le cnam

Développement d'applications avec les bases de données

Michel Crucianu

http://cedric.cnam.fr/~crucianm/abd.html

le cnam

Contenu du cours

- PL/SQL
  - Variables, structures de contrôle
  - Curseurs, interaction avec la base
  - Sous-programmes, paquetages
  - Exceptions
  - Transactions
  - Déclencheurs (*triggers*)
- JDBC
  - Généralités
  - Connexion à une base
  - Interaction avec la base, curseurs
  - Procédures stockées, procédures externes
- Relationnel ↔ objet

le cnam

Bibliographie

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

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

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

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

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

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

→ Suite : NSY135 Applications orientées données - patrons, frameworks, ORM

le cnam

PL/SQL

- *Procedural Language* / *Structured Query Language* (PL/SQL) : langage propriétaire Oracle
- Syntaxe de PL/SQL inspirée du langage Ada
- D'autres éditeurs de SGBDR utilisent des langages procéduraux similaires
- PL/SQL n'est pas très éloigné du langage normalisé *Persistent Stored Modules* (PSM)
- Documentation Oracle (en anglais) :

http://download.oracle.com/docs/cd/B10501_01/appdev.920/a96624/toc.htm

le cnam

Quel est l'intérêt de PL/SQL ?

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

… mais les applications complexes exigent plus :

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

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

le cnam

Structure d'un programme

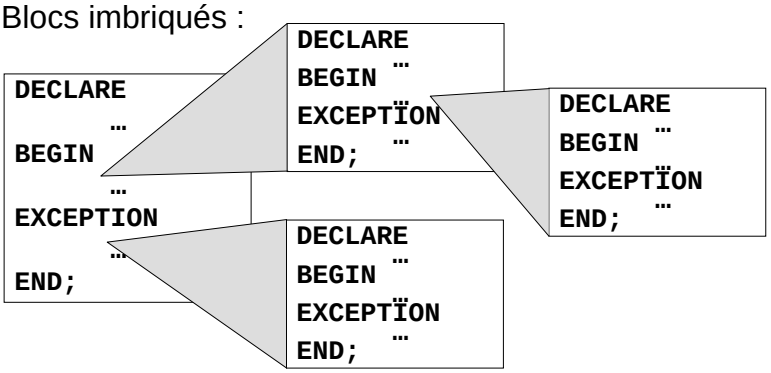
- Programme PL/SQL = bloc (procédure anonyme, procédure nommée, fonction nommée) :

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

le cnam

Structure d'un programme (2)

- Blocs imbriqués :



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

le cnam

Identificateurs, commentaires

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

- Commentaires :

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

le cnam

Variables

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

le cnam

Variables scalaires

- Syntaxe de déclaration :
  Identificateur [CONSTANT] type [[NOT NULL] {:= | DEFAULT} expression];
  - **CONSTANT** : c'est une constante (sa valeur ne peut pas changer, doit être initialisée lors de la déclaration)
  - **NOT NULL** : on ne peut pas lui affecter une valeur nulle (en cas de tentative, une exception **VALUE_ERROR** est levée)
  - Initialisation : affectation **:=** ou **DEFAULT**
- Pas de déclaration multiple dans PL/SQL !
  number1, number2 NUMBER; ← déclaration incorrecte !
- Autre possibilité pour affecter une valeur à une variable
  SELECT … INTO identificateur FROM … ;

le cnam

Nouveaux types PL/SQL

- Nouveaux types prédéfinis :

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

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

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

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

  - Exemple de sous-type utilisateur :

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


le cnam

Base pour les exemples

avion :

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

pilote :

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

vol :

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

le cnam

Variables scalaires : exemple

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


le cnam

Conversions

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

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

le cnam

Quelques conversions implicites

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

le cnam

Déclaration %TYPE

- Déclarer une variable de même type que
  - Une colonne (attribut) d'une table existante :
    ```
    nomNouvelleVariable NomTable.NomColonne%TYPE
    [{:= | DEFAULT} expression];
    ```
  - Une autre variable, déclarée précédemment :
    ```
    nomNouvelleVariable nomAutreVariable%TYPE
    [{:= | DEFAULT} expression];
    ```
- Cette propagation du type permet de réduire le nombre de changements à apporter au code PL/SQL en cas de modification des types de certaines colonnes

le cnam

Déclaration %TYPE : exemple

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


le cnam

Variables %ROWTYPE

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

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

le cnam

Variables %ROWTYPE : exemple

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


le cnam

Variables RECORD

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

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

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

le cnam

Variables **RECORD** : exemple

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


le cnam

Variables **TABLE**

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

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

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

le cnam

Variables **TABLE** : exemple

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


le cnam

Variables de substitution

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

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

le cnam

Variables hôtes, variables de session

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

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

23 décembre 2010 NFA011 25

le cnam

Résolution des noms

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

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

23 décembre 2010 NFA011 26

le cnam

Entrées et sorties

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

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

  - Mise dans le tampon du caractère fin de ligne :

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

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

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

- Rendre les sorties possibles avec SQL*Plus :

```
SET SERVEROUT ON
```

(ou **SET SERVEROUTPUT ON**)

le cnam

Entrées et sorties (2)

  - Lecture d'une ligne :

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

  - Lecture de plusieurs lignes :

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

- Exemple :

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

le cnam

Entrées et sorties (3)

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

le cnam

Structures de contrôle

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

le cnam

Structures conditionnelles : **IF**

- Syntaxe :

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

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

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

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

le cnam

IF : exemple

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

le cnam

Structures conditionnelles : **CASE**

- Syntaxe avec expressions :

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

- Syntaxe avec conditions (*searched case*) :

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

- Le premier cas valide est traité, les autres sont ignorés
- Aucun cas valide : exception **CASE_NOT_FOUND** levée !

le cnam

CASE : exemple

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

le cnam

Structures répétitives : tant que

- Syntaxe :

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

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

le cnam

Tant que (**WHILE**) : exemple

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

le cnam

Structures répétitives : répéter

- Syntaxe :

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

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

le cnam

Répéter : exemple

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

le cnam

Répéter : boucles imbriquées

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


le cnam

Structures répétitives : pour (**FOR**)

- Syntaxe :

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

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

le cnam

Pour (**FOR**) : exemple

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


le cnam

Interaction avec la base

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

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

- Manipulation des données :
  - Insertions :

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

  - Modifications :

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

  - Suppressions :

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

le cnam

Curseurs

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

le cnam

Curseurs implicites

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

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

le cnam

Curseurs explicites : instructions

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

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

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

```
OPEN nomCurseur;
```

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

```
FETCH nomCurseur INTO listeVariables;
```

- Fermeture du curseur (avec libération zone mémoire) :

```
CLOSE nomCurseur;
```


le cnam

Curseurs explicites : attributs

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

```
IF nomCurseur%ISOPEN THEN … END IF;
```

- **nomCurseur%FOUND** : **TRUE** si le dernier accès (**FETCH**) a retourné une ligne

```
IF nomCurseur%FOUND THEN … END IF;
```

