANGE' THEN
        DBMS_OUTPUT.PUT_LINE('Baignade dangereuse');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Baignade autorisée youhouuu ! :');
    END IF;
END;
```



L1 :

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

L2 :

On y déclare une variable nommée couleur_drapeau à laquelle on affecte la valeur VERT

L3 :

On arrive ensuite dans la section BEGIN qui contient le traitement.

Dans ce traitement on fait des tests sur la variable couleur_drapeau.

L4 :

On teste si couleur_drapeau est égal à rouge grâce au IF et THEN

Ce n'est pas le cas donc on n'exécutera pas la ligne du dessous qui affiche 'Baignade interdite'

L6 :

On arrive sur le ELSIF qui permet de faire une autre comparaison.

Est-ce que couleur_drapeau est égal à ORANGE. NON

Ce n'est pas le cas donc on n'exécutera pas la ligne du dessous qui affiche 'Baignade dangereuse'

L8 :

On passe donc sur le ELSE qui sera le traitement par défaut.

L9 :

On affiche donc 'Baignade autorisée youhouuuu' ;

L10 :

Et on arrive à la fin de notre structure de contrôle.

L11 :

Puis la fin de notre BLOC PL/SQL

La section BEGIN

Le traitement conditionnel CASE avec valeur

```
DECLARE
    couleur_drapeau VARCHAR(30) := 'VERT';
BEGIN
    CASE couleur_drapeau
        WHEN 'ROUGE' THEN
            DBMS_OUTPUT.PUT_LINE('Baignade interdite !!');
        WHEN 'ORANGE' THEN
            DBMS_OUTPUT.PUT_LINE('Attention, la mer est dangereuse');
        WHEN 'VERT' THEN
            DBMS_OUTPUT.PUT_LINE('Tous à l'eau !!');
        WHEN 'NOIR' THEN
            DBMS_OUTPUT.PUT_LINE('Marée noire');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Drapeau non repertorié');
        END CASE;
END;
```



```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:                 #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

Le traitement conditionnel CASE avec valeur – Vidéo 6

Le PL/SQL est un langage procédural, on a donc la possibilité de faire des traitements conditionnels

Il existe le célèbre contrôle CASE avec valeur!! présentation !

[DIAPO 7]

L1 :

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

L2 :

On y déclare une variable nommée couleur_drapeau à laquelle on affecte la valeur VERT

L3 :

On arrive ensuite dans la section BEGIN qui contient le traitement.

Dans ce traitement on fait des tests sur la variable couleur_drapeau.

L4 :

Derrière le CASE on met la variable que l'on veut tester.

L5 :

Et pour tester sa valeur on utilise le WHEN THEN.

Est-ce que couleur_drapeau est égal à ROUGE. NON

L7 :

On passe donc au CASE suivant.

Est-ce que couleur_drapeau est égal à ORANGE. NON

L9 :

On passe donc au CASE suivant.

Est-ce que couleur_drapeau est égal à VERT. OUI

L10 :

On exécute donc le traitement se trouvant en dessous.

Le traitement affiche Tous à l'eau !!!

L15 :

Ensuite on quitte le CASE sans tester les autres valeurs ;

L16 :

Puis on quitte notre bloc.

La section BEGIN

Le traitement conditionnel CASE avec condition

```
DECLARE
    couleur_drapeau VARCHAR(30) := 'VERT';
BEGIN
    CASE
        WHEN couleur_drapeau = 'VERT' THEN
            DBMS_OUTPUT.PUT_LINE('Le drapeau est vert.');
```

```
        WHEN couleur_drapeau = 'ORANGE' THEN
            DBMS_OUTPUT.PUT_LINE('Le drapeau est orange.');
```

```
        WHEN couleur_drapeau = 'ROUGE' THEN
            DBMS_OUTPUT.PUT_LINE('Le drapeau est rouge.');
```

```
        ELSE
            DBMS_OUTPUT.PUT_LINE('Je ne connais pas cette couleur.');
```

```
    END CASE;
END;
```



Le traitement conditionnel Condition avec condition – Vidéo 7

Le PL/SQL est un langage procédural, on a donc la possibilité de faire des traitements conditionnels

Il existe le célèbre contrôle CASE avec condition !! présentation !

[DIAPHO 8]

L1 :

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

L2 :

On y déclare une variable nommée couleur_drapeau à laquelle on affecte la valeur VERT

L3 :

On arrive ensuite dans la section BEGIN qui contient le traitement.

Dans ce traitement on fait des tests sur la variable couleur_drapeau.

L4 :

Derrière le CASE on ne met rien.

L5 :

Les conditions sont positionnées dans les WHEN THEN

Est-ce que couleur_drapeau est égal à VERT. OUI

L6 :

On exécute donc le traitement se trouvant en dessous.

Le traitement affiche Le drapeau est vert !!!

L15 :

Ensuite on quitte le CASE sans tester les autres conditions;

L16 :

Puis on quitte notre bloc .

La section BEGIN

Le traitement répétitif LOOP

```
DECLARE
    index_loop INTEGER := 0;
BEGIN
    LOOP
        IF index_loop = 3 THEN
            EXIT;
        ELSE
            DBMS_OUTPUT.PUT_LINE(index_loop);
        END IF;
        index_loop := index_loop + 1;
    END LOOP;
END;
```



Le traitement répétitif LOOP – Vidéo 8

Le PL/SQL est un langage procédural, on a donc la possibilité de faire des traitements répétitifs

Il existe le traitement LOOP qui itère à l'infini jusqu'à ce qu'il rencontre le mot clé EXIT.

[DIAPHO 9]

Encadré ligne 2

On déclare une variable nommée INDEX_LOOP.

On affecte à INDEX_LOOP la valeur 0 ;

Encadré ligne 3

On arrive sur la section BEGIN qui contient notre traitement.

Encadré ligne 4

On définit notre traitement itératif grâce au mot clé LOOP

On va donc itérer à l'infini jusqu'à ce que l'on rencontre le mot clé EXIT.

Encadré ligne 5 (IF index_loop = 3 THEN)

Nous voici à l'intérieur de notre traitement itératif.

On teste index_loop pour savoir s'il est égal à 3

Ce n'est pas le cas

Encadré ligne 7 (ELSE)

On arrive sur le ELSE

Encadré ligne 8

Et on exécute le traitement du ELSE.

Ce traitement affiche la valeur de la variable index_loop.
 Index_loop est actuellement égal à 0 ;
 Encadré ligne 10
 On incrémente index_loop qui passe de 0 à 1.
 Encadré ligne 5 (IF index_loop = 3 THEN)
 On teste index_loop pour savoir s'il est égal à 3
 Ce n'est pas le cas
 Encadré ligne 7 (ELSE)
 On arrive sur le ELSE
 Encadré ligne 8
 Et on exécute le traitement du ELSE.
 Ce traitement affiche la valeur de la variable index_loop.
 Index_loop est actuellement égal à 1 ;
 Encadré ligne 10
 On incrémente index_loop qui passe de 1 à 2.
 Encadré ligne 5 (IF index_loop = 3 THEN)
 On teste index_loop pour savoir s'il est égal à 3
 Ce n'est pas le cas
 Encadré ligne 7 (ELSE)
 On arrive sur le ELSE
 Encadré ligne 8
 Et on exécute le traitement du ELSE.
 Ce traitement affiche la valeur de la variable index_loop.
 Index_loop est actuellement égal à 2 ;
 Encadré ligne 10
 On incrémente index_loop qui passe de 2 à 3.
 Encadré ligne 5 (IF index_loop = 3 THEN)
 On teste index_loop pour savoir s'il est égal à 3
 C'est le cas !
 Encadré ligne 6
 On exécute le traitement du IF.
 C'est l'instruction EXIT.
 Encadré ligne 11
 Cela signifie que l'on quitte le traitement itératif
 Encadré ligne 12
 Puis on quitte le bloc PL/SQL

La section BEGIN

Le traitement répétitif FOR

```
BEGIN
  FOR i IN 0..3 LOOP
    DBMS_OUTPUT.PUT_LINE(i);
  END LOOP;
END;
```



```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
# cette adresse:               #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

Le PL/SQL est un langage procédural, on a donc la possibilité de faire des traitements répétitifs

Il existe le traitement FOR qui itère à un nombre de fois défini ! présentation

[DIAPO 10]

Encadré ligne 1 (BEGIN)

On passe par la section BEGIN qui définit le traitement.

Encadré ligne 2 (FOR i IN 0.. 3 LOOP)

On arrive sur la définition de la boucle for, ici on indique qu'il va y avoir 4

itérations ou la variable i aura les valeurs 0,1,2 et 3.

Il n'est pas nécessaire de définir i dans la section DECLARE

Encadré ligne 3 DBMS_OUTPUT.PUT_LINE(i) ;

Le traitement affiche la valeur de i qui est à 0 .

Encadré ligne 2 (FOR i IN 0.. 3 LOOP)

On retourne au début du traitement

Encadré ligne 3 DBMS_OUTPUT.PUT_LINE(i) ;

Le traitement affiche la valeur de i qui est à 1 .

Encadré ligne 2 (FOR i IN 0.. 3 LOOP)

On retourne au début du traitement

Encadré ligne 3 DBMS_OUTPUT.PUT_LINE(i) ;

Le traitement affiche la valeur de i qui est à 2.

Encadré ligne 2 (FOR i IN 0..3 LOOP)

On retourne au début du traitement

Encadré ligne 3 DBMS_OUTPUT.PUT_LINE(i);

Le traitement affiche la valeur de i qui est à 3 .

Encadré ligne 4 (END LOOP)

i a atteint sa valeur maximale on quitte le traitement itératif

Encadré ligne 5 (END)

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

Le traitement répétitif WHILE

```
DECLARE
    index_while INTEGER := 0;
BEGIN
    WHILE(index_while<3) LOOP
        DBMS_OUTPUT.PUT_LINE(index_while);
        index_while := index_while + 1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('FIN : index_while = ' || index_while);
END;
```



Le PL/SQL est un langage procédural, on a donc la possibilité de faire des traitements répétitifs

Il existe le célèbre traitement WHILE qui itère tant qu'une condition retourne true.

[DIAPO 11]

Encadré ligne 1 (DECLARE)

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 2 (index_while INTEGER := 0 ;)

On définit une variable nommée index_while de TYPE NUMBER et qui est égale à 1.

Encadré ligne 3 (BEGIN)

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 4 (WHILE(index_while<3) LOOP)

On arrive sur le traitement répétitif WHILE.

On indique que tant que la variable index_while est inférieur à 3 on continue à itérer .

Encadré ligne 5 (DBMS_OUTPUT.PUT_LINE(index_while))

On débute le traitement de la boucle while.

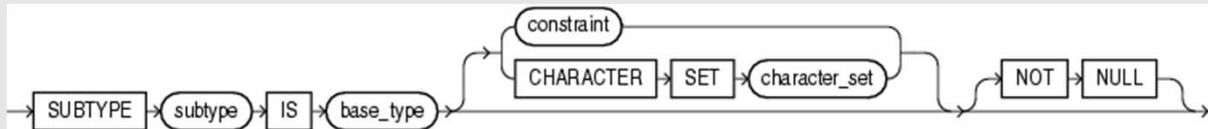
Le traitement affiche la valeur de la variable index_while.

index_while est égale à 0;
 Encadré ligne 6 (index_while := index_while + 1)
 Ensuite on incrémente la variable index_while .
 index_while est égal à 1 ;
 Encadré ligne 4 (WHILE(index_while<3) LOOP)
 On retourne au début du traitement.
 Le traitement vérifie que la variable index_while est bien inférieure à 3.
 C'est le cas donc on réexécute le traitement de la boucle while.
 Encadré ligne 5 (DBMS_OUTPUT.PUT_LINE(index_while))
 Le traitement affiche la valeur de la variable index_while.
 index_while est égale à 1;
 Encadré ligne 6 (index_while := index_while + 1)
 Ensuite on incrémente la variable index_while.
 index_while est égal à 2;
 Encadré ligne 4 (WHILE(index_while<3) LOOP)
 On retourne au début du traitement.
 Le traitement vérifie que la variable index_while est bien inférieure à 3.
 C'est le cas donc on réexécute le traitement de la boucle while.

Encadré ligne 5 (DBMS_OUTPUT.PUT_LINE(index_while))
 Le traitement affiche la valeur de la variable index_while.
 index_while est égale à 2;
 Encadré ligne 6 (index_while := index_while + 1)
 Ensuite on incrémente la variable index_while .
 index_while est égal à 3 ;
 Encadré ligne 4 (WHILE(index_while<3) LOOP)
 On retourne au début du traitement.
 Le traitement vérifie que la variable index_while est bien inférieure à 3.
 Ce n'est pas le cas car maintenant index_while est égale à 3
 Encadré ligne 7 (END LOOP)
 Donc on quitte le traitement répétitif.
 Encadré ligne 8 (DBMS_OUTPUT.PUT_LINE('FIN' || index_while))
 On affiche ensuite la valeur de index_while
 Encadré ligne 9 (END;)
 Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

Les sous-types



Le PL/SQL autorise l'utilisateur à définir ses propres sous-types.

Ce sont des types de base avec une contrainte particulière.

Il existe des sous type prédéfinie par Oracle.

Par exemple, il existe un sous type nommé `POSITIVE` qui est un entier refusant des valeurs négatives.

Mais comme je vous l'ai dit vous pouvez aussi définir vos propre sous types.

[Voici la documentation Oracle qui permet de définir un sous type.](#)

[Commentez la syntaxe.](#)

Je vous remontre la syntaxe sous une autre forme que j'ai choisi afin de bien bien comprendre.

La section BEGIN

Les types définis par l'utilisateur

```
SUBTYPE nom_sous_type IS type_de_base[(contrainte)][NOT NULL];
```



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

[Commentez la syntaxe](#)

Et rien de mieux qu'un exemple pour maîtriser le sujet.

La section BEGIN

Exemple de types définis par l'utilisateur

```
SET SERVEROUTPUT ON;

DECLARE
    SUBTYPE salaire IS NUMBER(4);
    SUBTYPE date_nn IS DATE NOT NULL;

    premiere_annee salaire := 2000;
    premier_jour date_nn := SYSDATE;

BEGIN

    DBMS_OUTPUT.PUT_LINE(premiere_annee);
    DBMS_OUTPUT.PUT_LINE(premier_jour);

END;
```



Encadré ligne 1

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 4

On définit un sous type nommé salaire dont le type de base est un NUMBER.
Sa contrainte est que sa taille maximale en chiffre est 4.
Sa valeur maximale est donc 9999.

Encadré ligne 5

On définit un sous type nommé date_nn dont le type de base est un DATE.
Sa contrainte est que les variables de ce type ne peuvent être NULL.
Sa valeur maximale est donc 9999.

Encadré ligne 7

On crée une variable nommé premiere_annee de type salaire.

On lui affecte une valeur égale à 2000.

Cela est possible car 2000 est inférieure à 9999 ;

Encadré ligne 8

Ensuite on crée une variable nommée premier_jour de type date_nn.
On lui donne la date du jour grâce à SYSDATE.

Encadré ligne 10

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 12

On affiche premiere_annee.

Encadré ligne 13

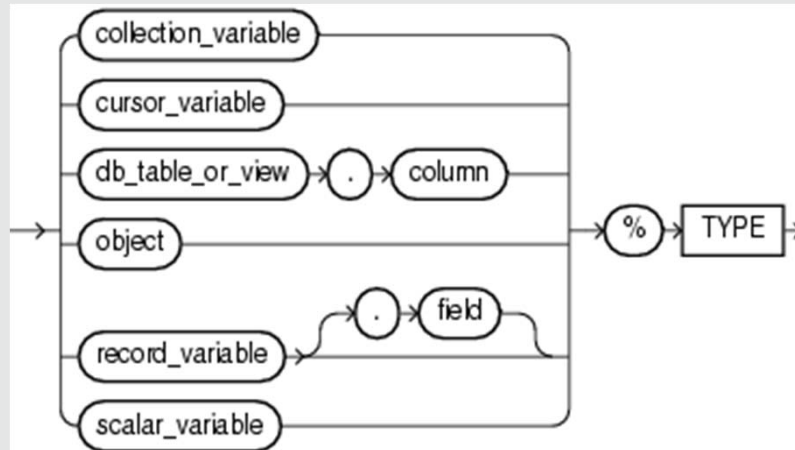
On affiche premier_jour.

Encadré ligne 14

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

L'attribut %TYPE



[DIAPO]

L'attribut %Type permet de déclarer une variable selon la définition :

D'une autre variable précédemment déclarer ou

D'une colonne d'une table

Ou d'une colonne d'une vue existante.

Je vous montre sa syntaxe via la documentation Oracle :

[Commentez la syntaxe.](#)

Et avec ma syntaxe fait maison :

La section BEGIN

L'attribut %TYPE

```
nom_variable nom_table.nom_colonne%TYPE;
```



Commentez la syntaxe.

Mais quoi de mieux qu'un exemple :

La section BEGIN

Exemple d'utilisation de l'attribut %TYPE

```
SET SERVEROUTPUT ON;

DECLARE

    continent regions.region_name%TYPE;

BEGIN

    continent := 'Europe';
    DBMS_OUTPUT.PUT_LINE(continent);

END;
```



Encadré ligne 1

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 5

On déclare une variable nommée continent.

Le type de cette variable sera exactement le même que le type de la colonne region_name de la table regions.

Soit VARCHAR2(50) ;

Encadré ligne 7

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 9

On affecte la valeur 'Europe' à la variable continent

Encadré ligne 10

On affiche la variable continent

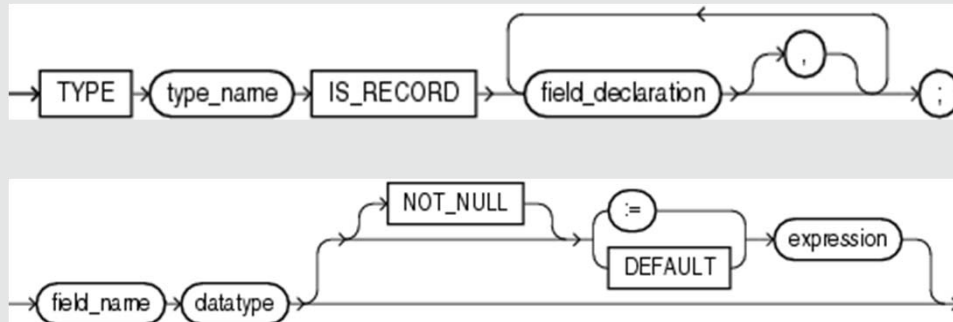
Encadré ligne 12

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

Voilà, c'est terminé.

La section BEGIN

Les enregistrements de type RECORD



Un enregistrement, ou un record en pl/sql, permet de regrouper dans une même type un ensemble d'informations.

Il permet de combiner différents types de données et est défini par l'utilisateur.

Les éléments d'un record sont généralement appelés les champs de l'enregistrement.

Allez ! On va aller voir la syntaxe

[Commentez la syntaxe](#)

Et quoi de mieux qu'un bel exemple pour bien comprendre !

La section BEGIN

Exemple d'utilisation du type RECORD