- **nomCurseur%NOTFOUND** : **TRUE** si le dernier accès (**FETCH**) n'a pas retourné de ligne (fin curseur)

```
IF nomCurseur%NOTFOUND THEN … END IF;
```

- **nomCurseur%ROWCOUNT** : nombre total de lignes traitées jusqu'à présent (par ex. nombre de **FETCH**)

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

le cnam

Curseurs explicites : exemple

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


le cnam

Curseurs explicites paramétrés

- Objectif : paramétrer la requête associée à un curseur (procédures, éviter de multiplier les curseurs similaires)
- Syntaxe de définition :
  CURSOR nomCurseur(param1[, param2, …]) IS …;
  avec, pour chaque paramètre,
  nomPar [IN] type [{:= | DEFAULT} valeur];
  (**nomPar** est inconnu en dehors de la définition !)
- Les valeurs des paramètres sont transmises à l'ouverture du curseur :
  OPEN nomCurseur(valeurPar1[, valeurPar2, …]);
- Il faut évidemment fermer le curseur avant de l'appeler avec d'autres valeurs pour les paramètres

le cnam

Curseurs paramétrés : exemple

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


le cnam

Boucle **FOR** avec curseur

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

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

le cnam

Boucle **FOR** avec curseur : exemple

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


le cnam

Curseurs et verrouillage

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

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

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

le cnam

Modification des lignes verrouillées

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

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

- Suppression de la ligne courante d'un curseur :

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


le cnam

Modification des lignes : exemple

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

le cnam

Variables curseurs

- Éviter de manipuler de nombreux curseurs associés à des requêtes différentes : définir des variables curseurs, associées dynamiquement aux requêtes
- Syntaxe de déclaration :
  ```
  TYPE nomTypeCurseur IS REF CURSOR [RETURN type];
  nomVariableCurseur nomTypeCurseur;
  ```
- Le curseur est typé si **RETURN type** est présente (en général, **type** est **nomDeTable%ROWTYPE**) ; ne peut être associé qu'aux requêtes ayant le même type de retour
- Association à une requête et ouverture :
  ```
  OPEN nomVariableCurseur FOR
      SELECT … FROM … WHERE …;
  ```


le cnam

Variables curseurs : exemple

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

le cnam

Sous-programmes

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

le cnam

Procédures

- Syntaxe :

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

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

le cnam

Fonctions

- Syntaxe :

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

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

le cnam

Paramètres de sous-programme

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

le cnam

Définition de sous-programme

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

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

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

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


le cnam

Manipulation de sous-programme

- Création ou modification de sous-programme :

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

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

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

  - Affichage des erreurs de compilation sous SQL*Plus :

```
SHOW ERRORS
```

- Suppression de sous-programme :

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

le cnam

Appel de sous-programme

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

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

le cnam

Procédure locale : exemple

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

le cnam

Procédure stockée : exemple

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

- Appel depuis SQL*Plus :

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

- Appel depuis un bloc PL/SQL :

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

23 décembre 2010 NFA011 65

le cnam

Fonction locale : exemple

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

23 décembre 2010 NFA011 66

le cnam

Fonction stockée : exemple

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

- Appel depuis SQL*Plus :

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

- Appel depuis un bloc PL/SQL :

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


le cnam

Paquetages

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

le cnam

Section de spécification

- Syntaxe :

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


le cnam

Spécification : exemple

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

le cnam

Section d'implémentation

- Syntaxe :

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


le cnam

Implémentation : exemple

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

le cnam

Référence au contenu d'un paquetage

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

```
nomPaquetage.nomObjet
nomPaquetage.nomSousProgramme(…)
```

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

le cnam

Manipulation d'un paquetage

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

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

le cnam

Exceptions

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

le cnam

Traitement des exceptions

- Syntaxe :

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

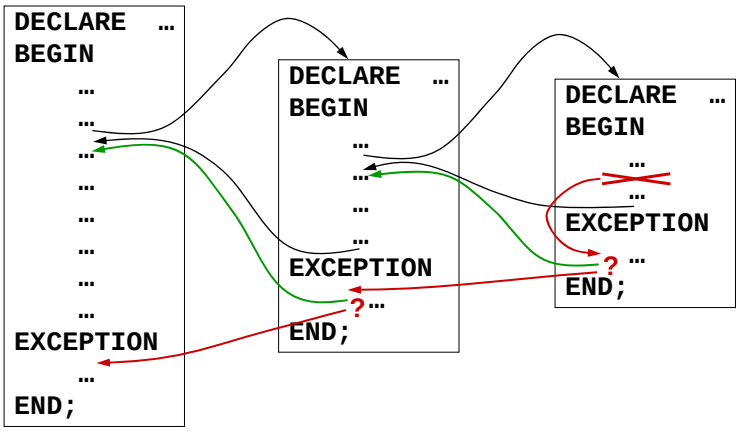
le cnam

Suites de l'apparition d'une exception

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

le cnam

Suites de l'apparition d'une exception



le cnam

Mécanismes de déclenchement

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

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

```
RAISE nomExceptionPreDefinie;
```


le cnam

Exceptions non nommées

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

```
WHEN OTHERS THEN instructions;
```

  Exemple :

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

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

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

  - Traitement spécifique :

```
WHEN nomAPrendre THEN instructions;
```

le cnam

Mécanismes de déclenchement (2)

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

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

- Déclenchement avec message et code d'erreur personnalisé :

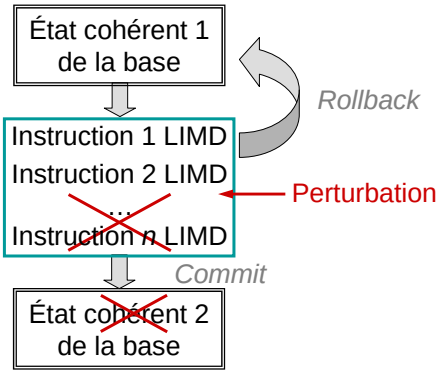
```
RAISE_APPLICATION_ERROR(numErr, messageErr,
                                [TRUE | FALSE]);
```

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

le cnam

Transactions

- Objectif : rendre indivisible une suite de manipulations des données



le cnam

Transactions : exigences

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

le cnam

Transactions : contrôle

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

le cnam

Transactions : contrôle (2)

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

  SAVEPOINT nomPoint; -- insertion point de validation

  ROLLBACK TO nomPoint; -- retour à l'état d'avant **nomPoint**

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

le cnam

Transactions : exemple

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

le cnam

Déclencheurs

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

le cnam

Définition d'un déclencheur

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

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

le cnam

Déclencheurs sur instruction LMD

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

le cnam

Déclencheurs sur instruction LMD

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

le cnam

Base exemple

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

```
immeuble (Adr, NbEtg, DateConstr, NomGerant)
appart (Adr, Num, Type, Superficie, Etg, NbOccup)
personne (Nom, Age, CodeProf)
occupant (Adr, NumApp, NomOccup, DateArrivee,
          DateDepart)
propriete (Adr, NomProprietaire, QuotePart)
```