```
SET SERVEROUTPUT ON;

DECLARE

    TYPE fiche_parking IS RECORD
    (
        prenom VARCHAR2(50),
        nom VARCHAR2(50) NOT NULL := 'XXX',
        nombre NUMBER NOT NULL DEFAULT 1
    );

    fiche1 fiche_parking;

BEGIN

    fiche1.prenom := 'Anthony';
    fiche1.nom := 'Cosson';

    DBMS_OUTPUT.PUT_LINE('Prenom : ' || fiche1.prenom);
    DBMS_OUTPUT.PUT_LINE('Nom : ' || fiche1.nom);
    DBMS_OUTPUT.PUT_LINE('Nombre voiture : ' || fiche1.nombre);

END;
```



#####

==> Amélioration Continue <==

#

En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:

#

bit.ly/eni_ac

#

#####

[DIAPO]

Encadré ligne 1

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer
nos variables

Encadré ligne 5

On définit un nouveau type nommé fiche_parking

Encadré ligne 6

La parenthèse définit le début de la définition du type

Encadré ligne 7

Une variable de type `fiche_parking` contiendra un prenom de type `VARCHAR(50)`.

Encadré ligne 8

Une variable de type `fiche_parking` contiendra un nom de type `VARCHAR(50)`.

Par default sa valeur sera xxx et ne pourra pas être nul.

Encadré ligne 9

Une variable de type `fiche_parking` contiendra un nombre de type `NUMBER`

Par default sa valeur sera 1 et ne pourra pas être nul.

Encadré ligne 10

La parenthèse point-virgule définit la fin de la définition du type

Encadré ligne 12

Maintenant que l'on a défini notre type.

On va créer une variable de ce type.

Donc la variable `fiche1` est de type `fiche_parking`

Encadré ligne 14

On arrive dans la section `BEGIN` qui contient le traitement.

Encadré ligne 16

On affecte une valeur au prenom de la variable `fiche 1`.

Encadré ligne 17

On affecte une valeur au nom de la variable `fiche 1`.

Encadré ligne 19

On affiche le prenom

Encadré ligne 20

On affiche le nom

Encadré ligne 21

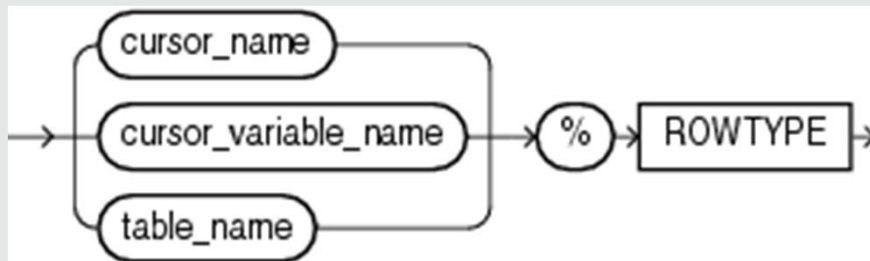
On affiche le nombre

Encadré ligne 23

Et enfin le `END` qui indique que l'on arrive à la fin du bloc `PL/SQL`

La section BEGIN

L'attribut %ROWTYPE



[DIAPO]

L'attribut ROWTYPE permet de déclarer un record ou un enregistrement basé sur la liste de colonnes d'une table, d'une vue ou même d'un autre RECORD.

Les noms et types de données des champs du record sont identiques à ceux des colonnes de la table ou de la vue.

Allons voir la syntaxe d'Oracle :

[Commentez la syntaxe](#)

Et pour finir un petit exemple :

La section BEGIN

Exemple d'utilisation de l'attribut %ROWTYPE

```
SET SERVEROUTPUT ON;

DECLARE

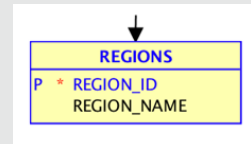
    region_rec regions%ROWTYPE;

BEGIN

    region_rec.region_id = 5;
    region_rec.region_name = 'Antarctique';

    DBMS_OUTPUT.PUT_LINE('Id de la region : ' || region_rec.region_id);
    DBMS_OUTPUT.PUT_LINE('Nom de la region : ' || region_rec.region_name);

END;
```



Encadré ligne 1

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 5

On définit une variable nommée region_rec de type RECORD qui contiendra autant de variables qu'il y a de colonnes dans la table regions.

Ainsi la variable region_rec contient une variable REGION_ID de type number et une variable region_name de type varchar(50).

Encadré ligne 7

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 9

On affecte la valeur 5 à la variable region_id de region_rec.

Encadré ligne 10

On affecte la valeur Antarctique à la variable region_name de region_rec.

Encadré ligne 12

On affiche l'id de la region

Encadré ligne 13

Puis le nom.

Encadré ligne 15

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

Les collections

TYPE	Nombre d'éléments	Indexation	Doit être initialisé
INDEX BY TABLE (Tableau associatif (clé/valeur))	Illimité	Alphanumérique	Non
NESTED TABLE (Tableau imbriqué)	Illimité	Entier	Oui
VARRAY (Taille variable)	Limité	Entier	Non



Les collections PL/SQL permettent de regrouper des éléments de même type.

Il existe trois types de collections avec chacune ses spécificités.

- Il y a le type de collection INDEX BY TABLE qui permet d'enregistrer des informations sous forme de clé/valeur.

- Il y a le type de collection NESTED TABLE qui dispose de certaine souplesse concernant sa taille.

- Il y a le type de collection VARRAY qui dispose d'une taille fixe et un accès aux données plus rapide.

Penchons-nous sur le cas de la collection index by table

[DIAPO 23]

Elle permet d'enregistrer des informations sous forme de paires clé-valeur.

La clé peut être numérique ou alpha numérique.

Sa taille est dynamique

Elle est généralement utilisée pour stocker des données de manière temporaire.

Comment l'utiliser ?

[DIAPO 24]

Encadré ligne 1 (SET SERVEROUTPUT ON)

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3 (DECLARE)

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer

nos variables

Encadré ligne 5 (TYPE population IS TABLE OF NUMBER BY VARCHAR(50))

Déclaration d'un nouveau TYPE de collection index by table nommé population qui permet de stocker des NUMBER et qui est indexé par une clé de type varchar(50)

Encadré ligne 7 (population_ville population)

Déclaration d'une variable nommée population ville de type population

Encadré ligne 9 (ville varchar(50))

Déclaration d'une variable nommée ville de type varchar(50)

Encadré ligne 10 (BEGIN)

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 12 (population_ville('Soliers') := 2151)

On ajoute une nouvelle valeur à la collection population_ville.

Cette valeur est égale à 2151 et sa clé est égale à Soliers

Encadré ligne 13 (population_ville('Caen') := 106538)

On ajoute une nouvelle valeur à la collection population_ville.

Cette valeur est égale à 2151 et sa clé est égale à Soliers

Encadré ligne 14 (population_ville('Limoges') := 134577)

On ajoute une nouvelle valeur à la collection population_ville.

Cette valeur est égale à 134577 et sa clé est égale à Limoges

Encadré ligne 15 (population_ville('Rennes') := 213454)

On ajoute une nouvelle valeur à la collection population_ville.

Cette valeur est égale à 213454 et sa clé est égale à Rennes

Encadré ligne 17 (ville := population_ville.FIRST)

Les collections ont des méthodes.

Par exemple ici on utilise la méthode FIRST pour récupérer le premier élément de la collection.

Ici on récupère Soliers.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)

Grâce à cette boucle while on va parcourir et afficher toute les villes de la liste.

Tant que l'on arrive à récupérer des villes dans la collection on itère....

Encadré ligne 20&21 (DBMS...)

On affiche les informations de la ville récupérer

Encadré ligne 23 (ville := population_ville.NEXT(ville))

Puis on récupère la suivante.

S'il n'y a pas de ville suivante la méthode NEXT retourne NULL.

Dans ce cas on quitte la boucle WHILE.

Ici on récupère la valeur Caen.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)

On test

Encadré ligne 20&21 (DBMS...)

On affiche

Encadré ligne 23 (ville := population_ville.NEXT(ville))

On récupère la valeur suivante.

Ici on récupère la valeur Limoges.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)

On test

Encadré ligne 20&21 (DBMS...)

On affiche

Encadré ligne 23 (ville := population_ville.NEXT(ville))

On récupère la valeur suivante.

A un moment on récupère null.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)

On fait le test dans le while.

Et là c'est le moment de quitter la boucle.

Encadré ligne 24 (END LOOP)

Fin de boucle while identifié par le END LOOP.

Encadré ligne 25 (END)

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

La collection de type INDEX BY TABLE

Ensemble de paires clé-valeur
Clé numérique ou alpha numérique
Taille dynamique
Stockage temporaire de données



Elle permet d'enregistrer des informations sous forme de paires clé-valeur.

La clé peut être numérique ou alpha numérique.

Sa taille est dynamique

Elle est généralement utilisée pour stocker des données de manière temporaire.

Comment l'utiliser ?

[DIAPO 24]

Encadré ligne 1 (SET SERVEROUTPUT ON)

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3 (DECLARE)

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 5 (TYPE population IS TABLE OF NUMBER BY VARCHAR(50))

Déclaration d'un nouveau TYPE de collection index by table nommé population qui permet de stocker des NUMBER et qui est indexé par une clé de type varchar(50)

Encadré ligne 7 (population_ville population)

Déclaration d'une variable nommée population_ville de type population

Encadré ligne 9 (ville varchar(50))

Déclaration d'une variable nommée ville de type varchar(50)

Encadré ligne 10 (BEGIN)

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 12 (population_ville('Soliers') := 2151)

On ajoute une nouvelle valeur à la collection population_ville.
 Cette valeur est égale à 2151 et sa clé est égale à Soliers

Encadré ligne 13 (population_ville('Caen') := 106538)
 On ajoute une nouvelle valeur à la collection population_ville.
 Cette valeur est égale à 2151 et sa clé est égale à Soliers

Encadré ligne 14 (population_ville('Limoges') := 134577)
 On ajoute une nouvelle valeur à la collection population_ville.
 Cette valeur est égale à 134577 et sa clé est égale à Limoges

Encadré ligne 15 (population_ville('Rennes') := 213454)
 On ajoute une nouvelle valeur à la collection population_ville.
 Cette valeur est égale à 213454 et sa clé est égale à Rennes

Encadré ligne 17 (ville := population_ville.FIRST)
 Les collections ont des méthodes.
 Par exemple ici on utilise la méthode FIRST pour récupérer le premier élément de la collection.
 Ici on récupère Soliers.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)
 Grâce à cette boucle while on va parcourir et afficher toute les villes de la liste.
 Tant que l'on arrive à récupérer des villes dans la collection on itère....

Encadré ligne 20&21 (DBMS...)
 On affiche les informations de la ville récupérer

Encadré ligne 23 (ville := population_ville.NEXT(ville))
 Puis on récupère la suivante.
 S'il n'y a pas de ville suivante la méthode NEXT retourne NULL.
 Dans ce cas on quitte la boucle WHILE.
 Ici on recupère la valeur Caen.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)
 On test

Encadré ligne 20&21 (DBMS...)
 On affiche

Encadré ligne 23 (ville := population_ville.NEXT(ville))
 On récupère la valeur suivante.
 Ici on récupère la valeur Limoges.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)
 On test

Encadré ligne 20&21 (DBMS...)
 On affiche

Encadré ligne 23 (ville := population_ville.NEXT(ville))
 On récupère la valeur suivante.
 A un moment on récupère null.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)
 On fait le test dans le while.
 Et là c'est le moment de quitter la boucle.

Encadré ligne 24 (END LOOP)
 Fin de boucle while identifié par le END LOOP.

Encadré ligne 25 (END)

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

Exemple de collection INDEX BY TABLE

```
SET SERVEROUTPUT ON;

DECLARE

    TYPE population IS TABLE OF NUMBER INDEX BY VARCHAR(50);

    population_ville population;

    ville VARCHAR2(50);
BEGIN

    population_ville('Soliers') := 2151;
    population_ville('Caen') := 106538;
    population_ville('Limoges') := 134577;
    population_ville('Rennes') := 213454;

    ville := population_ville.FIRST;

    WHILE ville IS NOT NULL LOOP
        DBMS_Output.PUT_LINE('La population de ' || ville
        || ' est de ' || TO_CHAR(population_ville(ville)) || ' habitants');

        ville := population_ville.NEXT(ville);
    END LOOP;
END;
```



Encadré ligne 1 (SET SERVEROUTPUT ON)

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3 (DECLARE)

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 5 (TYPE population IS TABLE OF NUMBER BY VARCHAR(50))

Déclaration d'un nouveau TYPE de collection index by table nommé population qui permet de stocker des NUMBER et qui est indexe par une clé de type varchar(50)

Encadré ligne 7 (population_ville population)

Déclaration d'une variable nommée population ville de type population

Encadré ligne 9 (ville varchar(50))

Déclaration d'une variable nommée ville de type varchar(50)

Encadré ligne 10 (BEGIN)

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 12 (population_ville('Soliers') := 2151)

On ajoute une nouvelle valeur à la collection population_ville.

Cette valeur est égale à 2151 et sa clé est égale à Soliers

Encadré ligne 13 (population_ville('Caen') := 106538)

On ajoute une nouvelle valeur à la collection population_ville.

Cette valeur est égale à 2151 et sa clé est égale à Soliers

Encadré ligne 14 (population_ville('Limoges') := 134577)

On ajoute une nouvelle valeur à la collection population_ville.
 Cette valeur est égale à 134577 et sa clé est égale à Limoges

Encadré ligne 15 (population_ville('Rennes') := 213454)
 On ajoute une nouvelle valeur à la collection population_ville.
 Cette valeur est égale à 213454 et sa clé est égale à Rennes

Encadré ligne 17 (ville := population_ville.FIRST)
 Les collections ont des méthodes.
 Par exemple ici on utilise la méthode FIRST pour récupérer le premier élément de la collection.
 Ici on récupère Soliers.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)
 Grâce à cette boucle while on va parcourir et afficher toute les villes de la liste.
 Tant que l'on arrive à récupérer des villes dans la collection on itère....

Encadré ligne 20&21 (DBMS...)
 On affiche les informations de la ville récupérer

Encadré ligne 23 (ville := population_ville.NEXT(ville))
 Puis on récupère la suivante.
 S'il n'y a pas de ville suivante la méthode NEXT retourne NULL.
 Dans ce cas on quitte la boucle WHILE.
 Ici on recupère la valeur Caen.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)
 On test

Encadré ligne 20&21 (DBMS...)
 On affiche

Encadré ligne 23 (ville := population_ville.NEXT(ville))
 On récupère la valeur suivante.
 Ici on récupère la valeur Limoges.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)
 On test

Encadré ligne 20&21 (DBMS...)
 On affiche

Encadré ligne 23 (ville := population_ville.NEXT(ville))
 On récupère la valeur suivante.
 A un moment on récupère null.

Encadré ligne 19 (WHILE ville IS NOT NULL LOOP)
 On fait le test dans le while.
 Et là c'est le moment de quitter la boucle.

Encadré ligne 24 (END LOOP)
 Fin de boucle while identifié par le END LOOP.

Encadré ligne 25 (END)
 Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

La collection de type NESTED TABLE

Taille dynamique

Index numérique

Données parsemées

Array of Integers										Fixed Upper Bound
321	17	99	407	83	622	105	19	67	278	
x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)	x(8)	x(9)	x(10)	

Nested Table after Deletions										Unbounded →
321		99	407		622	105	19		278	
x(1)		x(3)	x(4)		x(6)	x(7)	x(8)		x(10)	



Les collections PL/SQL permettent de regrouper des éléments de même type.

Il existe trois types de collections avec chacune ses spécificités.

- Il y a le type de collection INDEX BY TABLE qui permet d'enregistrer des informations sous forme de clé/valeur.

- Il y a le type de collection NESTED TABLE qui dispose de certaine souplesse concernant sa taille.

- Il y a le type de collection VARRAY qui dispose d'une taille fixe et un accès aux données plus rapide.

Penchons-nous sur le cas de la collection de type NESTED table

Sa taille est dynamique

Les index sont seulement numériques.

Ses données sont parsemées. Donc l'accès aux données peut être plus long comparé à d'autre collections.

Elle est généralement utilisée quand on ne connaît pas par avance la taille de la collection.

Comment l'utiliser ?

La section BEGIN

Exemple de collection NESTED TABLE

```
SET SERVEROUTPUT ON;

DECLARE

    TYPE tableau_noms_objets IS TABLE OF VARCHAR2(50);

    tableau_musique tableau_noms_objets;

    objet_courant VARCHAR2(50);
BEGIN

    tableau_musique := tableau_noms_objets('Guitare', 'Tambour', 'Batterie', 'Basse');

    tableau_musique.DELETE(2);

    DBMS_OUTPUT.PUT_LINE(tableau_musique(1));
    --DBMS_OUTPUT.PUT_LINE(tableau_musique(2));
    DBMS_OUTPUT.PUT_LINE(tableau_musique(3));

END;
```



Encadré ligne 1 (SET SERVEROUTPUT ON)

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3 (DECLARE)

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 5

Déclaration d'un nouveau TYPE de collection nested table nommé tableau_noms_objets qui permet de stocker des varchar(50) et qui est indexé par des entiers

Encadré ligne 7

Déclaration d'une variable nommée tableau_musique de type tableau_noms_objets

Encadré ligne 9

Déclaration d'une variable de type objet_courant de type varchar2(50) ;

Encadré ligne 10 (BEGIN)

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 12

On affecte notre collection avec une liste d'instruments

Encadré ligne 14 (tableau_musique.DELETE(2))

On supprime l'élément numéro 2.

Ainsi notre collection n'aura plus d'élément en position deux.

Encadré ligne 16

On affiche l'élément 1

Encadré ligne 17

On ne peut pas afficher l'élément 2 car il n'existe pas.

Cela élèverait une erreur.

Encadré ligne 18

On affiche l'élément 3

Encadré ligne 20 (END)

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

La collection de type VARRAY

Taille fixe

Index numérique

Données denses

Array of Integers										Fixed Upper Bound
321	17	99	407	83	622	105	19	67	278	
x(1)	x(2)	x(3)	x(4)	x(5)	x(6)	x(7)	x(8)	x(9)	x(10)	

Nested Table after Deletions										Unbounded →
321		99	407		622	105	19		278	
x(1)		x(3)	x(4)		x(6)	x(7)	x(8)		x(10)	



Les collections PL/SQL permettent de regrouper des éléments de même type.

Il existe trois types de collections avec chacune ses spécificités.

- Il y a le type de collection INDEX BY TABLE qui permet d'enregistrer des informations sous forme de clé/valeur.

- Il y a le type de collection NESTED TABLE qui dispose de certaine souplesse concernant sa taille.

- Il y a le type de collection VARRAY qui dispose d'une taille fixe et un accès aux données plus rapide.

Penchons-nous sur le cas de la collection de type VARRAY

Sa taille est fixe

Les index sont seulement numériques.

Ses données sont denses. Donc l'accès aux données est rapide car toutes les informations sont stockées en mémoire les unes derrière les autres.

Elle est généralement utilisée quand on ne connaît pas par avance la taille de la collection.

Comment l'utiliser ?

La section BEGIN

Exemple de collection VARRAY

```
SET SERVEROUTPUT ON;

DECLARE

    TYPE tableau_couleur IS VARRAY(5) OF VARCHAR2(50);

    france tableau_couleur := tableau_couleur('Bleu','Blanc','Rouge');

BEGIN

    DBMS_OUTPUT.PUT_LINE(france(1));
    DBMS_OUTPUT.PUT_LINE(france(2));
    DBMS_OUTPUT.PUT_LINE(france(3));

END;
```



Encadré ligne 1 (SET SERVEROUTPUT ON)

On active les fonctionnalités du package

DBMS_OUTPUT

Encadré ligne 3 (DECLARE)

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 5 (TYPE tableau_couleur IS VARRAY(5) OF VARCHAR2(50))

Déclaration d'un nouveau TYPE de collection varray nommé tableau_couleur qui permet de stocker des varchar(50) et peut contenir 5 éléments.

Encadré ligne 7 (France tableau_couleur := tableau_couleur('bleu','blanc','rouge'))

Déclaration d'une nouvelle variable nommé France de type tableau_couleur avec trois éléments 'bleu', 'blanc' et 'rouge'

Encadré ligne 9 (BEGIN)

On arrive dans la section BEGIN qui contient le traitement.

Encadré ligne 11 (DBMS_OUTPUT.PUT_LINE(France(1)))

On affiche l'élément 1

Encadré ligne 12 (DBMS_OUTPUT.PUT_LINE(France(2)))

On affiche l'élément 2

Encadré ligne 13 (DBMS_OUTPUT.PUT_LINE(France(3)))

On affiche l'élément 3

Encadré ligne 15 (END)

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

Voilà, et avant de finir voici une liste exhaustive de toutes les méthodes existantes pour manipuler les collections.

[DIAPO 29]

COUNT permet de compter le nombre d'élément dans une collection.

DELETE permet de supprimer un élément dans une collection.

EXISTS permet de savoir si une position dans la collection existe.

EXTEND permet d'augmenter la taille d'une collection.

FIRST permet de récupérer le premier élément de la collection.

LAST permet de récupérer le dernier élément de la collection.

LIMIT permet de retourner le nombre maximum

d'éléments que la collection peut avoir.

PRIOR (n) renvoie le numéro d'index qui précède l'index n dans la collection.

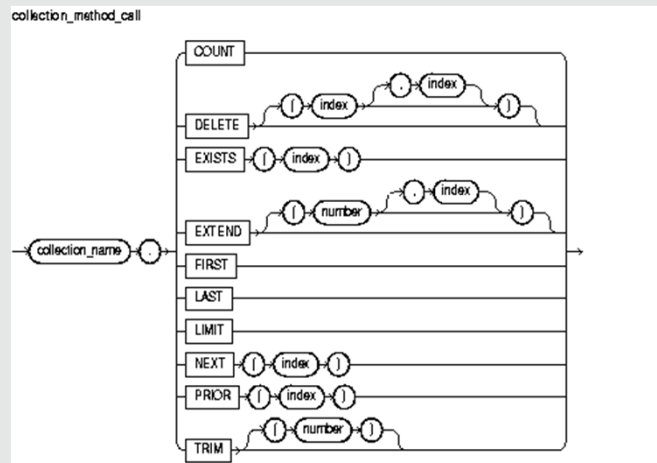
NEXT (n) renvoie le numéro d'index qui succède l'index n.

TRIM supprime un élément de la fin d'une collection.

TRIM (n) supprime n éléments de la fin d'une collection.

La section BEGIN

Méthodes associées aux collections



COUNT permet de compter le nombre d'élément dans une collection.

DELETE permet de supprimer un élément dans une collection.

EXISTS permet de savoir si une position dans la collection existe.

EXTEND permet d'augmenter la taille d'une collection.

FIRST permet de récupérer le premier élément de la collection.

LAST permet de récupérer le dernier élément de la collection.

LIMIT permet de retourner le nombre maximum d'éléments que la collection peut avoir.

PRIOR (n) renvoie le numéro d'index qui précède l'index n dans la collection.

NEXT (n) renvoie le numéro d'index qui succède l'index n.

TRIM supprime un élément de la fin d'une collection.

TRIM (n) supprime n éléments de la fin d'une collection.

La section BEGIN

Le curseur explicite

Un curseur permet de stocker le résultat d'une requête
afin de le traiter ligne par ligne



Parlons CURSEUR !

Un curseur permet de stocker le résultat d'une requête afin de le traiter ligne par ligne.

Il y a plusieurs façons d'utiliser un curseur.

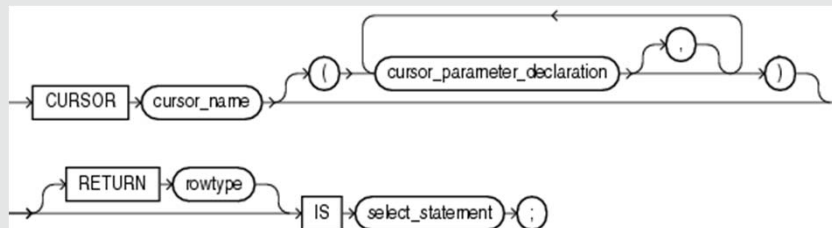
La première façon est la plus ancienne et celle qui laisse le plus de liberté pour des traitements spécifiques.

Les nouvelles manières de faire sont celles qui sont plutôt préconisées grâce à leur facilité d'utilisation.

Voyons déjà la syntaxe pour déclarer un curseur :

La section BEGIN

La déclaration d'un curseur explicite



Voici la syntaxe comme elle est définie dans la documentation Oracle.

Alors maintenant on va voir le fonctionnement des curseurs à l'ancienne !

La section BEGIN

L'utilisation du curseur explicite : 1^{ère} méthode

Fonctionnement :

1. Déclaration du curseur
2. Ouverture du curseur
3. Récupération du résultat ligne par ligne
4. Fermeture du curseur



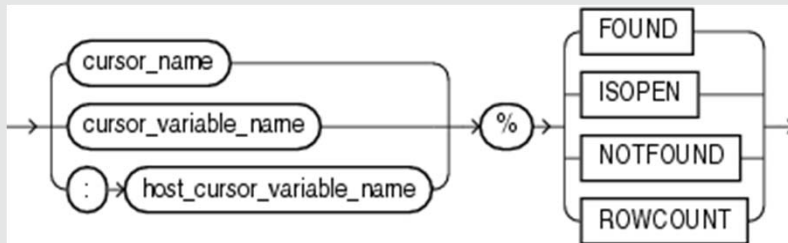
Fonctionnement des « curseurs à l'ancienne »

Cette façon est la plus ancienne et aussi celle qui laisse le plus de liberté pour des traitements spécifiques.

Les curseurs ont des attributs !

La section BEGIN

Les attributs de curseur



Voici les attributs qui peuvent être utilisés sur un curseur.

FOUND retourne vrai si le dernier FETCH à ramener une ligne

ISOPEN retourne vrai si le curseur est ouvert

NOTFOUND retourne vrai si le dernier FETCH n'a pas ramené de ligne.

ROWCOUNT compte le nombre de lignes contenues dans le curseur.

La section BEGIN

L'utilisation du curseur explicite : 1^{ère} méthode

```
SET SERVEROUTPUT ON;

DECLARE
  CURSOR cursor_employees_it IS SELECT * FROM EMPLOYEES WHERE DEPARTMENT_ID = 60;
  employee employees%ROWTYPE;
BEGIN
  OPEN cursor_employees_it;

  LOOP
    FETCH cursor_employees_it INTO employee;
    DBMS_OUTPUT.PUT_LINE(employee.last_name);
    EXIT WHEN cursor_employees_it%NOTFOUND;
  END LOOP;

  CLOSE cursor_employees_it;
END;
```



Encadré ligne 1 (SET SERVEROUTPUT ON)

On active les fonctionnalités du package DBMS_OUTPUT

Encadré ligne 3 (DECLARE)

On passe par la section DECLARE seule et unique section dans laquelle on doit déclarer nos variables

Encadré ligne 4 (CURSOR...)

On déclare notre curseur. Notre curseur contiendra donc le résultat de la requête SELECT * FROM EMPLOYEES WHERE DEPARTMENT_ID = 60 ;

Encadré ligne 6 (employee employees%ROWTYPE)

On déclare une variable de type enregistrement qui sera structuré comme la table employees.

Ca veut dire que cette variable pourra enregistrer la totalité des informations d'un employé

Encadré ligne 7 (BEGIN)

Début de la section qui contient le traitement.

Encadré ligne 8 (OPEN cursor...)

Comme on la verra dans le fonctionnement des curseurs à l'ancienne.

On ouvre le curseur.

Encadré ligne 10 (LOOP)

On utilise une boucle loop qui itérera autant de fois qu'il y a d'enregistrement dans le curseur

Encadré ligne 11 (FETCH)

Permet de mettre le premier enregistrement du resultat de la requete dans la variable employee

Encadré ligne 12 (DBMS)

On affiche le nom de l'employé

Encadré ligne 13 (EXIT WHEN)

S'il n'y a pas d'enregistrement suivant alors on quitte la boucle for...)

Encadré ligne 10 (LOOP)

On retourne sur la boucle loop qui iterera autant de fois qu'il y a d'enregistrement dans le curseur

Encadré ligne 11 (FETCH)

Permet de l'enregistrement suivant du resultat de la requete dans la variable employee

Encadré ligne 12 (DBMS)

On affiche le nom de l'employé

Encadré ligne 13 (EXIT WHEN)

S'il n'y a pas d'enregistrement suivant alors on quitte la boucle for...)

Encadré ligne 10 (LOOP)

On retourne sur la boucle loop qui iterera autant de fois qu'il y a d'enregistrement dans le curseur

Encadré ligne 11 (FETCH)

Permet de l'enregistrement suivant du resultat de la requete dans la variable employee

Encadré ligne 12 (DBMS)

On affiche le nom de l'employé

Encadré ligne 13 (EXIT)

Et donc à un moment il n'y aura plus d'employé à afficher donc plus d'enregistrement suivant

Encadré ligne 14 (END LOOP)

Alors on quittera la boucle loop

Encadré ligne 16 (CLOSE)

On doit alors fermer le curseur

Encadré ligne 17

Et enfin le END qui indique que l'on arrive à la fin du bloc PL/SQL

La section BEGIN

L'utilisation du curseur explicite : 2^{ème} méthode

FOR ... IN ... LOOP



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

La section BEGIN

L'utilisation du curseur explicite : 2^{ème} méthode

```
SET SERVEROUTPUT ON;

DECLARE
    CURSOR cursor_employees_it IS SELECT first_name,last_name FROM EMPLOYEES WHERE DEPARTMENT_ID = 60;
    employee employees%ROWTYPE;
BEGIN

    FOR employee IN cursor_employees_it LOOP
        DBMS_OUTPUT.PUT_LINE(employee.first_name);
    END LOOP;

END;
```



La section BEGIN

Le verrou d'intention FOR UPDATE

Permet de verrouiller la table ou les champs que l'on va mettre à jour à partir du curseur implicite



La section BEGIN

Le verrou d'intention FOR UPDATE

```
SET SERVEROUTPUT ON;

DECLARE
  CURSOR cursor_employees_it IS SELECT * FROM EMPLOYEES WHERE DEPARTMENT_ID = 60 FOR UPDATE;

BEGIN

  FOR employee IN cursor_employees_it LOOP

    UPDATE employees SET last_name = UPPER(last_name) WHERE employee_id = employee.employee_id;
    DBMS_OUTPUT.PUT_LINE(employee.first_name || ' ' || employee.last_name);

  END LOOP;

  COMMIT;
END;
```



Le verrou d'intention FOR UPDATE OF

```
SET SERVEROUTPUT ON;

DECLARE
  CURSOR cursor_employees_it IS SELECT * FROM EMPLOYEES WHERE DEPARTMENT_ID = 60 FOR UPDATE OF last_name;
BEGIN
  FOR employee IN cursor_employees_it LOOP
    UPDATE employees SET last_name = UPPER(last_name) WHERE employee_id = employee.employee_id;
    DBMS_OUTPUT.PUT_LINE(employee.first_name || ' ' || employee.last_name);
  END LOOP;

  COMMIT;
END;
```



La section BEGIN

La clause WHERE CURRENT OF du curseur explicite

Permet de simplifier la clause WHERE
des requêtes de modification



La section BEGIN

Exemple d'utilisation de la clause WHERE CURRENT OF

```
SET SERVEROUTPUT ON;

DECLARE
  CURSOR cursor_employees_it IS SELECT * FROM EMPLOYEES WHERE DEPARTMENT_ID = 60 FOR UPDATE OF last_name;
BEGIN
  FOR employee IN cursor_employees_it LOOP
    UPDATE employees SET last_name = UPPER(last_name) WHERE CURRENT OF cursor_employees_it;
    DBMS_OUTPUT.PUT_LINE(employee.first_name || ' ' || employee.last_name);
  END LOOP;

  COMMIT;

END;
```



La section BEGIN

Le curseur explicite paramétrable

Un curseur peut être paramétrable



La section BEGIN

Exemple de curseur explicite paramétrable

```
SET SERVEROUTPUT ON;

DECLARE
  CURSOR cursor_employees_it(dep NUMBER) IS SELECT * FROM EMPLOYEES WHERE DEPARTMENT_ID = dep FOR UPDATE OF last_name;
BEGIN

  FOR employee IN cursor_employees_it(60) LOOP
    UPDATE employees SET last_name = UPPER(last_name) WHERE CURRENT OF cursor_employees_it;
    DBMS_OUTPUT.PUT_LINE(employee.first_name || ' ' || employee.last_name);
  END LOOP;

  COMMIT;

END;
```



La section BEGIN

L'utilisation du curseur explicite : 3^{ème} méthode

FOR ... IN ... LOOP



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #
```

```
#####  
#####  
Parlons CURSEUR !  
Un curseur permet de stocker le résultat d'une requête afin de le traiter ligne par ligne.  
Il y a plusieurs façons d'utiliser un curseur.  
Voici la plus simple !!
```

On peut utiliser un curseur dans un loop et sans déclarer le curseur.
Cette façon peut être utilisée lorsque la requête du curseur est très courte.
Si la requête est longue cela risque de rendre le code incompréhensible.
Démonstration !

La section BEGIN

L'utilisation curseur explicite : 3^{ème} méthode

```
SET SERVEROUTPUT ON;

BEGIN

  FOR employee IN (SELECT first_name,last_name FROM EMPLOYEES WHERE DEPARTMENT_ID = 60) LOOP

    DBMS_OUTPUT.PUT_LINE(employee.first_name);

  END LOOP;

END;
```



Comme vous pouvez le voir on met directement la requête dans le FOR

La section BEGIN

Le curseur implicite

- C'est un curseur de session déclaré et géré implicitement par PL/SQL.
- Il contient des informations à propos des dernières instructions DML effectuées.
- Utilisation :
 - SQL%FOUND
 - SQL%ROWCOUNT



[INTRO]

C'est un curseur de session déclaré et géré implicitement par PL/SQL.

Il contient des informations à propos des dernières instructions DML effectuées.

[DIAPO]

SQL%FOUND : NULL si aucune requête DML n'a été exécutée, TRUE si la dernière requête retourne ou modifie au moins une ligne sinon FALSE.

SQL%ROWCOUNT : Retourne le nombre de lignes retournées ou modifiées par la dernière requête DML.

La section BEGIN

Bloc anonyme

Démonstration



[Afficher tous les départements](#)

Conclusion

- Vous connaissez des types de données complexes
- Vous savez définir et manipuler un tableau
- Vous savez définir et manipuler une collection
- Vous connaissez la syntaxe des structures de contrôle du langage PL/SQL
- Vous savez définir et manipuler un curseur explicite
- Vous connaissez les attributs de curseur



[DIAPO]

Vous voici arrivée à la fin de ce module et pour conclure on peut maintenant dire que :

- Vous connaissez des types de données complexes
- Vous savez définir et manipuler un tableau
- Vous savez définir et manipuler une collection
- Vous connaissez la syntaxe des structures de contrôle du langage PL/SQL
- Vous savez définir et manipuler un curseur explicite
- Vous connaissez les attributs de curseur

```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:      #  
#                bit.ly/eni_ac  
#  
#####  
#####
```

PL / SQL

Module 6 – La section EXCEPTION



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

Objectifs

- Savoir gérer les erreurs de programmation
- Savoir gérer les anomalies utilisateurs



[DIAPO]

Vous allez apprendre

- à gérer les erreurs de programmation qui risqueraient de stopper vos programmes de manière inapproprié.
- à gérer les anomalies utilisateurs afin d'interdire des utilisations inappropriées de vos fonctionnalités.

La section EXCEPTION

L'emplacement de la section Exception

```
DECLARE
  --Déclaration des variables
BEGIN
  --Traitement
EXCEPTION
  --Gestion des erreurs
END;
```



[DIAPO]

Dans un bloc PL/SQL, la section Exception se trouve toujours après le traitement.
Cette section est facultative.

Voici une vue d'ensemble des différentes sections que l'on peut trouver dans un bloc PL/SQL afin de se resituer.