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

le cnam

Déclencheur sur **INSERT**

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

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

- Événement déclencheur : **INSERT INTO occupant … VALUES …;**

le cnam

Déclencheur sur **INSERT (2)**

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

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

- Événement déclencheur : **INSERT INTO immeuble … VALUES …;**

le cnam

Déclencheur sur **DELETE**

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

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

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

le cnam

Déclencheur sur **UPDATE**

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

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

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

23 décembre 2010 NFA011 95

le cnam

Déclencheur sur conditions multiples

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

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

- Exemple d'événement déclencheur :
  INSERT INTO occupant … VALUES …;

23 décembre 2010 NFA011 96

le cnam

Déclencheurs sur instruction LDD

- Syntaxe pour déclenchement sur instruction LDD :

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

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

le cnam

Déclencheur sur LDD : exemple

- Enregistrement des changements de noms des objets du dictionnaire :

```
historiqueChangementNoms(Date,NomObjet,NomProprietaire)

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

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

le cnam

Déclencheurs d'instance

- Syntaxe :

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

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

le cnam

Déclencheur d'instance : exemple

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

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

le cnam

Manipulation d'un déclencheur

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

```
ALTER TRIGGER nomDeclencheur COMPILE;
```

- Désactivation de déclencheurs :

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

- Réactivation de déclencheurs :

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

- Suppression d'un déclencheur :

```
DROP TRIGGER nomDeclencheur;
```


le cnam

Droits de création et manipulation

- Déclencheurs d'instance : privilège

```
ADMINISTER DATABASE TRIGGER
```

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

    (rôle **RESOURCE**)

le cnam

ALL_TRIGGERS, DBA_TRIGGERS, USER_TRIGGERS

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

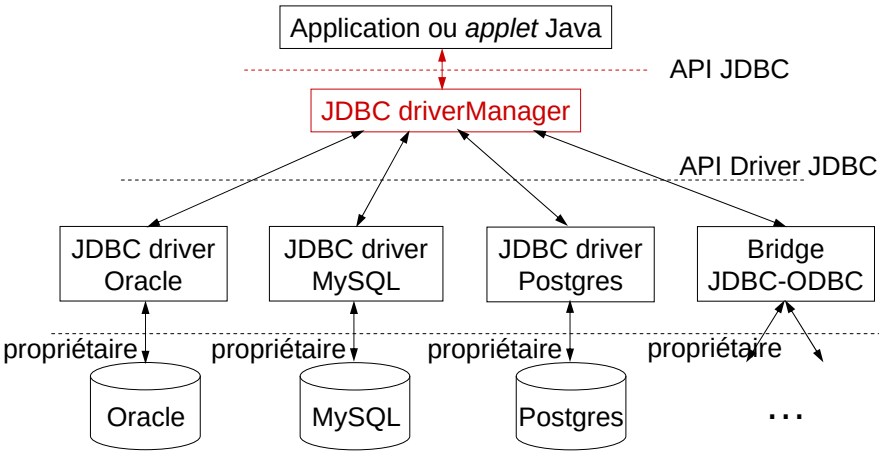
le cnam

JDBC

- *Java DataBase Connectivity* (JDBC) : API de bas niveau permettant de travailler avec une ou plusieurs bases de données depuis un programme Java
- Objectif : interface uniforme assurant l'indépendance du SGBDR cible
- Réalité : indépendance relative du SGBDR, l'interface étant assurée par un pilote (*driver*) fourni par l'éditeur du SGBDR ou par un tiers…
- Versions :
  - JDBC 1 : SDK 1.1 (java.sql)
  - JDBC 2 : SDK 1.2 (java.sql, javax.sql)
  - JDBC 3 : SDK 1.4

le cnam

JDBC : architecture



le cnam

Taxonomie des pilotes (JavaSoft)

- Type 1 : JDBC-ODBC *Bridge*
  - Utilisation comme intermédiaire de *Open DataBase Connectivity* (ODBC) de Microsoft
  - JDK : **`sun.jdbc.odbc.JdbcOdbcDriver`**
  - Exige l'exécution par le client (dit « épais ») de code non Java, ne peut donc pas être utilisé par les *applets*
- Type 2 : *Native-API Partly-Java Driver*
  - Appelle des fonctions natives (non Java) de l'API du SGBDR
  - Fournis par les éditeurs des SGBDR ou par des tiers
  - Exige l'exécution par le client (dit « épais ») de code non Java, ne peut donc pas être utilisé par les *applets*

le cnam

Taxonomie des pilotes (JavaSoft)

- Type 3 : *Net-protocol, all Java driver*
  - Utilise, à travers une API réseau générique, un serveur *middleware* comme intermédiaire avec le SGBDR
  - Client « léger » 100% Java, peut donc être employé par les *applets* à condition que l'adresse du *middleware* soit la même que celle du serveur Web
- Type 4 : *Native protocol, all Java driver*
  - Interagit avec le SGBDR directement à travers des *sockets*
  - Fournis par les éditeurs des SGBDR ou par des tiers
  - Client « léger » 100% Java, peut donc être employé par les *applets* à condition que l'adresse du SGBDR soit la même que celle du serveur Web

le cnam

Modèles d'interaction

- Modèle 2-*tiers* : interaction directe entre le client *applet*/application Java et le SGBDR
  - Avantage : facilité de mise en œuvre
  - Désavantages : tout le traitement est du côté du client, dépendance forte entre le client et le SGBDR
- Modèle 3-*tiers* : interaction par l'intermédiaire d'un serveur *middleware*
  - Avantages : une partie du traitement peut être transférée au serveur middleware, flexibilité pour l'interaction avec le client, le client peut être totalement indépendant du SGBDR
  - Désavantages : difficulté de mise en œuvre, exige des compétences plus vastes

le cnam

Structure de l'application Java

1. Importation de paquetages
2. Enregistrement du pilote
3. Établissement des connexions
4. Préparation des instructions SQL
5. Exécution des instructions SQL
6. Traitement des données retournées
7. Fermeture

le cnam

Importation de paquetages

- Importation du paquetage de base JDBC (obligatoire) :
  ```
  import java.sql.*;
  ```
- Importation de paquetages spécifiques :
  - Additions au paquetage de base :
    ```
    import javax.sql.*;
    ```
  - Paquetage permettant d'utiliser des types spécifiques Oracle :
    ```
    import oracle.sql.*;
    ```
  - Paquetage contenant le pilote Oracle (obligatoire) :
    ```
    import oracle.jdbc.driver.*;
    ```
- **`CLASSPATH`** doit inclure le paquetage à employer (en fonction du pilote choisi, par exemple **`ojdbc14.jar`**)

le cnam

Enregistrement du pilote

- Chargement de la classe du pilote, qui crée une instance et s'enregistre auprès du **`DriverManager`** (pour tous les types de pilotes) :

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

ou

- Création explicite d'une instance et enregistrement explicite (pilotes de type 2, 3 ou 4) :