- L1 La première section DECLARE que vous maîtrisez. Elle est facultative.
- L2 Elle permet de déclarer vos variables.
- L3 Ensuite on a la section BEGIN que vous maîtrisez aussi.
- L4 C'est elle qui contient le traitement
- L5 Et enfin la section EXCEPTION, celle que nous allons étudier. Cette section est facultative et se trouve toujours après la section BEGIN.

La section EXCEPTION

Les différents types d'erreurs

- Erreur de compilation
- Erreur d'exécution
- Erreur utilisateur



[DIAPO]

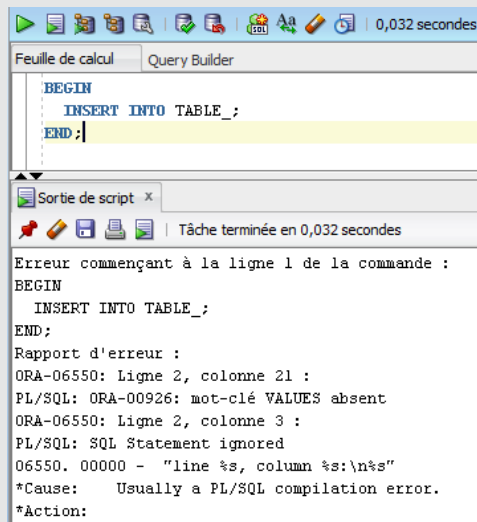
Nous avons donc les erreurs de compilation qui sont dus à des erreurs de syntaxe en général. C'est-à-dire qu'il y a une erreur dans ce que vous avez écrit et il est impossible pour le serveur de compiler votre code.

Puis nous avons les erreurs d'exécution, c'est lorsque votre code compile mais il n'a pas réussi à s'exécuter jusqu'à la fin lorsque par exemple il y a une division par zéro.

Et pour finir les erreurs utilisateur lorsque l'on rencontre par exemple des paramètres illogiques, un nom avec des chiffres, une commande avec une quantité négative etc etc...

La section EXCEPTION

Les erreurs de compilation



The screenshot shows a SQL Query Builder window with a toolbar at the top. The main area contains the following SQL code:

```
BEGIN
  INSERT INTO TABLE_
END;
```

Below the code editor, there is a 'Sortie de script' (Script Output) window. It displays the execution time 'Tâche terminée en 0,032 secondes' and an error message:

```
Erreur commençant à la ligne 1 de la commande :
BEGIN
  INSERT INTO TABLE_
END;
Rapport d'erreur :
ORA-06550: Ligne 2, colonne 21 :
PL/SQL: ORA-00926: mot-clé VALUES absent
ORA-06550: Ligne 2, colonne 3 :
PL/SQL: SQL Statement ignored
06550. 00000 - "line %s, column %s:\n%s"
*Cause:      Usually a PL/SQL compilation error.
*Action:
```

[DIAPO]

Une erreur de compilation arrive simplement lorsque la syntaxe de votre code est incorrecte.

Dans le cas présent, il n'y a rien à faire. Relisez votre programme afin qu'il respecte les bonnes syntaxes.

Rien de tel qu'un exemple pour bien comprendre.

Ici vous pouvez vous apercevoir que j'ai écrit dans mon traitement une requête DML incomplète.

Lorsque je demande au serveur de l'exécuter il me retourne une erreur de compilation.

Vous pouvez remarquer le nom du type d'erreur à la fin de la sortie de script.

La seule chose que je peux faire c'est d'écrire un code correct qui compilera...

Ce sera tout pour ce type d'erreur .

La section EXCEPTION

Les erreurs d'exécution prédéfinies

- Erreur Oracle
- Exception numérotée
- Exception nommée



[DIAPO]

On va aborder les erreurs d'exécution.

Une erreur d'exécution se fait dans un programme compilé mais qui n'a pas réussi à se terminer correctement.

Ces erreurs sont connues par le serveur Oracle. Il en existe deux types ! (Avec les doigts)

.

Parmi ces erreurs il y en a certaines qui sont prédéfinies c'est-à-dire qu'elles ont été associées à un nom par le serveur. Et il y a les erreurs non prédéfinies.

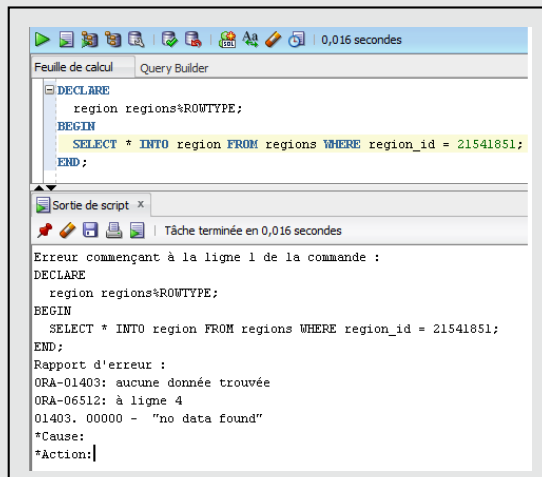
Penchons-nous sur les erreurs prédéfinies :

Une erreur prédéfinie est une erreur Oracle associée à un numéro et à un nom.

Voici un exemple de gestion d'erreur prédéfinie :

La section EXCEPTION

Les erreurs d'exécution prédéfinies



```
DECLARE
  region regions%ROWTYPE;
BEGIN
  SELECT * INTO region FROM regions WHERE region_id = 21541851;
END;
```

Sortie de script x

Tâche terminée en 0,016 secondes

Erreur commençant à la ligne 1 de la commande :

```
DECLARE
  region regions%ROWTYPE;
BEGIN
  SELECT * INTO region FROM regions WHERE region_id = 21541851;
END;
```

Rapport d'erreur :

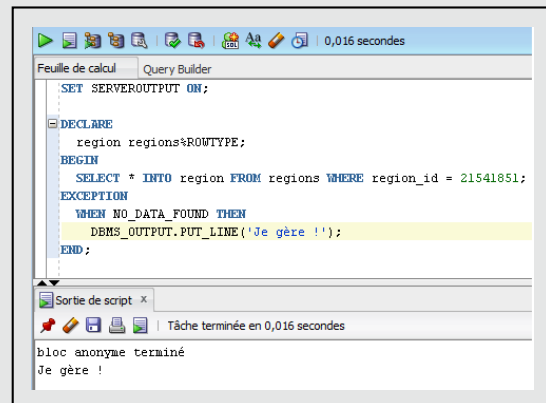
ORA-01403: aucune donnée trouvée

ORA-06512: à ligne 4

01403. 00000 - "no data found"

*Cause:

*Action:



```
SET SERVEROUTPUT ON;

DECLARE
  region regions%ROWTYPE;
BEGIN
  SELECT * INTO region FROM regions WHERE region_id = 21541851;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Je gère !');
END;
```

Sortie de script x

Tâche terminée en 0,016 secondes

bloc anonyme terminé

Je gère !

[DIAPO]

Dans notre premier exemple nous avons une erreur due au fait que l'on veut mettre le résultat d'une requête dans une variable mais la requête ne retourne rien.

Cela lève une erreur. Cette erreur se nomme NO_DATA_FOUND, dans le premier exemple nous n'avons pas de section pour intercepter cette erreur ainsi le résultat du programme se conclut par l'affichage d'une erreur.

On peut apercevoir dans la sortie de script un numéro (1403) et un nom (no data found) Sans que n'ayons rien pu faire. Le programme s'arrête donc de manière brutale.

Dans le second exemple, nous avons la même erreur mais nous avons mis en place une section EXCEPTION qui va nous permettre d'intercepter l'erreur grâce au nom de l'exception et d'exécuter un traitement choisi par l'utilisateur.

Ainsi, dans ce deuxième exemple on peut dire que l'arrêt n'a pas été brutal car anticipé. Notre traitement se contente d'afficher « je gère »

Vous avez vu, rien de très compliqué, alors pour retrouver la liste des erreurs prédéfinies je vous invite à vous rendre sur la documentation Oracle.

La section EXCEPTION

La liste des exceptions prédéfinies

NOM	NUMERO	SQLCODE
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPENED	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
NO_DATA_FOUND	ORA-01403	+100
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476



[DIAPO]

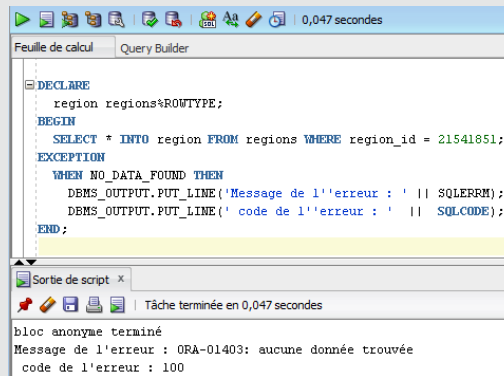
La liste des erreurs prédéfinies c'est-à-dire associée à un nom est courte.

C'est-à-dire que toutes les autres erreurs n'ont pas de nom...

On retrouve par exemple la division par zéro et bien d'autres dont vous retrouverez le détail sur la documentation Oracle.

La section EXCEPTION

Les fonctions SQLEERM & SQLCODE



```
DECLARE
    region regions%ROWTYPE;
BEGIN
    SELECT * INTO region FROM regions WHERE region_id = 21541851;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Message de l'erreur : ' || SQLEERM);
        DBMS_OUTPUT.PUT_LINE(' code de l'erreur : ' || SQLCODE);
END;
```

Sortie de script x

Tâche terminée en 0,047 secondes

bloc anonyme terminé
Message de l'erreur : ORA-01403: aucune donnée trouvée
code de l'erreur : 100



#####

==> Amélioration Continue <==

#

En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à cette adresse:

#

bit.ly/eni_ac

#

#####

[INTRO]

Alors maintenant je vais vous présenter deux mots clé importants qui vont vous permettre dans votre traitement de retrouver le numéro de l'erreur et son nom.

[DIAPO]

Pour chaque erreur il est possible de récupérer son code et le message associé à l'erreur grâce aux mots-clés SQLERRM et SQLCODE. Cela peut être très utile dans votre gestion des erreurs ou pour faire de la journalisation des erreurs.

Commenter le code...

Nous voici arrivé à la fin de cette présentation.

La section EXCEPTION

Les erreurs utilisateur

- Définies grâce au type EXCEPTION
- Levées grâce au mot-clé RAISE
- Permettent de se protéger de traitements illogiques



[INTRO]

Lorsque l'on rencontre par exemple des valeurs de paramètres illogiques, on doit pouvoir lever une erreur.

PL/SQL vous offre la possibilité de définir vos propres exceptions ! Vous devez déclarer et lever vous-même vos exceptions afin de gérer vos possible erreurs applicatives.

Pour se faire :

[DIAPO]

Vous devez déclarer votre exception personnalisée avec le type EXCEPTION.

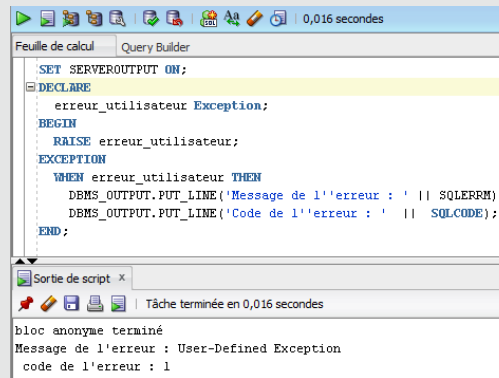
Ensuite vous pouvez lever votre erreur personnalisé grâce au mot clé RAISE (qui signifie lever en anglais)

Et tout ceci dans le but de vous protéger de traitements illogiques.

Pour bien comprendre voici un exemple très simple :

La section EXCEPTION

Les erreurs utilisateur



```
SET SERVEROUTPUT ON;
DECLARE
  erreur_utilisateur Exception;
BEGIN
  RAISE erreur_utilisateur;
EXCEPTION
  WHEN erreur_utilisateur THEN
    DBMS_OUTPUT.PUT_LINE('Message de l''erreur : ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('Code de l''erreur : ' || SQLCODE);
END;
```

Sortie de script x

Tâche terminée en 0,016 secondes

bloc anonyme terminé
Message de l'erreur : User-Defined Exception
code de l'erreur : 1



[DIAPO]

Je commente mon code

Je déclare mon exception

Je la lève

Je la traite

Nous avons donc vu ce qu'il y avait à voir concernant les exceptions utilisateurs

La section EXCEPTION

Les erreurs d'exécution non prédéfinies

- Erreurs Oracle
- Définies par un code erreur numérique



[DIAPO]

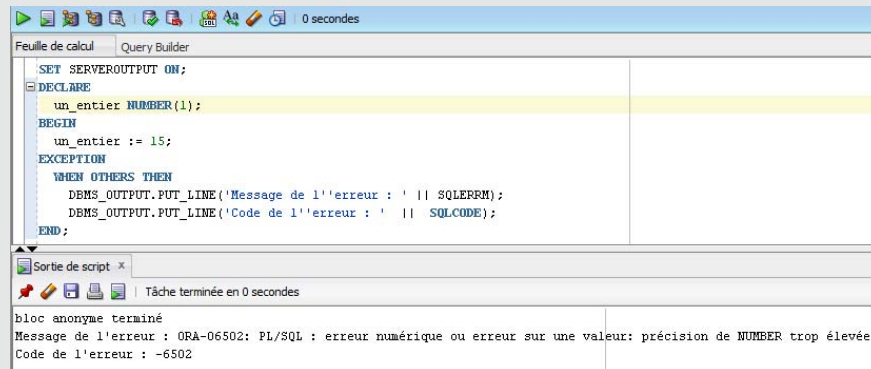
Une erreur d'exécution se fait dans un programme compilé mais qui n'a pas réussi à se terminer correctement.

Ces erreurs sont connues par le serveur Oracle. Parmi ces erreurs il y en a certaines qui sont prédéfinies c'est-à-dire qu'elles ont été associées à un numéro et un nom par le serveur et d'autres qui ne sont pas associée à un nom mais juste à un numéro. Ce sont des erreurs Oracles non prédéfinies.

Je récapitule :

La section EXCEPTION

Les erreurs d'exécution non prédéfinies



The screenshot shows a window titled "Feuille de calcul" with a "Query Builder" tab. The script contains the following PL/SQL code:

```
SET SERVEROUTPUT ON;
DECLARE
  un_entier NUMBER(1);
BEGIN
  un_entier := 15;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Message de l'erreur : ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('Code de l'erreur : ' || SQLCODE);
END;
```

Below the script, the "Sortie de script" window shows the execution results:

```
bloc anonyme terminé
Message de l'erreur : ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur: précision de NUMBER trop élevée
Code de l'erreur : -6502
```

[INTRO]

Une erreur non prédéfinie est un erreur Oracle qui est seulement associée à un numéro.
Voici un exemple d'erreur non préfinie :

[DIAPO]

On commente le code...

Pour intercepter ces erreurs nous pouvons utiliser le mot clé OTHERS dans le WHEN.

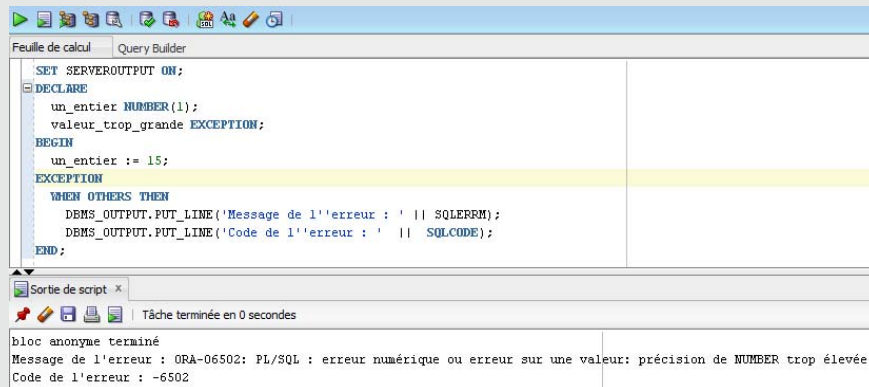
Le problème c'est que la seule manière de gérer cette erreur est d'utiliser le WHEN OTHERS ce qui me gêne un peu.

Ça me semble une mauvaise idée de gérer toute les erreurs dans le WHEN OTHER.

L'idéal serait

La section EXCEPTION

La directive de compilation PRAGMA EXCEPTION_INIT



```
SET SERVEROUTPUT ON;
DECLARE
  un_entier NUMBER(1);
  valeur_trop_grande EXCEPTION;
BEGIN
  un_entier := 15;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Message de l'erreur : ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('Code de l'erreur : ' || SQLCODE);
END;
```

Sortie de script x

Tâche terminée en 0 secondes

bloc anonyme terminé
Message de l'erreur : ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur: précision de NUMBER trop élevée
Code de l'erreur : -6502

De pouvoir lier notre erreur 6502 à une variable de type EXCEPTION afin de gérer notre erreur non prédéfinie de manière spécifique.

Alors ? (Menton en l'air) et bien c'est très simple ! On doit utiliser l'instruction PRAGMA EXCEPTION_INIT !!!

Comme ceci :



La section EXCEPTION

La directive de compilation PRAGMA EXCEPTION_INIT

```
SET SERVEROUTPUT ON;
DECLARE
  un_entier NUMBER(1);
  valeur_trop_grande EXCEPTION;
  PRAGMA EXCEPTION_INIT(valeur_trop_grande, -6502);
BEGIN
  un_entier := 15;
  EXCEPTION
    WHEN valeur_trop_grande THEN
      DBMS_OUTPUT.PUT_LINE('Nous sommes ici');
      DBMS_OUTPUT.PUT_LINE('Message de l''erreur : ' || SQLERRM);
      DBMS_OUTPUT.PUT_LINE('Code de l''erreur : ' || SQLCODE);
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Message de l''erreur : ' || SQLERRM);
      DBMS_OUTPUT.PUT_LINE('Code de l''erreur : ' || SQLCODE);
END;
```

Sortie de script x

Tâche terminée en 0,015 secondes

bloc anonyme terminé
Nous sommes ici
Message de l'erreur : ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur: précision de NUMBER trop élevée
Code de l'erreur : -6502

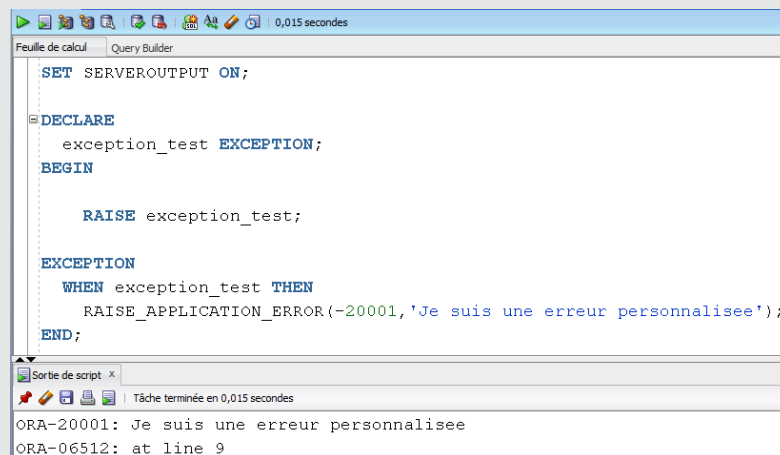


On commente le code...

Voilà bravo maintenant vous savez comment gérer vos erreurs oracle non prédéfinie.

La section EXCEPTION

La procédure RAISE_APPLICATION_ERROR



```
SET SERVEROUTPUT ON;

DECLARE
    exception_test EXCEPTION;
BEGIN
    RAISE exception_test;

EXCEPTION
    WHEN exception_test THEN
        RAISE_APPLICATION_ERROR(-20001, 'Je suis une erreur personnalisée');
END;
```

Sortie de script : *

Tâche terminée en 0,015 secondes

ORA-20001: Je suis une erreur personnalisée
ORA-06512: at line 9

Je vais vous parler d'une fonctionnalité très importante.

Une fonctionnalité qui permet de publier ses propres messages d'erreur et propres codes d'erreur.

On peut choisir des codes d'erreurs compris entre -20000 et -20999.

Ainsi, grâce à RAISE_APPLICATION_ERREUR on peut remonter à une application appelante un code d'erreur bien précis pour une situation bien précise.

Par exemple lorsque je veux commander un nombre négatif d'article je peux renvoyer à mon application le code -20100 avec comme message « Impossible d'utiliser des nombres négatifs dans les commandes »

Et l'application réagira en conséquence.

Voici comment la mettre en place :

[Je commente le code.](#)

La section EXCEPTION

La propagation des exceptions 1/3

```
SET SERVEROUTPUT ON;

DECLARE
    exception_test EXCEPTION;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Je suis avant l''erreur');
    RAISE exception_test;
    DBMS_OUTPUT.PUT_LINE('Je suis apres l''erreur');
EXCEPTION
    WHEN exception_test THEN
        DBMS_OUTPUT.PUT_LINE('Je traite l''erreur');
END;
```



```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
# cette adresse:               #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

Pour bien gérer vos erreurs vous devez bien comprendre la propagation des erreurs entre les Blocs. Alors sans plus attendre voici les exemples qui vont vous permettre de comprendre ces mécanismes !

Lorsqu'une erreur est levée le traitement après l'erreur ne sera pas exécutée.

Maintenant on va voir un nouveau cas de figure

La section EXCEPTION

La propagation des exceptions 2/3

```
SET SERVEROUTPUT ON;

DECLARE
    exception_test EXCEPTION;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Je suis avant le sous bloc');
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Je suis avant l''erreur');
        RAISE exception_test;
        DBMS_OUTPUT.PUT_LINE('Je suis apres l''erreur');
    END;
    DBMS_OUTPUT.PUT_LINE('Je suis après le sous bloc');
EXCEPTION
    WHEN exception_test THEN
        DBMS_OUTPUT.PUT_LINE('Je traite l''erreur');
END;
```



Ici nous avons un bloc principal dans lequel se trouve un sous bloc levant une erreur. Le sous bloc ne gère pas d'erreur ainsi l'erreur sera remontée au bloc principal qui gèrera lui-même l'erreur.

Par contre tout le traitement qu'il soit dans le sous bloc ou bloc principal après cette erreur ne sera pas exécuté.

Un petit dernier pour la route

La section EXCEPTION

La propagation des exceptions 3/3

```
SET SERVEROUTPUT ON;

DECLARE
    exception_test EXCEPTION;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Je suis avant le sous bloc');
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Je suis avant l''erreur');
        RAISE exception_test;
        DBMS_OUTPUT.PUT_LINE('Je suis apres l''erreur');
    EXCEPTION
        WHEN exception_test THEN
            DBMS_OUTPUT.PUT_LINE('Je traite l''erreur dans le sous bloc');
    END;
    DBMS_OUTPUT.PUT_LINE('Je suis après le sous bloc');
EXCEPTION
    WHEN exception_test THEN
        DBMS_OUTPUT.PUT_LINE('Je traite l''erreur');
END;
```



Ici nous avons un bloc principal qui contient un sous bloc.

Le sous bloc lève une erreur et la gère lui-même, ainsi le bloc principal pourra continuer son exécution de manière normale et n'aura pas à gérer l'erreur du sous bloc (Vu que le sous bloc s'en est déjà chargé)

Conclusion

- Vous savez ce qu'est une erreur utilisateur
- Vous savez ce qu'est une erreur prédéfinie & non prédéfinie
- Vous savez utiliser le type EXCEPTION
- Vous savez lever une erreur l'instruction avec RAISE
- Vous savez associer une erreur non prédéfinie à une variable EXCEPTION
- Vous savez associer un message à une erreur avec RAISE_APPLICATION_ERROR
- Vous connaissez le mécanisme de propagation des erreurs



C'est la fin du module. Maintenant

- Vous savez ce qu'est une erreur utilisateur
- Vous savez ce qu'est une erreur prédéfinie & non prédéfinie
- Vous savez utiliser le type EXCEPTION
- Vous savez lever une erreur l'instruction avec RAISE
- Vous savez associer une erreur non prédéfinie à une variable

EXCEPTION

- Vous savez associer un message à une erreur avec

RAISE_APPLICATION_ERROR

- Vous connaissez le mécanisme de propagation des erreurs

```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à cette  
adresse:      #  
#                bit.ly/eni_ac  
#                #  
#####  
#####
```

PL / SQL

Module 7 – Les procédures stockées



```
#####
#####
#                               ==> Amélioration Continue <==
#
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:                 #
#                               bit.ly/eni_ac
#                               #
#####
#####
```


Objectifs

- Savoir créer une procédure stockée
- Savoir utiliser les différents types de paramètres pour les procédures stockées
- Savoir appeler une procédure stockée



[INTRO]

Les procédures stockées vont nous permettent de stocker nos blocs PL/SQL sur le serveur afin d'être réutilisée dès que nous le désirons.

[DIAPO]

Voici vos objectifs pour ce module :

- Savoir créer une procédure stockée
- Savoir utiliser les différents types de paramètres pour les procédures stockées
- Savoir appeler une procédure stockée

Les procédures stockées

Définition

- Programme défini par l'utilisateur
- Programme stocké sur le serveur
- Bloc PL/SQL associé à un nom



[DIAPO]

Les procédures stockées permettent de stocker des blocs PL/SQL sur le serveur pour être appelé lorsque cela est nécessaire.

Elle peut contenir tout code SQL, y compris :

- transaction
- mise à jour,
- SQL dynamique
- cursor
- commande DDL

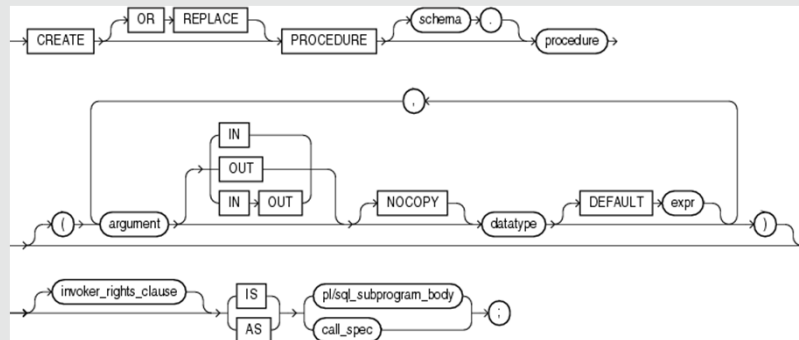
Revoyons ça :

Une PS, c'est :

- Programme défini par l'utilisateur
- Programme stockée sur le serveur
- Bloc PL/SQL associé à un nom

Les procédures stockées

La syntaxe de création



[DIAPO]

Vous avez appris ce qu'est une procédure stockée maintenant on va voir la pratique et quoi de mieux que de se pencher sur la documentation officielle pour découvrir la syntaxe

Les procédures stockées

Exemple de création d'une procédure stockée

```
SET SERVEROUTPUT ON;

CREATE OR REPLACE PROCEDURE afficher_employes
AS
    CURSOR cursor_employees_it IS SELECT first_name,last_name FROM EMPLOYEES WHERE DEPARTMENT_ID = 60;
    employee employees%ROWTYPE;
BEGIN

    FOR employee IN cursor_employees_it LOOP
        DBMS_OUTPUT.PUT_LINE(employee.first_name);
    END LOOP;

END;
```



[INTRO]

On commente le schéma

[DIAPO]

Commentez

```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
# cette adresse:               #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

Les procédures stockées

L'appel à une procédure stockée

- En utilisant la commande EXECUTE
- En utilisant le nom de la procédure stockée



[INTRO]

Avoir une procédure stockée c'est bien mais comment faire pour l'utiliser ??

On peut appeler une procédure stockée de deux manières différentes :

[DIAPO]

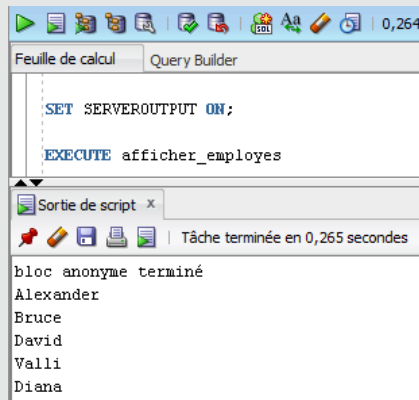
En utilisant la commande EXECUTE

En utilisant le nom de la procédure stockée

Lors de ce module je ne serai pas avar d'exemple ! En voilà un de plus pour la première façon d'appeler une procédure stockée 😊

Les procédures stockées

L'appel par la commande EXECUTE



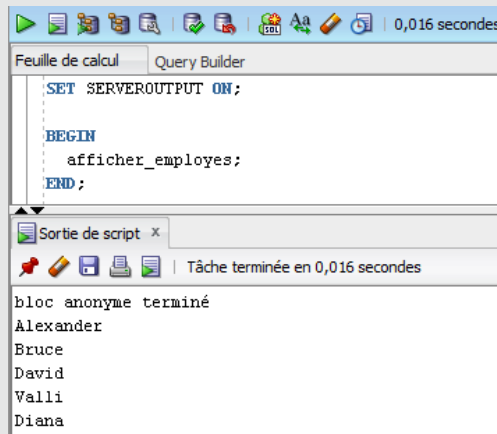
[DIAPO]

[Je commente le code](#)

Et encore un pour la seconde façon de faire :

Les procédures stockées

L'appel à partir d'un autre bloc



The screenshot shows a SQL Query Builder window with a toolbar at the top. The main text area contains the following SQL code:

```
SET SERVEROUTPUT ON;  
  
BEGIN  
    afficher_employees;  
END;
```

Below the code editor, there is a 'Sortie de script' (Script Output) window. It shows the execution result:

```
bloc anonyme terminé  
Alexander  
Bruce  
David  
Valli  
Diana
```

The status bar at the bottom of the script output window indicates 'Tâche terminée en 0,016 secondes' (Task completed in 0.016 seconds).



[DIAPO]

[Je commente le code](#)

Voilà et c'est tout, alors félicitation maintenant vous savez faire appel à des proc stock comme on dit par chez nous !

Les procédures stockées

Les paramètres de procédure

- IN
- OUT
- IN OUT



[INTRO]

Utiliser des fonctions sans paramètre serait tout de même un peu triste.

Le PL/SQL nous offre trois types de paramètres différents !

Je vous les présente de suite !

[DIAPO]

IN -> Paramètre est passé en entrée de procédure

OUT -> Le paramètre est valorisé dans la procédure et renvoyé à l'environnement appelant.

IN OUT -> le paramètre est passé en entrée de la procédure et il est renvoyé à l'environnement appelant.

Commençons par voir le paramètre de type IN dans les détails :

Les procédures stockées

Les paramètres IN

- Type par défaut
- Paramètre d'entrée



Type par défaut Quand on n'indique rien

Paramètre d'entrée C'est le paramètre d'entrée donc le Paramètre est passé en entrée de procédure

```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
cette adresse:      #  
#                bit.ly/eni_ac  
#  
#####  
#####
```

Les procédures stockées

Exemple de paramètre d'entrée

```
SET SERVEROUTPUT ON;

CREATE OR REPLACE PROCEDURE afficher_employes(id_department IN NUMBER)
AS
    CURSOR cursor_employees_it IS SELECT first_name,last_name FROM EMPLOYEES WHERE DEPARTMENT_ID = id_department;
    employee employees%ROWTYPE;
BEGIN
    FOR employee IN cursor_employees_it LOOP
        DBMS_OUTPUT.PUT_LINE(employee.first_name);
    END LOOP;
END;
```



[INTRO]

Voici un exemple !

[DIAPO]

Encadré ligne 1

Encadré ligne 3

Encadré ligne 4

Encadré ligne 5

Encadré ligne 6

Encadré ligne 7

Encadré ligne 8

Encadré ligne 9

Encadré ligne 10

Encadré ligne 11

Nous allons maintenant regarder le paramètre de type OUT en détails !

Les procédures stockées

Les paramètres OUT

- Paramètre de sortie



[DIAPO 12]

Donc le paramètre OUT est valorisé dans la procédure et renvoyé à l'environnement appelant.

Exemple :

Les procédures stockées

Exemple de paramètre de sortie

```
SET SERVEROUTPUT ON;

CREATE OR REPLACE PROCEDURE afficher_employees(id_department IN NUMBER, total_employees OUT NUMBER)
AS
    CURSOR cursor_employees_it IS SELECT first_name, last_name FROM EMPLOYEES WHERE DEPARTMENT_ID = id_department;
    employee employees%ROWTYPE;
BEGIN
    SELECT COUNT(*) INTO total_employees FROM employees WHERE DEPARTMENT_ID = id_department;

    FOR employee IN cursor_employees_it LOOP
        DBMS_OUTPUT.PUT_LINE(employee.first_name);
    END LOOP;
END;
```



[DIAPO 13]

Encadré ligne 1

Encadré ligne 3

Encadré ligne 4

Encadré ligne 5

Encadré ligne 6

Encadré ligne 7

Encadré ligne 9

Encadré ligne 11

Encadré ligne 12

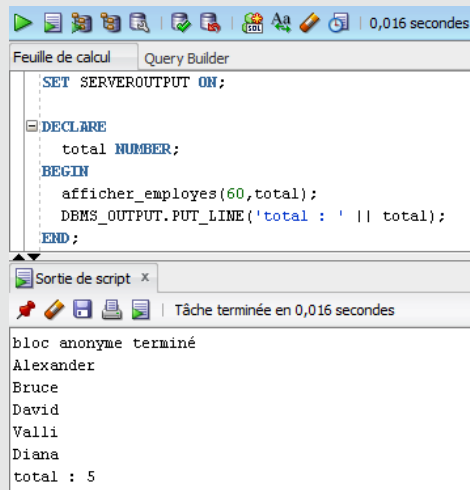
Encadré ligne 13

Encadré ligne 14

Regardons le résultat de cette proc stock !

Les procédures stockées

Le passage de paramètres à une procédure



The screenshot shows a SQL Query Builder window with a toolbar at the top. The main text area contains the following SQL code:

```
SET SERVEROUTPUT ON;  
  
DECLARE  
    total NUMBER;  
BEGIN  
    afficher_employes(60,total);  
    DBMS_OUTPUT.PUT_LINE('total : ' || total);  
END;
```

Below the code editor, there is a 'Sortie de script' (Script Output) window. It displays the execution results:

```
bloc anonyme terminé  
Alexander  
Bruce  
David  
Valli  
Diana  
total : 5
```

The status bar at the bottom of the script output window indicates 'Tâche terminée en 0,016 secondes' (Task completed in 0.016 seconds).



[DIAPO 14]

Commenter le code

Et pour finir nous allons voir le paramètre IN OUT dans les détails :

Les procédures stockées

Les paramètres IN OUT

- Paramètres d'entrées et de sorties



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:                #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

Le paramètre IN OUT est passé en entrée de la procédure et il est renvoyé à l'environnement appelant.
Un dernier exemple !

Les procédures stockées

Exemple de paramètre entrée / sortie

```
SET SERVEROUTPUT ON;

CREATE OR REPLACE PROCEDURE afficher_employees(info IN OUT NUMBER)
AS
    CURSOR cursor_employees_it IS SELECT first_name,last_name FROM EMPLOYEES WHERE DEPARTMENT_ID = info;
    employee employees%ROWTYPE;
BEGIN

    FOR employee IN cursor_employees_it LOOP
        DBMS_OUTPUT.PUT_LINE(employee.first_name);
    END LOOP;

    SELECT COUNT(*) INTO info FROM employees WHERE DEPARTMENT_ID = info;
END;
```



Encadré ligne 1

Encadré ligne 2

Encadré ligne 3

Encadré ligne 4

Encadré ligne 5

Encadré ligne 6

Encadré ligne 8

Encadré ligne 9

Encadré ligne 10

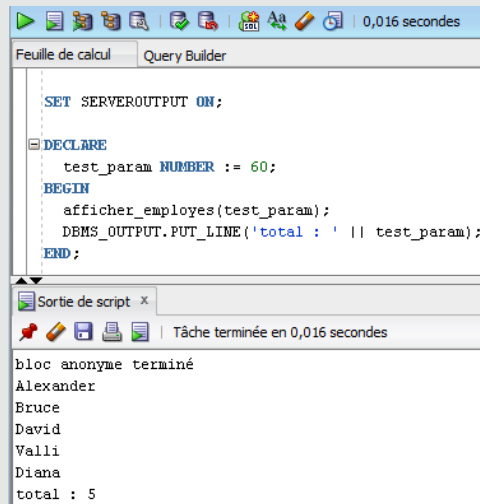
Encadré ligne 12

Encadré ligne 13

Regardons le résultat de cette proc stock !