```
DriverManager.registerDriver(new
    oracle.jdbc.driver.OracleDriver());
```


le cnam

Établissement de connexion

- Connexion = instance de la classe qui implémente l'interface **`Connection`**
- Appel de la méthode **`getConnection`**(URLconnexion, *login*, *password*) de la classe **`DriverManager`**, avec URLconnexion = "**`jdbc:`**sousProtocole**`:`**identifiantBase"
  - **`jdbc`** : protocole
  - sousProtocole : **`odbc`** pour pilote type 1, **`oracle:oci`** pour pilote Oracle de type 2, **`oracle:thin`** pour pilote Oracle de type 4, etc.
  - identifiantBase : dépend du pilote utilisé ; exemple pour pilote de type 4 : nom de la machine (ou adresse IP) + numéro de port + nom de la base

le cnam

Établissement de connexion (2)

- Exemple :

```
Connection nomConnexion =
    DriverManager.getConnection
          ("jdbc:oracle:thin:@odessa:1521:NFA011",
           "Julien", "monpass");
```

  - Pilote léger de type 4 **`oracle:thin`**
  - Nom machine **`odessa`**
  - Numéro de port **`1521`**
  - Nom de la base **`NFA011`**
- **`DriverManager`** essaye tous les drivers, respectant le sous-protocole indiqué, qui se sont enregistrés (dans l'ordre d'enregistrement) et utilise le premier qui accepte la connexion

le cnam

Options d'une connexion

- Après l'ouverture d'une connexion on peut préciser des options :
  - Lecture seulement ou non : méthode **`setReadOnly(boolean)`** de **`Connection`**

`nomConnexion.setReadOnly(true)`;

  - *Commit* automatique ou non : méthode **`setAutoCommit(boolean)`** de **`Connection`**

    (voir transactions plus loin)

`nomConnexion.setAutoCommit(false)`;

  - Degré d'isolation : méthode **`setTransactionIsolation(…)`** de **`Connection`**

le cnam

Passage des appels SQL

- Pour transmettre un appel SQL il faut commencer par créer une instance de classe qui implémente l'interface correspondante
- Interfaces utilisables pour les appels SQL :
  - Interface **`Statement`** : pour les instructions SQL simples
  - Interface **`PreparedStatement`** : pour les instructions SQL paramétrées (mais peut servir pour les instructions simples)
  - Interface **`CallableStatement`** : pour les procédures ou fonctions cataloguées (PL/SQL ou autres)

le cnam

Interface **`Statement`**

- Création :

```
Statement stmt = connexion.createStatement();
```

- Méthodes :
  - **`ResultSet executeQuery(String)`** : exécute la requête présente dans le **`String`** et retourne un ensemble d'enregistrements (**`ResultSet`**) ; utilisée pour **`SELECT`**
  - **`int executeUpdate(String)`** : exécute la requête présente dans le **`String`** et retourne le nombre d'enregistrements traités (ou 0 pour instructions du LDD) ; utilisée pour **`INSERT`**, **`UPDATE`**, **`DELETE`** (LMD), **`CREATE`**, **`ALTER`**, **`DROP`** (LDD)

le cnam

Interface **`Statement`** (2)

- Méthodes (suite) :
  - **`boolean execute(String)`** : exécute la requête présente dans le **`String`**, retourne **`true`** si c'est un **`SELECT`** et **`false`** sinon ; employée dans des cas particuliers
  - **`Connection getConnection()`** : retourne la connexion correspondante
  - **`void setMaxRows(int)`** : borne supérieure sur le nombre d'enregistrements à extraire par toute requête de l'instance
  - **`int getUpdateCount()`** : nombre d'enregistrements affectés par la dernière instruction SQL associée (-1 si **`SELECT`** ou si l'instruction n'a affecté aucun enregistrement)
  - **`void close()`** : fermeture de l'instance

le cnam

Curseurs statiques

- Le résultat d'une requête est disponible dans une instance de classe qui implémente l'interface **`ResultSet`** (équivalent des curseurs PL/SQL)
- Méthodes de **`ResultSet`** :
  - **`boolean next()`** : positionnement sur l'enregistrement suivant ; retourne **`false`** quand il n'y a plus d'enregistrements
  - **`getXXX(int)`** : retourne la colonne de numéro donné par l'argument **`int`** et de type **`XXX`** de l'enregistrement courant
  - **`updateXXX(int, XXX)`** : dans l'enregistrement courant, donne à la colonne de numéro **`int`** et de type **`XXX`**, une valeur de type **`XXX`**
  - **`void close()`** : fermeture de l'instance
- L'instance est automatiquement fermée quand le **`Statement`** correspondant est fermé ou associé à une autre instruction SQL

le cnam

Curseurs statiques : exemple

```
…
int delCount;
Statement stmt1 = connexion.createStatement();
Statement stmt2 = connexion.createStatement();
ResultSet rset = stmt1.executeQuery("SELECT Nom
                                     FROM pilote");
while (rset.next())
  System.out.println(rset.getString(1));
rset.close();
stmt1.close();
delCount = stmt2.executeUpdate("DELETE FROM vol
                  WHERE Ville_depart = 'Paris'");
stmt2.close();
…
```


le cnam

Curseurs navigables

- Les options du curseur sont déclarées comme paramètres de la méthode **`createStatement`** :
  `createStatement(int typeRSet, int modifRSet)`
- Types possibles (selon paramètre **`typeRSet`**) :
  - **`ResultSet.TYPE_FORWARD_ONLY`** : non navigable (valeur par défaut)
  - **`ResultSet.TYPE_SCROLL_INSENSITIVE`** : navigable mais insensible aux modifications, c'est à dire ne reflète pas les modifications de la base
  - **`ResultSet.TYPE_SCROLL_SENSITIVE`** : navigable et sensible aux modifications (reflète les modifications de la base)

le cnam

Curseurs navigables (2)

- Quelques méthodes spécifiques :
  - **`int getType()`** : retourne le type de navigabilité du curseur
  - **`void setFetchDirection(int)`** : définit la direction du parcours
    - valeurs du paramètre : **`ResultSet.FETCH_FORWARD`**, **`ResultSet.FETCH_BACKWARD`**, **`ResultSet.FETCH_UNKNOWN`**
    - est aussi méthode de **`Statement`**, ayant dans ce cas effet sur tous les curseurs associés !
  - **`int getFetchDirection()`** : retourne la direction courante
  - **`boolean isBeforeFirst()`** : indique si le curseur est positionné avant le premier enregistrement (**`true`** après ouverture, sauf si aucun enregistrement)

le cnam

Curseurs navigables (3)

- Quelques méthodes spécifiques (suite) :
  - **`void beforeFirst()`** : positionne le curseur avant le premier enregistrement
  - **`boolean isFirst()`** : indique si le curseur est positionné sur le premier enregistrement ; si aucun enregistrement : **`false`**
  - **`boolean absolute(int)`** : positionne le curseur sur l'enregistrement de numéro indiqué (depuis début si >0, depuis fin si <0) ; **`false`** si aucun enregistrement n'a ce numéro
  - **`boolean relative(int)`** : positionne le curseur sur le *n*-ième enregistrement en partant de la position courante (>0 ou <0) ; **`false`** si aucun enregistrement n'a cette position

le cnam

Curseurs navigables : exemple