Les procédures stockées

L'utilisation d'un paramètre entrée / sortie



The screenshot shows the Oracle SQL Developer interface. At the top, there's a toolbar and a status bar indicating '0,016 secondes'. Below the toolbar, there are two tabs: 'Feuille de calcul' and 'Query Builder'. The 'Query Builder' tab is active, displaying a PL/SQL procedure. The code is as follows:

```
SET SERVEROUTPUT ON;

DECLARE
test_param NUMBER := 60;
BEGIN
afficher_employes(test_param);
DBMS_OUTPUT.PUT_LINE('total : ' || test_param);
END;
```

Below the code editor, there's a 'Sortie de script' (Script Output) window. It shows the execution results:

```
bloc anonyme terminé
Alexander
Bruce
David
Valli
Diana
total : 5
```

Commentez le code

Les procédures stockées

Démonstration



Inserer un department avec contrôle...

Conclusion

- Vous savez créer des procédures stockées
- Vous savez utiliser des procédures stockées
- Vous connaissez les différents types de paramètres applicables aux procédures stockées



Vous savez maintenant créer des procédures stockées.

Je vous rappelle que cela permet vraiment d'optimiser l'accès à nos données dans 4 domaines.

La sécurité

Le trafic Réseau

Les temps de calcul

La productivité

La sécurité

En permettant par exemple aux utilisateurs d'accéder à une table en lecture ou en écriture uniquement via des fonctions PL/SQL.

Ainsi :

Ces fonctions peuvent empêcher des actions dangereuses lors des modifications.

Ces fonctions peuvent contrôler les données à fournir à l'utilisateur (et permettre par exemple de cacher un compte en banque) et cela sans donner d'informations sur la structure de la base de données.

Le Trafic réseau

Les traitements logiques de données étant stockés dans des fonctions sur le serveur de base de données cela permet de ne pas envahir le trafic réseau entre le client et le serveur de base de données avec les requêtes nécessaires au traitement.

De plus, lorsqu'on appelle une procédure stockée, nous avons seulement besoin de

préciser son nom et ses paramètres, cela s'avèrera toujours moins coûteux que d'envoyer une commande complète.

Le temps de calcul

Un serveur de base de données est principalement fait pour faire des calculs de données contrairement aux serveurs d'applications, donc si à cela on ajoute l'optimisation du trafic réseau et le fait que les fonctions stockées sont déjà compilées sur le serveur de base de données (contrairement aux requêtes). On peut être sûr que les calculs seront toujours plus performants lorsqu'ils sont effectués sur un serveur de base de données.

La productivité

Toutes les fonctions PL/SQL sont centralisées au même endroit permettant une meilleure gestion du code.

De plus, on peut utiliser un environnement dédié au PL/SQL qui offre un confort au développeur pour la conception et le débogage des requêtes et des fonctionnalités d'accès aux données.

Sur le serveur de base de données nous n'avons qu'une seule copie des fonctionnalités plutôt qu'une copie des fonctionnalités pour chaque client. On peut donc faire plus simplement des mises à jour sans affecter les différentes applications clientes.

Et pour finir, on peut faire plus simplement des modifications du schéma de la base de données sans impacter toutes les applications clientes.

Bon revenons s'en a ce que vous savez faire maintenant :

```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
cette adresse:          #  
#                bit.ly/eni_ac  
#  
#####  
#####
```

PL / SQL

Module 8 – Les fonctions



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

Objectifs

- Savoir créer une fonction utilisateur
- Savoir utiliser une fonction utilisateur
- Savoir exploiter la valeur retournée par une fonction



Les objectifs du module :

- Savoir créer une fonction utilisateur
- Savoir utiliser une fonction utilisateur
- Savoir exploiter la valeur retournée par une fonction

Les fonctions

Définition

- Bloc de code nommé et stocké sur le serveur
- Retourne toujours une et seule une valeur



[INTRO]

Une fonction est un programme contenant des traitements et qui retourne un seul résultat.

Quelle est la différence entre une fonction et une procédure stockée ? Et bien c'est qu'elle ne retourne qu'une seule valeur 😊

On récapitule :

[DIAPO]

Programme exécutant une ou des actions

Retourne toujours une et seulement une valeur

Les fonctions

Les spécificités

- Les paramètres d'entrées sont exclusivement de type IN
- Les paramètres d'entrées doivent être de type SQL et non PL/SQL
- Le paramètre de retour doit être de type SQL et non PL/SQL
- Les fonctions ne doivent pas faire de DML (INSERT, UPDATE, DELETE)



[INTRO]

Alors bien sur les fonctions ont quelques spécificités :

[DIAPO]

Les paramètres d'entrées sont exclusivement de type IN

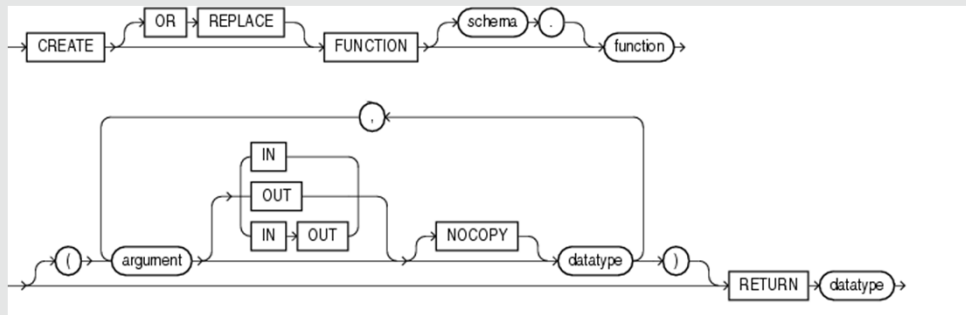
Les paramètres d'entrées doivent être de type SQL et non PL/SQL

Le paramètre de retour doit être de type SQL et non PL/SQL donc pas de boolean de pls_integer, de curseur etc etc.

Les fonctions ne doivent pas faire de DML (INSERT, UPDATE, DELETE)

Les fonctions

La syntaxe de création



Bon rien de tel qu'une bonne documentation Oracle pour vous introduire la syntaxe des fonctions Oracle.

[Commenter le schéma](#)

Alors concrètement ça donne quoi, vous voulez un exemple ?? Et bien voilà !

Les fonctions

Exemple de création d'une fonction

```
CREATE OR REPLACE FUNCTION multiplier_par_deux(nombre_a_multiplier IN NUMBER)
RETURN NUMBER
IS
resultat NUMBER;
BEGIN
    resultat := nombre_a_multiplier * 2;
    RETURN resultat;
END;
```



Commentez le code

Toute mes félicitations vous savez désormais écrire des fonctions.

Les fonctions

L'appel à une fonction

- Depuis un bloc PL/SQL
- Depuis une requête SQL



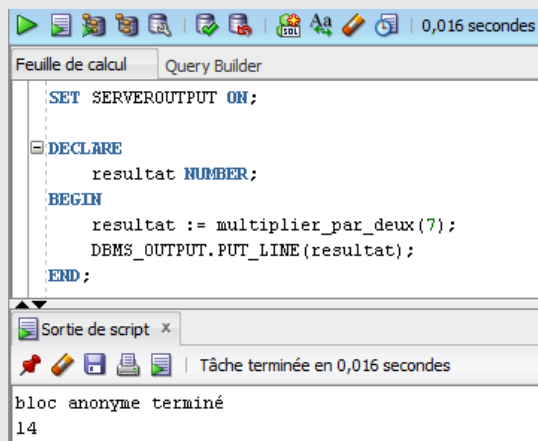
Avoir une fonction perso c'est bien mais comment faire pour l'utiliser ??

On peut appeler une fonction de deux manières différentes :

- Depuis un bloc PL/SQL
- Depuis une requête SQL

Les fonctions

L'appel à partir d'un bloc



```
SET SERVEROUTPUT ON;

DECLARE
    resultat NUMBER;
BEGIN
    resultat := multiplier_par_deux(7);
    DBMS_OUTPUT.PUT_LINE(resultat);
END;
```

Sortie de script x

Tâche terminée en 0,016 secondes

bloc anonyme terminé
14



```
#####
#####
```

```
#                ==> Amélioration Continue <==
```

```
#
```

```
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
cette adresse:  #
```

```
#                bit.ly/eni_ac
```

```
#
```

```
#####
#####
```

[INTRO]

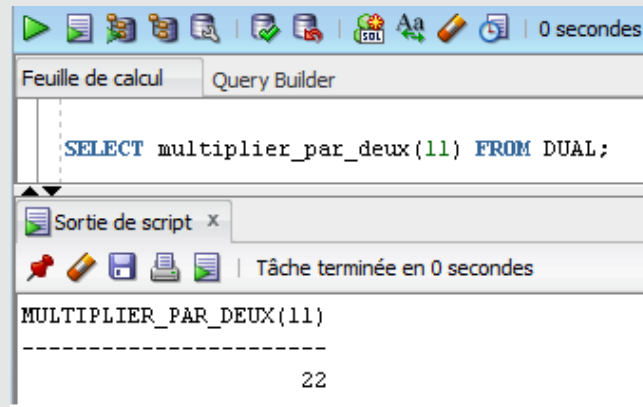
Et voici quelques exemples pour vous chers développeurs !

[DIAPO]

[Commentez le code](#)

Les fonctions

L'appel à partir d'une requête



[DIAPO]

On appelle la fonction via une requête.

Maintenant on ne peut plus rien vous cacher concernant les fonctions !!

Les fonctions

Démonstration



[DEMO]

Fonction qui retourne le nombre de ligne dans une table

Conclusion

- Vous savez créer des fonctions
- Vous savez utiliser des fonctions
- Vous savez exploiter la valeur retournée par la fonction



[INTRO]

Une fonction est un programme contenant des traitements et qui retourne un seul résultat.

Ça permet de faire des calculs complexes

Optimiser les calculs

Augmenter l'efficacité de vos requêtes !

[DIAPO]

Alors bravo car maintenant :

-Vous savez utiliser des fonctions

-Vous savez créer des fonctions

#####

==> Amélioration Continue <==
#

En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse: #

bit.ly/eni_ac
#

#####

PL / SQL

Module 9 – Les déclencheurs de base de données



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```


Objectifs

- Savoir expliquer ce qu'est un trigger
- Savoir créer un trigger



Traitements qui seront exécutés automatiquement lorsqu'un évènement spécifique se produira.

Bref, je vous en reparlerai plus tard. Quels sont les objectifs de ce cours :

- Savoir expliquer ce qu'est un trigger
- Savoir créer un trigger

Définition

- Trigger = Déclencheur
- Traitement s'exécutant automatiquement lorsqu'un évènement (insertion, suppression, mise à jour) se produit sur une table ou une vue



Trigger ! C'est aussi le nom plus couramment utilisé pour parler des déclencheurs de base de données trigger signifie déclencheur en anglais. Incroyable !
Bref revenons en a nos moutons. Alors :

Un déclencheur de base de données ou bien trigger est donc un traitement s'exécutant automatiquement lorsqu'un évènement (insertion, suppression, update) se produit sur une table ou une vue.

Alors bien sur les triggers ont quelques spécificités :

Les déclencheurs de base de données

Les spécificités

- Blocs associés à un nom
- Peuvent appeler des sous-programmes
- Non paramétrables
- COMMIT et ROLLBACK interdits



Blocs associés à un nom

Peuvent appeler des sous-programmes

Non paramétrables

Et il a peut réagir de manière différente face à une requête

[DIAPO 5]

Blablabla

Ainsi ça nous fait un certain nombre de possibilité différentes de trigger. Regardons ça de plus près :

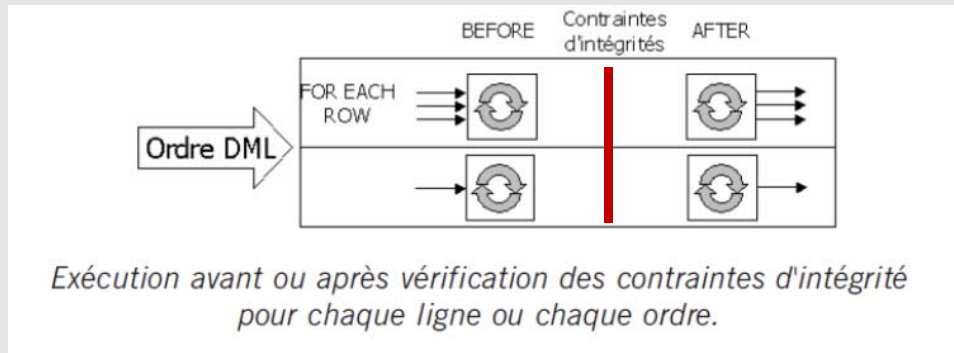
[DIAPO 6]

Il y a douze types de trigger possible.

Blabla

Les déclencheurs de base de données

Le principe de fonctionnement



Blablabla

Ainsi ça nous fait un certain nombre de possibilité différentes de trigger. Regardons ça de plus près :

Les déclencheurs de base de données

Les différents types de trigger de table

TRIGGER	BEFORE	STATEMENT	INSERT	SELECT	
			UPDATE	SELECT	
			DELETE	SELECT	
		ROW	INSERT	SELECT	New
			UPDATE		New / Old
			DELETE		New
	AFTER	STATEMENT	INSERT	SELECT	
			UPDATE	SELECT	
			DELETE	SELECT	
		ROW	INSERT		New
			UPDATE		New / Old
			DELETE		New

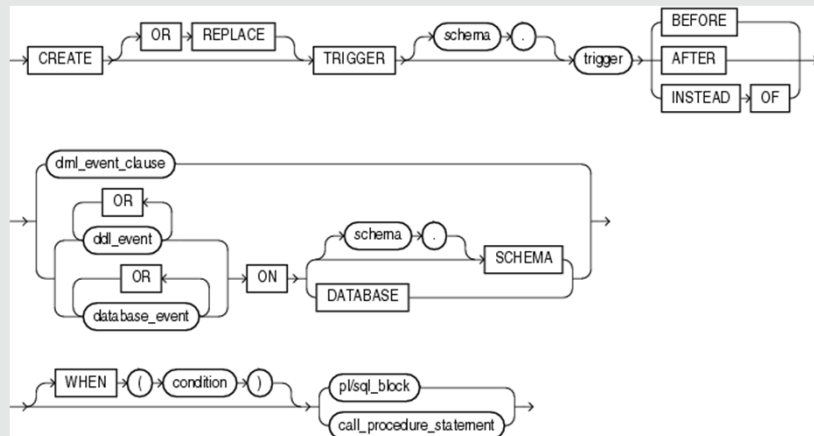


```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
# cette adresse:               #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

Il y a douze types de trigger possible.
Blabla

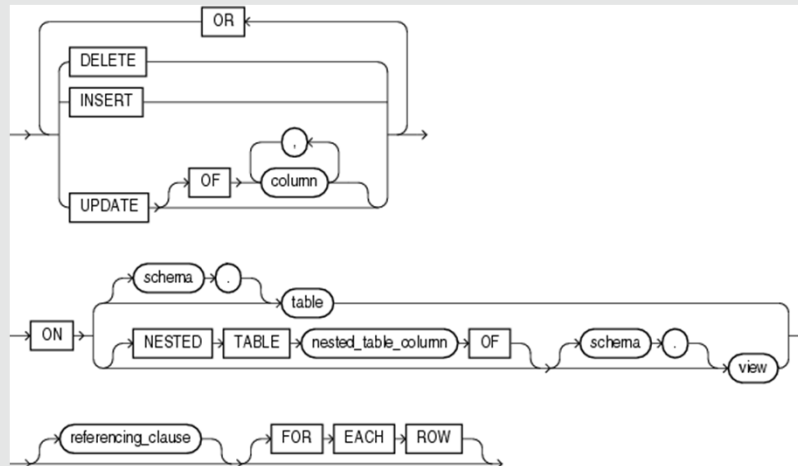
Les déclencheurs de base de données

La syntaxe de création 1/2



Commentez

La syntaxe de création 2/2



Commentez

On peut utiliser un seul et même trigger pour plusieurs évènements par exemple on peut exécuter un même trigger lors de l'arrivée d'une requête DELETE UPDATE et INSERT Sans pour autant exécuter le même traitement.

Pour ce faire il existe des clauses qui nous permettent de distinguer quelle type de requête a déclenché le trigger

Les déclencheurs de base de données

Les prédicats de trigger

- INSERTING
- DELETING
- UPDATING



Il y a la clause

INSERTING

DELETING

UPDATING

Voilà voilà, maintenant regardons ce que ça donne avec un bel exemple

[\[DIAPO 10\]](#)

[Commentez le code](#)

Les déclencheurs de base de données

L'utilisation des prédicats

```
CREATE OR REPLACE TRIGGER securisation_horaires
BEFORE INSERT OR UPDATE OR DELETE ON employees
BEGIN
    IF TO_CHAR (SYSDATE, 'HH24:MI') NOT BETWEEN '08:00' AND '18:00' OR TO_CHAR (SYSDATE, 'DY') IN ('SAT', 'SUN') THEN
        IF INSERTING THEN
            RAISE_APPLICATION_ERROR (-20205, 'Vous ne pouvez pas faire d'insertions hors des heures normales de bureau');
        END IF;

        IF DELETING THEN
            RAISE_APPLICATION_ERROR (-20206, 'Vous ne pouvez pas faire de suppressions hors des heures normales de bureau');
        END IF;

        IF UPDATING THEN
            RAISE_APPLICATION_ERROR (-20207, 'Vous ne pouvez pas faire de mises à jour hors des heures normales de bureau');
        END IF;
    END IF;
END;
```



Commentez le code

```
#####
#####
#           ==> Amélioration Continue <==
#
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à cette
adresse:      #
#           bit.ly/eni_ac
#
#####
#####
```

Les déclencheurs de base de données

Les variables implicites

- OLD
- NEW



Lorsque vous mettez à jour une donnée il y a donc l'ancienne valeur qui va être remplacée et la nouvelle qui va remplacer l'ancienne. Cela peut parfois être intéressant de connaître ces informations ! Et bien il existe une façon très simple d'y accéder grâce à deux variable explicite !!

[\[DIAPO 11\]](#)

[Commentez](#)

Maintenant je vais vous présenter un petit exemple afin de mieux saisir !

Les déclencheurs de base de données

L'appel aux variables OLD et NEW