```
…
Statement stmt = connexion.createStatement
             (ResultSet.TYPE_SCROLL_INSENSITIVE,
              ResultSet.CONCUR_READ_ONLY);
ResultSet rset = stmt.executeQuery("SELECT Nom FROM
                                                pilote");
if(rset.absolute(5)) {
   System.out.println("5ème pilote :" +
                      rset.getString(1));
   if(rset.relative(2))
        System.out.println("7ème pilote :" +
                           rset.getString(1));
   else
        System.out.println("Echec, pas de 7ème pilote !");
} else
   System.out.println("Echec, pas de 5ème pilote !");
…
```


le cnam

Curseurs modifiables

- Permettent de modifier le contenu de la base
- Types possibles (selon paramètre **`modifRSet`**) :
  - **`ResultSet.CONCUR_READ_ONLY`** : n'autorise pas la modification (valeur par défaut)
  - **`ResultSet.CONCUR_UPDATABLE`** : autorise la modification
- Contraintes d'utilisation :
  - Pas de verrouillage automatique comme avec **`CURSOR … IS SELECT … FOR UPDATE`** dans PL/SQL !
  - Seules les requêtes qui extraient des colonnes sont autorisées à produire des curseurs modifiables (**`SELECT tableau.*`** plutôt que **`SELECT *`**, pas de **`AVG()`**, …) ; aussi, pas de jointure dans la requête !

le cnam

Curseurs modifiables (2)

- Quelques méthodes spécifiques :
  - **`int getConcurrency()`** : retourne la valeur du paramètre d'autorisation des modifications
  - **`void deleteRow()`** : suppression enregistrement courant
  - **`void updateRow()`** : propagation à la table des modifications de l'enregistrement courant
  - **`void cancelRowUpdates()`** : annulation des modifications de l'enregistrement courant
  - **`void moveToInsertRow()`** : préparation du curseur pour insertion d'un enregistrement
  - **`void insertRow()`** : propagation à la table de l'insertion
  - **`void moveToCurrentRow()`** : retourne à l'enregistrement courant

le cnam

Curseurs modifiables : exemple 1

```
// Exemple de suppression
…
connexion.setAutoCommit(false);
Statement stmt = connexion.createStatement
                 (ResultSet.TYPE_FORWARD_ONLY,
                  ResultSet.CONCUR_UPDATABLE);
ResultSet rset = stmt.executeQuery("SELECT Nom FROM
                                    pilote");
String nomsAEffacer = "Philippe";
while(rset.next()) {
  if(rset.getString(1).equals(nomsAEffacer)) {
      rset.deleteRow();
      connexion.commit(); // valide une suppression
  }
}
// connexion.commit(); // regroupe les suppressions
rset.close();
```

le cnam

Curseurs modifiables : exemple 2

```
// Exemple d'insertion
…
Statement stmt = connexion.createStatement
             (ResultSet.TYPE_SCROLL_INSENSITIVE,
              ResultSet.CONCUR_UPDATABLE);
ResultSet rset = stmt.executeQuery("SELECT Matricule,
                    Nom, Ville, Age, Salaire FROM pilote");
if(rset.absolute(3))
   System.out.println(rset.getString(2));
rset.moveToInsertRow();
rset.updateInt(1,3);
rset.updateString(2,"Philippe");
rset.updateString(3,"Paris");
rset.updateInt(4,36);
rset.updateFloat(5,38000);
rset.insertRow();   // demande l'insertion d'une ligne
connexion.commit(); // valide l'insertion
rset.moveToCurrentRow();
…
```


le cnam

Curseurs modifiables : exemple 3

```
// Exemple de modification
…
Statement stmt = connexion.createStatement
            (ResultSet.TYPE_SCROLL_INSENSITIVE,
             ResultSet.CONCUR_UPDATABLE);
ResultSet rset = stmt.executeQuery("SELECT Nom FROM
                                          pilote");
while(rset.next()) {
  if(rset.getString(1).equals("Philipe")) {
      rset.updateString(1,"Philippe");
      rset.updateRow(); // demande la modification
  }
}
connexion.commit(); // valide la modification
rset.close();
```

le cnam

Interface **`PreparedStatement`**

- Pourquoi : meilleure efficacité (analyse + compilation + planification une seule fois, nombreuses exécutions)
- Création :

```
PreparedStatement prepStmt =
    connexion.prepareStatement(String instrSQL);
```

- **`PreparedStatement`** hérite de **`Statement`**
- Méthodes :
  - **`ResultSet executeQuery()`** : exécute la requête préparée et retourne un ensemble d'enregistrements (**`ResultSet`**) ; pour **`SELECT`**
  - **`int executeUpdate()`** : exécute la requête préparée et retourne le nombre d'enregistrements traités (ou 0 pour instructions du LDD) ; pour **`INSERT`**, **`UPDATE`**, **`DELETE`** (LMD), **`CREATE`**, **`ALTER`**, **`DROP`** (LDD)

le cnam

Interface **`PreparedStatement`** (2)

- Méthodes (suite) :
  - **`boolean execute()`** : exécute la requête préparée, retourne **`true`** si c'est un **`SELECT`** et **`false`** sinon ; employée dans des cas particuliers
  - **`Connection getConnection()`** : retourne la connexion correspondante
  - **`void setMaxRows(int)`** : borne supérieure sur le nombre d'enregistrements à extraire par toute requête de l'instance
  - **`int getUpdateCount()`** : nombre d'enregistrements traités par la dernière instruction SQL associée (-1 si **`SELECT`** ou si l'instruction n'a affecté aucun enregistrement)
  - **`void close()`** : fermeture de l'instance

le cnam

`PreparedStatement` : exemple 1

```
…
String instrSQL = “SELECT Nom FROM pilote”;
PreparedStatement prepStmt1 =
            connexion.prepareStatement(instrSQL);
ResultSet rset = prepStmt1.executeQuery();
while (rset.next())
  System.out.println(rset.getString(1));
rset.close();
prepStmt1.close();
…
```


le cnam

Paramétrage `PreparedStatement`

- Dans la chaîne de caractères qui représente l’instruction SQL on indique par « ? » les champs paramétrés
- Avant l’exécution il faut donner des valeurs aux « paramètres » par des méthodes **`setXXX`** de **`PreparedStatement`** : **`prepStmt.setXXX(numeroPar, valeurPar)`**, où **`numeroPar`** est la position du « ? » correspondant et **`XXX`** est le type du « paramètre »
- Donner la valeur **`NULL`** à un « paramètre » : **`prepStmt.setNull(numeroPar, typePar)`**

le cnam

`PreparedStatement` : exemple 2

```
…
int insCount;
String instrSQL = “INSERT INTO avion
                   VALUES(?,?,?,?)”;
PreparedStatement prepStmt2 =
            connexion.prepareStatement(instrSQL);
prepStmt2.setInt(1,210);
prepStmt2.setInt(2,570);
prepStmt2.setString(3,“A800”);
prepStmt2.setString(4,“Roissy”);
insCount = prepStmt2.executeUpdate();
prepStmt2.close();
…
```


le cnam

Interface `CallableStatement`

- Objectif : appeler des procédures ou fonctions stockées écrites en PL/SQL ou un autre langage
- Création :