```
CREATE OR REPLACE TRIGGER securisation_salaire
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    IF (:OLD.salary > :NEW.salary) THEN
        RAISE_APPLICATION_ERROR(-20210, 'Impossible de réduire un salaire');
    END IF;
END;
```



[Commentez le code](#)

Et pour finir je vais vous montrer comment les utiliser dans une clause conditionnelle propre aux triggers :

Les déclencheurs de base de données

La clause WHEN

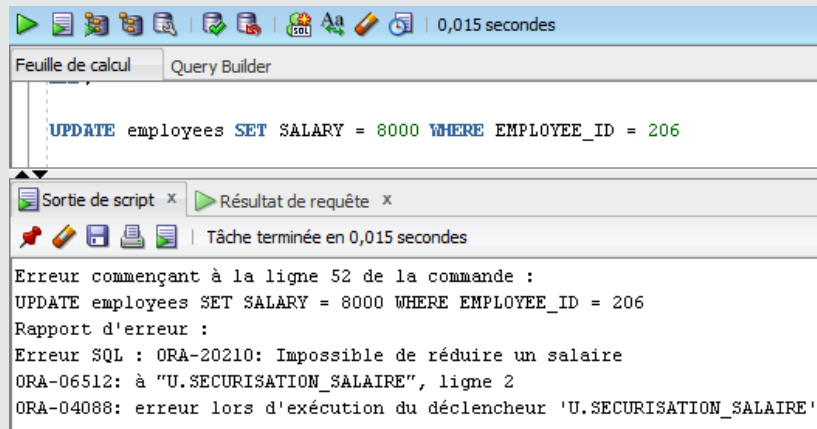
```
CREATE OR REPLACE TRIGGER securisation_salaire
BEFORE UPDATE ON employees
FOR EACH ROW
WHEN(OLD.salary > NEW.salary)
BEGIN
    RAISE_APPLICATION_ERROR(-20210, 'Impossible de réduire un salaire');
END;
```



[Commentez le code](#)

Les déclencheurs de base de données

L'exécution du trigger



The screenshot shows the Oracle SQL Developer interface. The top toolbar includes icons for running queries, saving, and other standard database operations. The main window is titled 'Query Builder' and contains the following SQL statement:

```
UPDATE employees SET SALARY = 8000 WHERE EMPLOYEE_ID = 206
```

Below the query, the 'Résultat de requête' (Query Result) tab is active, displaying an error message. The error message is as follows:

```
Erreur commençant à la ligne 52 de la commande :  
UPDATE employees SET SALARY = 8000 WHERE EMPLOYEE_ID = 206  
Rapport d'erreur :  
Erreur SQL : ORA-20210: Impossible de réduire un salaire  
ORA-06512: à "U.SECURISATION_SALAIRE", ligne 2  
ORA-04088: erreur lors d'exécution du déclencheur 'U.SECURISATION_SALAIRE'
```



[Commentez...](#)

Les déclencheurs de base de données

Les triggers sur vues

- La clause INSTEAD OF = Au lieu de
- Permet d'insérer, de modifier ou de supprimer à partir d'une vue **multitable**
- Utilise la clause FOR EACH implicitement



Maintenant nous allons parler d'un trigger bien différent des autres c'est le trigger sur vue. Trigger de type INSTEAD OF

INSTEAD OF = Au lieu de

Permet d'insérer, de modifier ou de supprimer à partir d'une vue multitable

Utilise la clause FOR EACH implicitement

Nous allons tester ça tout de suite, on va commencer par créer une vue

Exemple de trigger sur vue 1/4

```
CREATE OR REPLACE VIEW view_countries_regions
AS
  SELECT
    c.country_name AS country,
    r.region_name AS region
  FROM
    countries c
  JOIN regions r ON c.region_id = r.region_id;
```



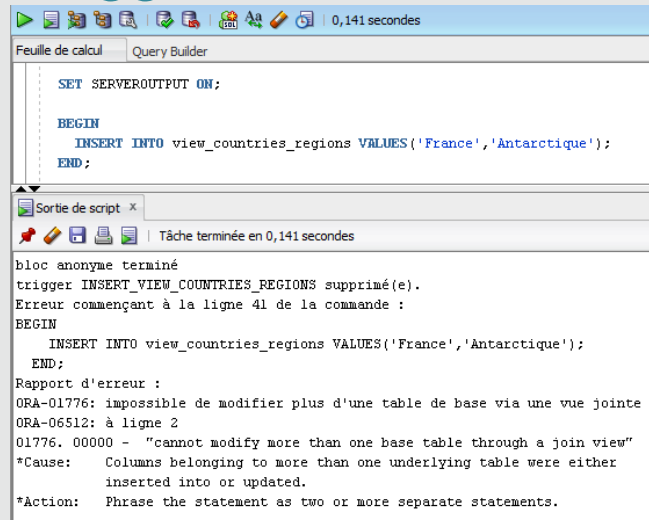
Commentez

Et maintenant on va tenter d'insérer une valeur dans la vue.

```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:                 #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

Les déclencheurs de base de données

Exemple de trigger sur vue 2/4



The screenshot shows the Oracle SQL Developer interface. At the top, there's a toolbar and a status bar indicating '0,141 secondes'. Below that, the 'Query Builder' tab is active, displaying a SQL script:

```
SET SERVEROUTPUT ON;

BEGIN
  INSERT INTO view_countries_regions VALUES('France','Antarctique');
END;
```

Below the script, the 'Sortie de script' (Script Output) window shows the execution results:

```
bloc anonyme terminé
trigger INSERT_VIEW_COUNTRIES_REGIONS supprimé(e).
Erreur commençant à la ligne 41 de la commande :
BEGIN
  INSERT INTO view_countries_regions VALUES('France','Antarctique');
END;
Rapport d'erreur :
ORA-01776: impossible de modifier plus d'une table de base via une vue jointe
ORA-06512: à ligne 2
01776. 00000 - "cannot modify more than one base table through a join view"
*Cause:      Columns belonging to more than one underlying table were either
              inserted into or updated.
*Action:     Phrase the statement as two or more separate statements.
```

Commentez

Vous voyez que cela n'est pas possible

On va donc créer notre trigger de vue...

Exemple de trigger sur vue 3/4

```
CREATE OR REPLACE TRIGGER insert_view_countries_regions
INSTEAD OF INSERT ON view_countries_regions
DECLARE
    total_region NUMBER;
    new_id_region NUMBER;
    total_country NUMBER;
    new_id_country NUMBER;
BEGIN
    SELECT COUNT(*) INTO total_region FROM REGIONS WHERE REGION_NAME = :NEW.region;
    IF total_region = 0 THEN
        SELECT (MAX(REGION_ID)+1) INTO new_id_region FROM REGIONS;
        INSERT INTO regions VALUES(new_id_region,:NEW.region);
    END IF;

    SELECT COUNT(*) INTO total_country FROM countries WHERE COUNTRY_NAME = :NEW.country;
    IF total_country = 0 THEN
        SELECT (MAX(REGION_ID)+1) INTO new_id_country FROM countries;
        INSERT INTO countries VALUES(new_id_country,:NEW.country,new_id_region);
    END IF;
END;
```

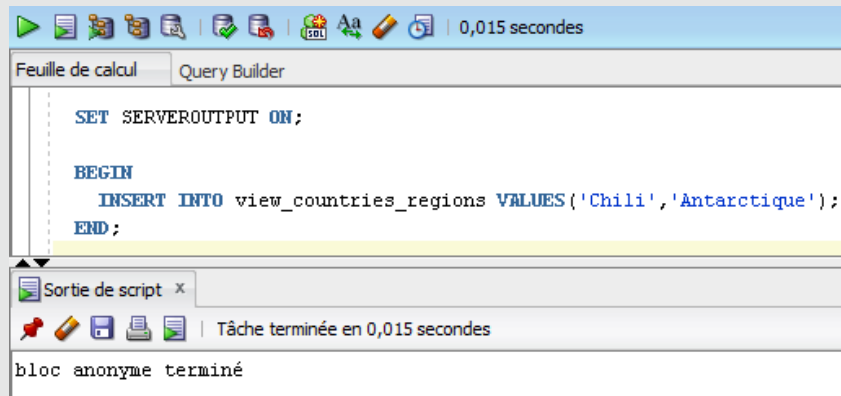


[Commentez](#)

Et on va retenter d'insérer dans notre vue

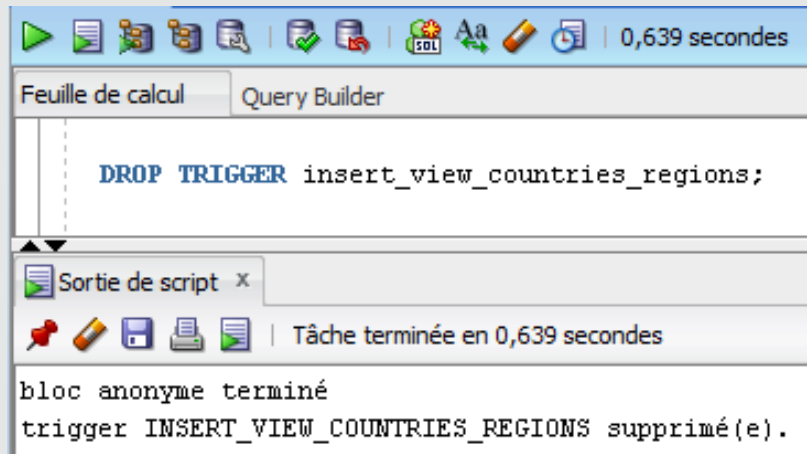
Les déclencheurs de base de données

Exemple de trigger sur vue 4/4



Commentez, et là on voit que tout se passe pour le mieux ! L'insertion a eu lieu
On supprime notre trigger

La suppression d'un trigger



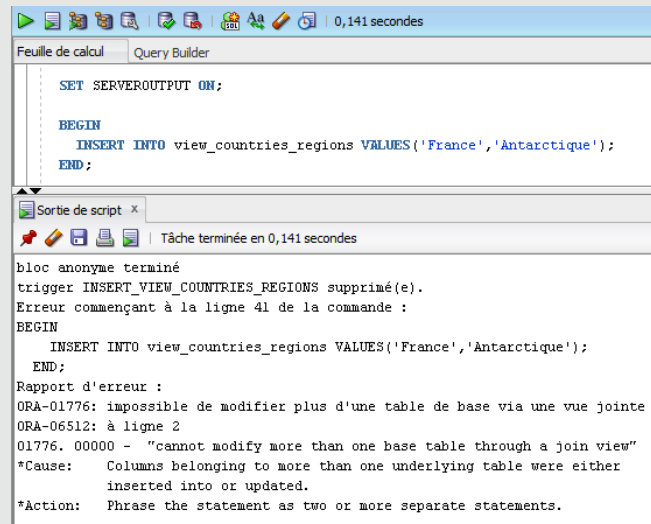
Commentez

Et on va retenter d'insérer dans notre vue

```
#####
#####
#                               ==> Amélioration Continue <==
#
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:                 #
#                               bit.ly/eni_ac
#
#####
#####
```

Les déclencheurs de base de données

La suppression d'un trigger



The screenshot shows the Oracle SQL Developer interface. The top toolbar includes icons for running, saving, and other database operations, along with a timer showing 0,141 secondes. Below the toolbar, there are two tabs: 'Feuille de calcul' and 'Query Builder'. The 'Query Builder' tab is active, displaying a SQL script:

```
SET SERVEROUTPUT ON;

BEGIN
  INSERT INTO view_countries_regions VALUES('France','Antarctique');
END;
```

Below the script, there is a 'Sortie de script' (Script Output) window. It shows the execution results and an error message:

```
bloc anonyme terminé
trigger INSERT_VIEW_COUNTRIES_REGIONS supprimé(e).
Erreur commençant à la ligne 41 de la commande :
BEGIN
  INSERT INTO view_countries_regions VALUES('France','Antarctique');
END;
Rapport d'erreur :
ORA-01776: impossible de modifier plus d'une table de base via une vue jointe
ORA-06512: à ligne 2
01776. 00000 - "cannot modify more than one base table through a join view"
*Cause:      Columns belonging to more than one underlying table were either
             inserted into or updated.
*Action:     Phrase the statement as two or more separate statements.
```

Et évidemment ça plante 😊

Les déclencheurs de base de données

Démonstration



Interdire la réduction des salaires

Conclusion

- Vous comprenez le mécanisme des triggers
- Vous savez créer des triggers de table
- Vous savez créer des triggers de vue



Conclusion – Vidéo 6

Conclusion !

Une fonction est un programme contenant des traitements et qui retourne un seul résultat.

Ça permet de faire des calculs complexes

Optimiser les calculs

Augmenter l'efficacité de vos requêtes !

[[DIAPO 22](#)]

[Vous comprenez le mécanisme des triggers](#)

[Vous savez créer des triggers de table](#)

[Vous savez créer des triggers de vue](#)

```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:                #  
#                               bit.ly/eni_ac  
#                               #
```


#####

PL / SQL

Module 10 – Les packages



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```


Objectifs

- Comprendre l'utilité des packages
- Savoir créer un package
- Savoir utiliser un élément de package



[DIAPO]

Les objectifs sont les suivants :

- Comprendre l'utilité des packages.
- Savoir créer un package.
- Savoir utiliser un élément de package.

Les packages

Définition

- Objet du schéma
- Permet d'améliorer la gestion des objets PL/SQL
- Permet de regrouper un ensemble d'objets homogènes



[DIAPO]

Un package est une structure PL/SQL qui permet de stocker un ensemble de plusieurs objets formant un ensemble homogène de services logiquement associés. Il permet d'utiliser des objets publics ou privés, et s'apparente au concept de classe en programmation.

- Un package est un objet SQL faisant parti du schéma de l'utilisateur.
- Permet d'améliorer la gestion des objets PL/SQL.
- Permet de stocker un ensemble d'objets homogènes.

Les packages

Les avantages

- Structuration du développement
- Conception orientée objet
- Développement des composants
- Amélioration des performances



[INTRO]

Et quel est l'intérêt d'utiliser des packages ?

[DIAPO]

-Structuration du développement

bla bla bla

-Conception orientée objet

bla bla bla

-Développement des composants

bla bla bla

-Amélioration des performances

bla bla bla

Les packages

Les spécificités

- Composé de deux parties distinctes
 - Spécification
 - Corps
- Éléments pouvant être encapsulés
 - Variable
 - Constante
 - Exception
 - Procédure
 - Fonction



Bien évidemment les packages ont quelques spécificités :

Composé de deux parties distinctes

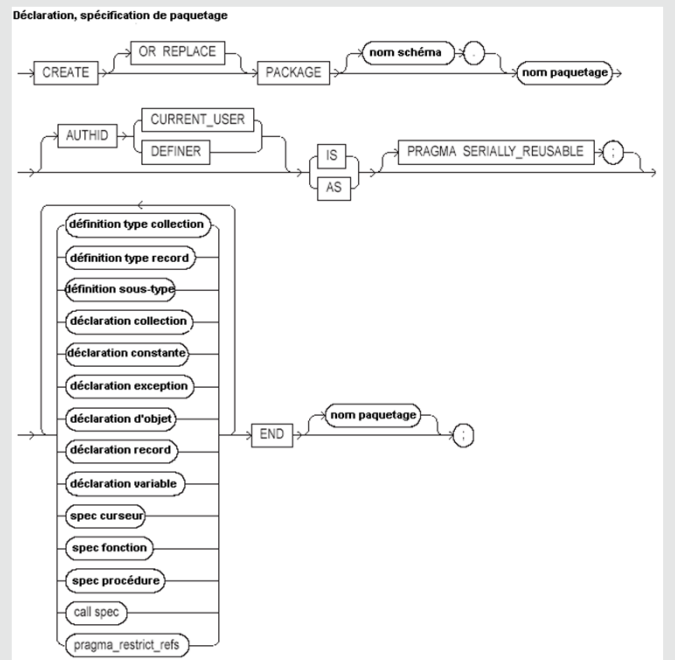
- Spécification
- Corps

Éléments pouvant être encapsulés

- Variable
- Constante
- Exception
- Procédure
- Fonction

Les packages

La syntaxe de création de la partie spécification



Pour découvrir la syntaxe des packages quoi de mieux que la documentation Oracle.

On va voir tout d'abord la syntaxe de l'entête

[Commentez](#)

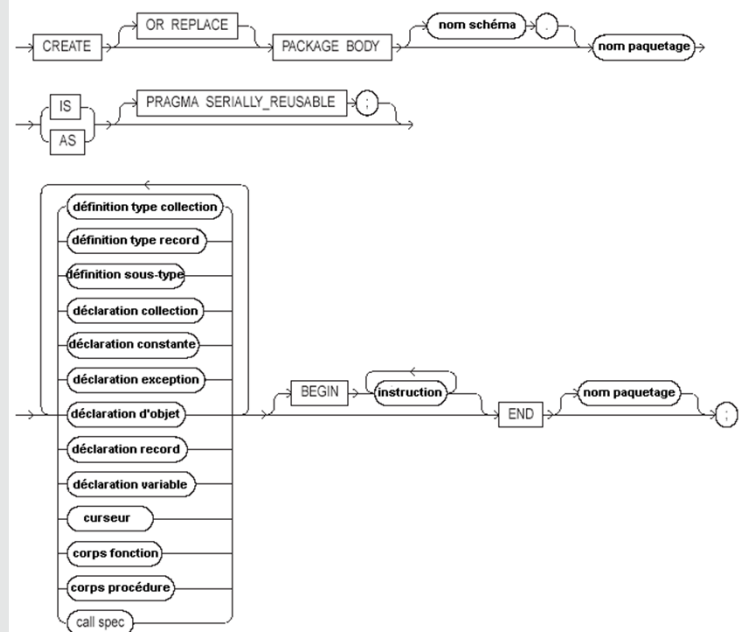
Maintenant la syntaxe du corps...

Les packages

La syntaxe de création de la partie corps



Corps de paquetage



[Commentez](#)

Le sens de création est très important. On doit obligatoirement commencer par l'entête du package.