```
CallableStatement callStmt =
    connexion.prepareCall(String prepCall);
```

- `CallableStatement` hérite de `PreparedStatement`
- Méthodes :
  - `boolean execute()` : voir `PreparedStatement`
  - `Void registerOutParameter(int,int)` : définit un paramètre de sortie de la procédure appelée ; le premier `int` indique le numéro du paramètre lors de l’appel, le second indique le type du paramètre (suivant `java.sql.Types`)

le cnam

Interface `CallableStatement` (2)

- Méthodes (suite) :
  - **`void setXXX(int, XXX)`** : donner une valeur à un paramètre ; **`int`** est la position du paramètre et **`XXX`** son type
  - **`XXX getXXX(int)`** : extraire la valeur d'un paramètre de sortie (**`OUT`**) ; **`int`** est la position du paramètre
  - **`boolean wasNull()`** : détermine si la valeur du dernier paramètre de sortie extrait est **`NULL`** ; utilisable après un **`get`**
  - **`void close()`** : fermeture de l'instance

le cnam

Formats des appels

- Format du **`String`** pour un appel de procédure stockée :
  `"{call nomProcedure(?,…)}"`
- Format du **`String`** pour un appel de fonction stockée :
  `"{? = call nomFonction(?,…)}"`
  - Pour une fonction, la valeur retournée est vue comme paramètre 1 : déclaré avec **`registerOutParameter`**, récupéré avec **`getXXX`**

le cnam

`CallableStatement` : exemple 1

- Procédure PL/SQL appelée :

```
PROCEDURE
    accordPrime(villePrime IN pilote.Ville%TYPE,
                valPrime IN NUMBER,
                nbPilotes OUT INTEGER);
```

- Appel en Java :

```
String prepCall = "{call accordPrime(?,?,?)}";
CallableStatement callStmt =
                  connexion.prepareCall(prepCall);
callStmt.setString(1,"Paris");
callStmt.setInt(2,500);
callStmt.registerOutParameter(3,
                                java.sql.Types.INTEGER);
callStmt.execute();
nbPilotePrimes = callStmt.getInt(3);
callStmt.close();
```


le cnam

`CallableStatement` : exemple 2

- Fonction PL/SQL appelée :

```
FUNCTION moyInt(ageInf IN pilote.Age%TYPE,
                ageSup IN pilote.Age%TYPE)
    RETURN pilote.Salaire%TYPE;
```

- Appel en Java :

```
String prepCall = "{? = call moyInt(?,?)}";
CallableStatement callStmt =
                  connexion.prepareCall(prepCall);
callStmt.registerOutParameter(1,java.sql.Types.NUMBER);
callStmt.setInt(2,38);
callStmt.setInt(3,55);
callStmt.execute();
moyenneSalaires = callStmt.getInt(1); // résultat
callStmt.close();
```

le cnam

Méta-données

- Objectif : retrouver les propriétés d'une base de données et de ses tables
- Interface **`DatabaseMetaData`** : retrouver l'identification de l'éditeur et de la version, la description des tables présentes, utilisateurs connectés, etc.
- Interface **`ResultSetMetaData`** : pour les tables auxquelles des **`Statement`** ou **`PreparedStatement`** ont accédé, retrouver les nombres, noms, types et autre caractéristiques des colonnes

le cnam

Méthodes de **`DatabaseMetaData`**

- **`String getDatabaseProductName()`** : retourne le nom de l'éditeur du SGBD qui a servi à créer la base
- **`String getDatabaseProductVersion()`** : retourne le numéro de version du SGBD
- **`ResultSet getTables(String, String, String, String[])`** : retourne une description de toutes les tables de la base
- **`String getUserName()`** : retourne le nom de l'utilisateur connecté
- **`boolean supportsSavepoints()`** : retourne **`true`** si la base supporte les points de validation pour les transactions

le cnam

Méthodes de `ResultSetMetaData`

- **`int getColumnCount()`** : retourne le nombre de colonnes de la table
- **`String getColumnName(int)`** : retourne le nom de la colonne **`int`**
- **`int getColumnType(int)`** : retourne le type de la colonne **`int`** (suivant **`java.sql.Types`**)
- **`String getColumnTypeName(int)`** : retourne le nom du type de la colonne **`int`**
- **`int isNullable(int)`** : indique si la colonne **`int`** accepte des valeurs **`NULL`**
- **`int getPrecision(int)`** : indique le nombre de chiffres après la virgule pour la colonne **`int`**

le cnam

Exceptions

- Classe **`SQLException`** qui hérite de la classe Java **`Exception`**
- Méthodes de **`SQLException`** :
  - **`String getMessage()`** : retourne le message décrivant l'erreur
  - **`String getSQLState()`** : retourne le code d'erreur SQL standard
  - **`int getErrorCode()`** : retourne le code d'erreur SQL du SGBD
  - **`SQLException getNextException()`** : retourne l'exception suivante si plusieurs ont été levées

le cnam

Exceptions : exemple

```
…
try {
   Connection connexion =
             DriverManager.getConnection(…);
   String instrSQL = “INSERT INTO avion
                          VALUES(?,?,?,?)”;
   PreparedStatement prepStmt =
             connexion.prepareStatement(instrSQL);
   …
   prepStmt.executeUpdate();
   prepStmt.close();
} catch (SQLException excSQL) {
   while (excSQL != NULL) {
       System.err.println(excSQL.getMessage());
       excSQL = excSQL.getNextException();
   }
} …
…
```


le cnam

Transactions

- Gestion des transactions avec JDBC :
  - Par défaut : chaque instruction SQL exécutée constitue une transaction (**`commit`** implicite) ; ce mode peut être désactivé avec **`nomConnexion.setAutoCommit(false)`**; quand ce mode est désactivé, l’exécution d’une instruction du LDD ou la fermeture d’une connexion valident implicitement la transaction
  - Explicite : les méthodes **`commit`** et **`rollback`** de **`Connection`** doivent être utilisées

le cnam

Transactions : points de validation

- Objectif : rendre possible l'annulation d'une partie des opérations (à partir de JDBC 3.0)
- Méthodes correspondantes de **`Connection`** :
  - **`Savepoint setSavepoint("NomPoint")`** : insertion point de validation intermédiaire ; si anonyme, **`"NomPoint"`** est absent
  - **`void releaseSavepoint("NomPoint")`** : supprime le point de validation intermédiaire
  - **`void rollback(nomPoint)`** : retour à l'état d'avant **`nomPoint`**
- Méthodes de l'interface **`Savepoint`** :
  - **`int getSavepointId(Savepoint)`** : retourne l'identifiant (entier) du point (pour les points anonymes)
  - **`String getSavepointName(Savepoint)`** : retourne le nom du point (vide si le point est anonyme)

le cnam

Transactions : exemple

```
…
connexion.setAutoCommit(false);
int insCount;
String instrSQL = "INSERT INTO avion VALUES(?, 250,
                  "A400", "Garches")";
PreparedStatement prepStmt =
           connexion.prepareStatement(instrSQL);
prepStmt.setInt(1,210);
insCount = prepStmt.executeUpdate();
prepStmt.setInt(1,211);
insCount = prepStmt.executeUpdate();
connexion.commit();
prepStmt.close();
…
```

le cnam

Procédures et fonctions stockées Java

- Procédure/fonction stockée Java = méthode compilée (*byte-code*), stockée avec la base et exécutée par la JVM
- Étapes :
  1. Programmation de la classe qui contient la méthode visée :

```
public class NomClasse {
…
  public static TypeJava nomMethode(paramètres) {
          …
  }
  …
}
```

  2. Compilation de la classe (vérifier d'abord CLASSPATH)

```
javac NomClasse.java
```

  3. Chargement dans la base de la ressource Java contenant la classe :

```
loadjava –user nom/motdepasse NomClasse.class
```

  (on peut charger aussi des archives `.jar`, des fichiers sources `.java`)

le cnam

Procédures et fonctions stockées Java

- Étapes (suite) :
  4. Publication de la méthode Java comme une procédure ou fonction stockée PL/SQL :

```
CREATE [OR REPLACE]
{ FUNCTION nomFonction (paramètres) RETURN TypeSQL
| PROCEDURE nomProcedure (paramètres) }
{ IS | AS } LANGUAGE JAVA
NAME 'NomClasse.nomMethode(paramètres) [return TypeJava]';
```

  5. Appel de la méthode :
     - À partir de l'interface SQL*Plus :

```
VARIABLE nomVariableGlobale TypeSQL;
SET SERVEROUTPUT ON SIZE 10000
CALL DBMS_JAVA.SET_OUTPUT(10000);
CALL nomFonction(paramètres) INTO :nomVariableGlobale;
CALL nomProcedure(paramètres);
```

le cnam

Procédures et fonctions stockées Java

- À partir de SQL (pour les fonctions) :

```
SELECT … FROM …
      WHERE nomColonne = nomFonction(paramètres);
```

- Comme un déclencheur :

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
{BEFORE | AFTER | INSTEAD OF}
{DELETE | INSERT | UPDATE [OF colonne 1, …] [OR …]}
ON {nomTable | nomVue} [FOR EACH ROW]
BEGIN
      nomProcedure(paramètres);
END;
```

- À partir d'un programme PL/SQL : comme toute fonction ou procédure cataloguée
- À partir d'un programme Java : avec **`CallableStatement`** (comme toute fonction ou procédure cataloguée)

le cnam

Procédures et fonctions stockées Java

- Si un paramètre de la procédure/fonction Java stockée est déclaré **`OUT`** ou **`IN OUT`**, il doit correspondre en Java à un tableau à 1 élément (exemple : **`float[] table;`**) et la valeur transmise est celle d'indice 0 (**`table[0]`**)
- Communication de la procédure/fonction Java stockée avec la base :
  - Par défaut, pilote JDBC « interne » :

```
Connection connexion =
              DriverManager.getConnection
                    ("jdbc:default:connection:");
```

  - D'autres connexions peuvent être établies avec un autre schéma que celui de l'utilisateur appelant la procédure (ou avec une autre base), en utilisant explicitement d'autres pilotes

le cnam

Procédures et fonctions externes Java

- Procédure/fonction externe Java = méthode compilée (*byte-code*), exécutée par la JVM, mais non stockée avec la base
- Étapes :
  1. Programmation de la classe qui contient la méthode visée
  2. Compilation de la classe, le résultat étant placé dans un répertoire **`repertoireClasse`** (en général externe aux répertoires Oracle)
  3. Création d'une librairie :
     ```
     CREATE DIRECTORY repProcExternes AS 'repertoireClasse';
     ```
  4. Chargement de la classe :
     ```
     CREATE JAVA CLASS USING BFILE(repProcExternes,
                                   'NomClasse.class');
     ```
  5. Publication : comme pour une procédure/fonction stockée Java
  6. Appel : comme pour une procédure/fonction stockée Java

le cnam

Insuffisances de JDBC

- API de bas niveau, qui exige une bonne connaissance de SQL et de la base avec laquelle il faut travailler
- Aucun contrôle avant exécution pour
  - La validité syntaxique des instructions SQL transmises
  - La bonne correspondance entre ces instructions et la structure des tables

  ⇒ mise au point difficile des programmes…
- Indépendance relative du SGBDR utilisé

le cnam

SQLJ

- Principe : API au-dessus de JDBC, qui permet l'inclusion directe d'instructions SQL dans le code Java et, grâce à un pré-compilateur, les traduit en appels à des méthodes JDBC
- Le pré-compilateur assure également la vérification de la validité des instructions SQL (par rapport à un SGBDR particulier, chaque SGBDR aura donc son pré-compilateur SQLJ… mais le code SQLJ sera plus portable !)
- Contrainte : les instructions SQL utilisées doivent être connues lors de l'écriture du programme (alors que JDBC permet de les construire dynamiquement) ; Oracle propose une solution propriétaire qui évite cette contrainte

le cnam

Environnement et connexions

- Environnement : **`CLASSPATH`** doit inclure
  - Pré-compilateur : **`Oracle_home/sqlj/lib/translator.jar`** (ou **`.zip`**)
  - JDK : **`Oracle_home/sqlj/lib/classes11.jar`** (ou **`.zip`**), **`Oracle_home/sqlj/lib/classes12.jar`** (ou **`.zip`**)
  - Pilote JDBC : **`Oracle_home/sqlj/lib/runtime11.jar`** (ou **`.zip`**), **`Oracle_home/sqlj/lib/runtime12.jar`** (ou **`.zip`**)
- Connexion : fichier **`connect.properties`**

```
sqlj.url = jdbc:oracle:thin:@odessa:1521:NFA011
sqlj.user = Julien
sqlj.password = monpass
```

le cnam

Introduction de SQL dans Java

- Introduction d'instructions SQL :

```
#sql{CREATE TABLE avion (Numav INTEGER, Capacite
     INTEGER, Type VARCHAR2, Entrepot VARCHAR2)};
#sql{INSERT INTO avion VALUES (14, 25, "A400",
                                "Garches")};
```

- Introduction de blocs PL/SQL :

```
#sql{
    [DECLARE …]
    BEGIN
          …
    [EXCEPTION …]
    END;
};
```


le cnam

Affectation et extraction

- Affectation d'une valeur à une variable Java (traitée dans SQL comme variable hôte) :

```
#sql{SET :variableJava = expression};
```

Exemple :

```
#sql{SET :dateJava = SYSDATE};
```

- Extraction d'un seul enregistrement :

```
#sql{SELECT col1,… INTO :var1Java,…
      FROM listeTables WHERE condition};
```