Les packages

Le processus de création

1. Créer la partie Spécification
2. Créer la partie Corps



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

[INTRO]

Je récapitule le process :

[DIAPO]

1. Créer la partie Spécification
2. Créer la partie Corps

Les packages

Exemple d'une partie spécification

```
CREATE OR REPLACE PACKAGE pkg_regions
IS
    nom_region regions.region_name%TYPE;

    nom_region_default CONSTANT regions.region_name%TYPE := 'Europe';

    FUNCTION get_nombre_regions RETURN NUMBER;

    PROCEDURE formater_regions;
END;
```



[DIAPO]

Commentez

Voilà, maintenant allons voir concrètement ce que cela donne :

Les packages

Exemple d'une partie corps

```
CREATE OR REPLACE PACKAGE BODY pkg_regions
IS
    FUNCTION get_nombre_regions RETURN NUMBER
    IS
        total NUMBER;
    BEGIN
        SELECT COUNT(*) INTO total FROM regions;
        RETURN total;
    END;

    PROCEDURE formater_regions
    IS
    BEGIN
        UPDATE regions SET region_name = UPPER(region_name);
    END;
END;
```

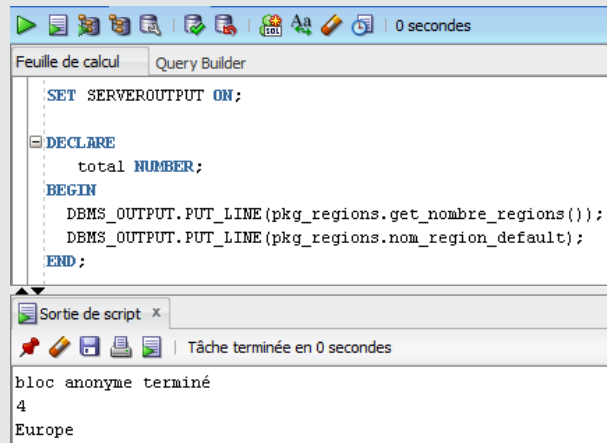


[Commentez](#)

Avec ce code nous avons un package complet il ne nous reste plus qu'à l'utiliser.

Les packages

L'appel à un composant de package



```
SET SERVEROUTPUT ON;

DECLARE
    total NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE(pkg_regions.get_nombre_regions());
    DBMS_OUTPUT.PUT_LINE(pkg_regions.nom_region_default);
END;
```

Sortie de script x

Tâche terminée en 0 secondes

bloc anonyme terminé
4
Europe

[Commentez](#)

Magnifique.

On va finir avec une petite futilité.



Les packages

La suppression d'un package

```
DROP PACKAGE pkg_regions;
```



Voilà comment supprimer un package.

Bon j'ai terminé mes explications sur comment développer un package.

Conclusion

- Vous comprenez l'utilité des packages
- Vous savez créer un package
- Vous savez supprimer un package
- Vous savez utiliser un package



[INTRO]

Les packages vous aiderons à bien structurer vos Dev, et vous avez bien vu comment faire.

[DIAPO]

Faisons maintenant le point :

- Vous comprenez l'utilité des packages
- Vous savez créer un package
- Vous savez supprimer un package
- Vous savez utiliser un package

```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####
```

#####

PL / SQL

Module 11 – Le SQL dynamique



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```

Objectifs

- Savoir expliquer ce qu'est le SQL dynamique
- Savoir quels sont les avantages du SQL dynamique
- Savoir utiliser le SQL dynamique



[DIAPO]

Donc qu'allons-nous voir dans ce module ?

- Savoir expliquer ce qu'est le SQL dynamique
- Savoir quels sont les avantages du SQL dynamique
- Savoir utiliser le SQL dynamique

Le SQL dynamique

Les spécificités

- Permet de créer des traitements génériques
- Permet de construire dynamiquement des requêtes



[DIAPO]

Le Sql dynamique permet d'écrire de fantastiques fonctionnalités générique et réutilisable.

Avec le SQL standard nous avons seulement la possibilité de paramétrer la valeur des paramètres, et bien avec le Sql dynamique vous pourrez modeler votre requête comme vous l'entendez.

Récapitulons :

- Permet de créer des traitements génériques
- Permet de construire des requêtes

Maintenant on va entrer dans le vif du sujet pour découvrir comment faire. Mais avant toute chose j'aimerais vous parler d'une nouvelle fonctionnalité, la clause returning !

La clause RETURNING

```
CREATE TABLE employees_temp
AS SELECT employee_id, first_name, last_name
FROM employees;

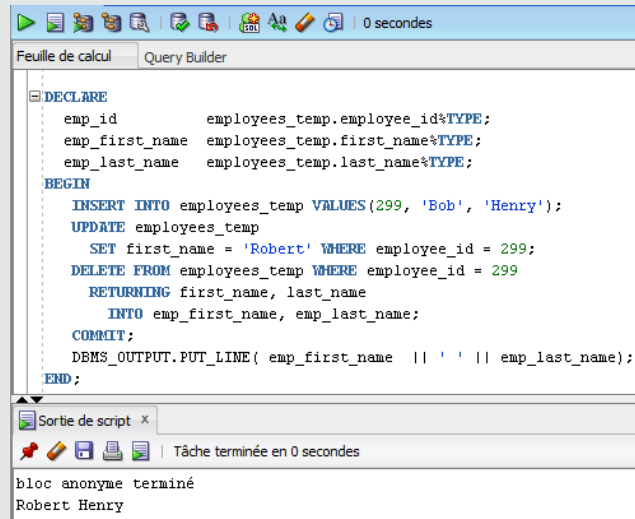
DECLARE
emp_id          employees_temp.employee_id%TYPE;
emp_first_name  employees_temp.first_name%TYPE;
emp_last_name   employees_temp.last_name%TYPE;
BEGIN
INSERT INTO employees_temp VALUES(299, 'Bob', 'Henry');
UPDATE employees_temp
SET first_name = 'Robert' WHERE employee_id = 299;
DELETE FROM employees_temp WHERE employee_id = 299
RETURNING first_name, last_name
INTO emp_first_name, emp_last_name;
COMMIT;
DBMS_OUTPUT.PUT_LINE( emp_first_name || ' ' || emp_last_name);
END;
```



```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:                 #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

[Commentez le code](#)

Exemple d'utilisation de la clause RETURNING



The screenshot shows a SQL Query Builder window with a script editor and a results pane. The script is as follows:

```
DECLARE
emp_id      employees_temp.employee_id%TYPE;
emp_first_name employees_temp.first_name%TYPE;
emp_last_name employees_temp.last_name%TYPE;
BEGIN
INSERT INTO employees_temp VALUES(299, 'Bob', 'Henry');
UPDATE employees_temp
SET first_name = 'Robert' WHERE employee_id = 299;
DELETE FROM employees_temp WHERE employee_id = 299
RETURNING first_name, last_name
INTO emp_first_name, emp_last_name;
COMMIT;
DBMS_OUTPUT.PUT_LINE( emp_first_name || ' ' || emp_last_name);
END;
```

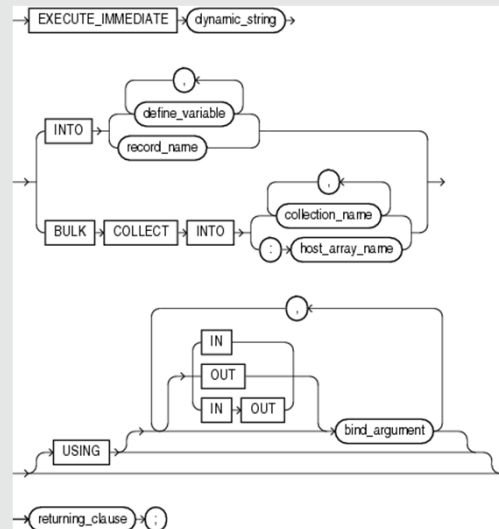
The results pane at the bottom shows the output of the script:

```
bloc anonyme terminé
Robert Henry
```

Commentez le résultat

Le SQL dynamique

La syntaxe de l'instruction EXECUTE IMMEDIATE



Bon maintenant que nous sommes tous d'accord et raccord niveau connaissance voyons la syntaxe du SQL dynamique. N'ayez pas peur cela se limite à deux clauses :`EXECUTE IMMEDIATE` et `RETURNING`

[Commentez](#)

Et maintenant notre premier exemple de SQL dynamique.

Le SQL dynamique

Exemple d'utilisation du EXECUTE IMMEDIATE

```
DECLARE
  requete VARCHAR2(256) ;
  nouveau_nom VARCHAR(20) := 'Jojo';
  id_employe employees.employee_id%TYPE := 100;
BEGIN
  requete := 'UPDATE employees SET FIRST_NAME = :1 WHERE employee_id = :2';
  EXECUTE IMMEDIATE requete USING nouveau_nom, id_employe;
END ;
```



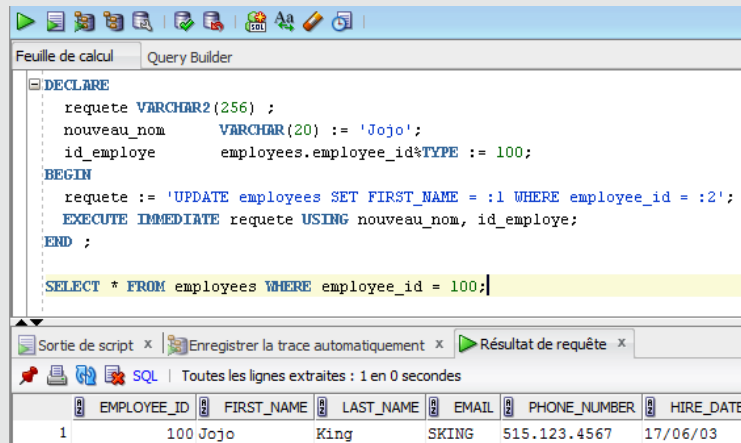
```
#####
#####
#                               ==> Amélioration Continue <==
#                               #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à
cette adresse:                 #
#                               bit.ly/eni_ac
#                               #
#####
#####
```

[Commentez](#)

Pour vous garantir que cela fonctionne bien voici le résultat :

Le SQL dynamique

Exemple d'utilisation du EXECUTE IMMEDIATE



The screenshot shows the Oracle SQL Developer interface. The main window is titled 'Feuille de calcul' and 'Query Builder'. It contains a SQL script with the following code:

```
DECLARE
  requete VARCHAR2(256) ;
  nouveau_nom VARCHAR(20) := 'Jojo';
  id_employe employees.employee_id%TYPE := 100;
BEGIN
  requete := 'UPDATE employees SET FIRST_NAME = :1 WHERE employee_id = :2';
  EXECUTE IMMEDIATE requete USING nouveau_nom, id_employe;
END ;

SELECT * FROM employees WHERE employee_id = 100;
```

Below the script, there are tabs for 'Sortie de script', 'Enregistrer la trace automatiquement', and 'Résultat de requête'. The 'Résultat de requête' tab is active, showing the results of the SELECT statement. The results are displayed in a table with the following columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, and HIRE_DATE. The table contains one row with the following data:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE
1	100 Jojo	King	SKING	515.123.4567	17/06/03

Commentez

Et pour finir je vous montre un dernier exemple ou j'ai pu mettre le nom de la table en paramètre.

Le SQL dynamique

Exemple d'utilisation du EXECUTE IMMEDIATE

```
SET SERVEROUTPUT ON;

DECLARE
    total number;
    table_recherche VARCHAR(30) := 'regions';
    requete VARCHAR(100);
BEGIN
    requete := 'SELECT COUNT(*) FROM ' || table_recherche;
    EXECUTE IMMEDIATE requete INTO total;
    DBMS_OUTPUT.PUT_LINE(total);
END ;
```



[Commentez](#)

Voilà ce sera tout pour les exemples d'utilisation sur le SQL dynamique !

Conclusion

- Vous savez expliquer ce qu'est le SQL dynamique
- Vous savez quels sont les avantages du SQL dynamique
- Vous savez utiliser le SQL dynamique



Vous voici arrivé au moment de faire le point sur ce module de formation.

- Vous savez expliquer ce qu'est le SQL dynamique
- Vous savez quels sont les avantages du SQL dynamique
- Vous savez utiliser le SQL dynamique

```
#####  
#####  
#                ==> Amélioration Continue <==  
#  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:      #  
#                bit.ly/eni_ac  
#  
#####  
#####
```

PL / SQL

Module 12 – Les transactions



```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:               #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```


Objectifs

- Savoir expliquer ce qu'est une transaction
- Savoir gérer des transactions
- Savoir expliquer ce qu'est une transaction autonome



[INTRO]

Module de formation d'un concept simple et immensément important.
Les transactions.

[DIAPO]

Donc qu'allons-nous voir dans ce module ?

- Savoir expliquer ce qu'est une transaction
- Savoir gérer des transactions
- Savoir expliquer ce qu'est une transaction autonome

Définition

- Gère un ensemble d'instructions du DML.
- S'assure que toutes les instructions ont été effectuées avant de les appliquer définitivement.
- Permet un retour en arrière en cas d'erreur.



[INTRO]

Vous avez souvent dû effectuer une série de requêtes sur une base de données et vous connaissez les risques que vous encourez si l'une d'entre elles échoue et qu'une autre réussit. Vous risquez d'avoir des données incohérentes et de perdre du temps ensuite pour trouver d'où vient le problème. Et c'est pour éviter ce type de désagrément il existe les transactions.

[DIAPO]

Gère un ensemble d'instructions du DML.
S'assurer que toutes les instructions ont été effectuées avant de les appliquer définitivement.
Permettre un retour en arrière en cas d'erreur.

Les transactions

Les spécificités

- Une seule transaction principale active
- Il y a toujours une transaction principale active en cours
- Le démarrage d'une transaction principale est automatique
- La transaction peut être validée ou invalidée
- La transaction permet d'assurer la cohérence des données vues par chaque utilisateur connecté



[DIAPO]

Une seule transaction principale active

Il y a toujours une transaction principale active en cours

Le démarrage d'une transaction principale est automatique

La transaction peut être validée ou invalidée

La transaction permet d'assurer la cohérence des données vues par chaque utilisateur connecté

Les transactions

La validation d'une transaction : COMMIT

Enregistre en base toutes les insertions, modifications et suppressions.

Tant qu'il n'y a pas eu COMMIT, **seule la connexion courante** voit ses mises à jour.



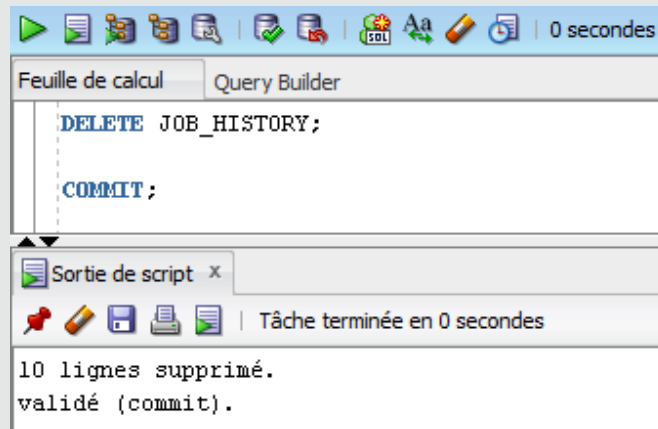
[DIAPO]

Enregistre en base toutes les insertions, modifications et suppressions.

Tant qu'il n'y a pas eu COMMIT, **seule la connexion courante** voit ses mises à jour.

Les transactions

Exemple de validation d'une transaction



```
#####  
#                ==> Amélioration Continue <==                #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à cette adresse:  
#  
#                bit.ly/eni_ac  
#  
#####
```

[DIAPO]

Requête DDL implique donc une validation de transaction.

Les transactions

L'invalidation d'une transaction : ROLLBACK

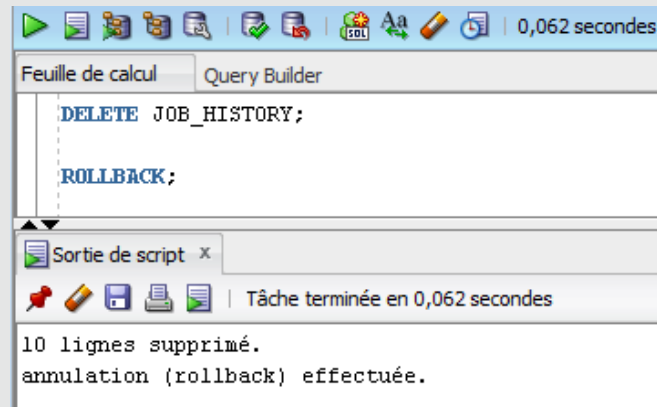
Annule toutes les insertions, modifications et suppressions depuis le début de la transaction.



Annule toutes les insertions, modifications et suppressions depuis le début de la transaction.

Les transactions

Exemple d'invalidation d'une transaction



[Commentez l'exemple](#)

Les transactions

Les transactions autonomes

- Transaction devant être exécutée indépendamment de la transaction appelante.
- La transaction appelante est suspendue temporairement.
- La transaction autonome est contrôlée par les ordres COMMIT et ROLLBACK.
- Mécanisme activé grâce à la directive de compilation (**pragma**) **AUTONOMOUS_TRANSACTION**.



[DIAPO]

Transaction devant être exécutée indépendamment de la transaction appelante.

La transaction appelante est suspendue temporairement.

La transaction autonome est contrôlée par les ordres COMMIT et ROLLBACK.

Mécanisme activé grâce à la directive de compilation (**pragma**) **AUTONOMOUS_TRANSACTION**.

Exemple de transaction autonome

```
CREATE OR REPLACE
PROCEDURE add_information(message VARCHAR2) IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO informations (INFORMATIONS_TEXT) VALUES (message);
    COMMIT;
END;
```



```
#####
#                ==> Amélioration Continue <==                #
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à cette adresse:
#                #
#                bit.ly/eni_ac
#                #
#####
```

[DIAPO]

Encadré ligne 1
Encadré ligne 2
Encadré ligne 3
Encadré ligne 4
Encadré ligne 5
Encadré ligne 6
Encadré ligne 7

Exemple de transaction autonome

```
BEGIN
  INSERT INTO regions values (100,'Paradis');
  COMMIT;
  add_information('Region insérée');
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    add_information('La region existe déjà');
    ROLLBACK;
  WHEN OTHERS THEN
    add_information('Erreur inconnue à l''insertion');
    ROLLBACK;
END;
```



[DIAPO]

- Encadré ligne 1
- Encadré ligne 2
- Encadré ligne 3
- Encadré ligne 4
- Encadré ligne 5
- Encadré ligne 6
- Encadré ligne 7
- Encadré ligne 8
- Encadré ligne 9
- Encadré ligne 10
- Encadré ligne 11
- Encadré ligne 12

Conclusion

- Vous savez utiliser les transactions
- Vous savez utiliser les transactions autonomes



[DIAPO]

- Vous savez utiliser les transactions
- Vous savez utiliser les transactions autonomes

```
#####  
#####  
#                               ==> Amélioration Continue <==  
#                               #  
# En cas d'erreur ou proposition d'amélioration du support merci de l'indiquer à  
# cette adresse:                #  
#                               bit.ly/eni_ac  
#                               #  
#####  
#####
```