Exemple :

```
#sql{SELECT Ville_arrivee INTO :villeArrivee
      FROM vol WHERE Numvol = :numVolAller};
```

le cnam

Extraire plusieurs enregistrements

- Avec une instance de **ResultSet** comme variable hôte :

```
ResultSet rset;
#sql{ BEGIN
          OPEN :OUT rset FOR SELECT Ville_arrivee
                             FROM vol;
      END; };
while(rset.next())…
```

- Avec un itérateur SQLJ (exploitable directement en SQL ou en Java à travers son **ResultSet**) :

```
#sql iterator nomTypeIterateur (type1 col1,…);
nomTypeIterateur nomIterateur;
#sql nomIterateur = {SELECT …};
…
nomIterateur.close();
```


le cnam

Appels de sous-programmes

- Appel de fonction stockée :

```
#sql :variableJava =
    {VALUES(nomFonction(parametres))};
```

- Appel de procédure stockée :

```
#sql{CALL nomProcedure(parametres)};
```

- Les paramètres effectifs sont des expressions qui peuvent inclure des variables Java comme variables hôtes :

```
… :IN numVolAller, :OUT villeArrivee, …
```

le cnam

Autres API Java ↔ SGBDR

- JDO (*Java Data Objects*) : possibilité de rendre persistants (de façon transparente) des objets Java ; le support de la persistance est assuré par un SGBDR, un SGBDO, des fichiers ordinaires, etc.
- JavaBlend : correspondance automatique relationnel ↔ objet ; par exemple, classe Java ↔ table et instance de classe ↔ enregistrement
- Serveurs d'application EJB (*Enterprise Java Beans*)

le cnam

Relationnel ↔ Objet

- Modèle relationnel et modèle objet
- Comment intégrer les aspects relationnels et objet ?
- Intégration ou interfaçage ?
- JDO (*Java Data Objects*) et les objets persistants
- *Frameworks* qui emploient la persistance
  - Exemple de Hibernate

→ NSY135 « Applications orientées données - patrons, frameworks, ORM » - au second semestre, enseignée par Philippe Rigaux

le cnam

Modèle relationnel

- Points forts
  - Représentation relationnelle conceptuellement simple et bien adaptée aux applications de gestion
  - Grande maturité du modèle et des SGBDR (optimisation, gestion des transactions)
- Points faibles
  - Pas de support direct (dans le modèle) d'attributs dont les valeurs ont une structure interne
  - Pas de support pour l'encapsulation car séparation entre données et traitements
- Modèle largement dominant dans les bases de données

le cnam

Modèle objet

- Points forts
  - Support d'objets complexes (qui présentent une structure interne)
  - Héritage, permettant la définition de traitements génériques
  - Encapsulation grâce à l'intégration entre données et traitements
- Points faibles
  - Modélisation plus complexe
  - Support plus difficile d'opérations d'optimisation
  - Transition difficile du tout relationnel au tout objet
- Modèle largement dominant dans le développement d'applications

le cnam

Intégrer les aspects relationnels et objet

- Extension des SGDBR et du langage (→ SQL3) pour inclure des aspects du modèle objet
  - + Types définis par l'utilisateur (*Abstract Data Types*) incluant des méthodes (→ données complexes, encapsulation)
    - Références d'objets → types récursifs (par ex. nœuds d'arbres)
  - + Héritage : types définis par l'utilisateur, tables
  - + Préservation de mécanismes performants dans les SGBDR : optimisation, gestion des transactions
  - – Absence de support théorique complet
  - – Différences syntaxiques importantes entre éditeurs de SGBD
  - ◆ Interrogation : prise en compte partielle des aspects objet mentionnés

le cnam

Types définis par l'utilisateur

- Création de type

```
CREATE OR REPLACE TYPE DESCTECHAVION AS OBJECT (
    SERIEMODELE NUMBER(10),
    FICHETECHNIQUE VARCHAR(1024),
    FICHEMAINTENANCE VARCHAR(1024),
    FICHESECURITE VARCHAR(1024))
```

- Affichage d'un type

```
DESC DESCTECHAVION;
```

- Destruction d'un type

```
DROP TYPE DESCTECHAVION;
```

- Des instances persistantes ou non peuvent être créées et utilisées dans un programme PL/SQL

le cnam

Types définis par l'utilisateur (2)

- Table avec attribut un type défini par l'utilisateur

```
CREATE TABLE avions (
    NUMAV NUMBER(4), CAPACITE NUMBER(4),
    TYPE VARCHAR(10), ENTREPOT VARCHAR(10),
    DESCTECH DESCTECHAVION);
```

- Insertion

```
INSERT INTO avions VALUES (25, 250, A320,
'Garches', DESCTECHAVION(3205520012, 'STX34V55',
'GV22FDT45667B', 'EUS0955VAP12'));
```

- Interrogation

```
SELECT NUMAV, a.DESCTECH.SERIEMODELE
FROM avions a;
```


le cnam

Types définis par l'utilisateur (3)

- Table avec lignes de type défini par l'utilisateur

```
CREATE TABLE descriptionstechniques
    OF DESCTECHAVION (
    CONSTRAINT descPK PRIMARY KEY (SERIEMODELE));
```

- Insertion

```
INSERT INTO descriptionstechniques
    VALUES (3205520012, 'STX34V55',
    'GV22FDT45667B', 'EUS0955VAP12');
```

- Interrogation

```
SELECT SERIEMODELE, FICHESECURITE
    FROM descriptionstechniques;
```

le cnam

Types définis par l'utilisateur (4)

- Extraction valeurs des membres d'un objet

```
SELECT VALUE(d) FROM descriptionstechniques d;
```

- Affectation des valeurs des membres d'un objet

```
DECLARE
    description DESCTECHAVION;
BEGIN
    SELECT VALUE(d) INTO description
          FROM descriptionstechniques d
          WHERE SERIEMODELE = 3205520012;
…
```


le cnam

Héritage

- Création de super-classe et classe dérivée

```
CREATE OR REPLACE TYPE DESCTECHAVION AS OBJECT (
    …)
    NOT FINAL
/

CREATE OR REPLACE TYPE DESCTECHAVIONMARCHANDISES
    UNDER DESCTECHAVION (
    FICHECHARGEMENT VARCHAR(1024),
    FICHEDOUANIERE VARCHAR(1024))
/
```

le cnam

Héritage (2)

- Utilisation
  - Une instance de sous-classe peut être affectée à une variable du type de la super-classe
  - Une référence du type de la super-classe peut pointer vers une instance de la sous-classe

```
INSERT INTO descriptionstechniques
    VALUES (DESCTECHAVIONMARCHANDISES
    (3205520012, 'STX34V55', 'GV22FDT45667B',
    'EUS0955VAP12', 'LU3202005680', 'USE428065'));

SELECT TREAT(DESCTECHAVION AS
    DESCTECHAVIONMARCHANDISES) FROM avions;
```


le cnam

Références d'objets

```
CREATE TABLE avionsCpct (
    NUMAV NUMBER(4), CAPACITE NUMBER(4),
    TYPE VARCHAR(10), ENTREPOT VARCHAR(10),
    REFDESCTECH REF DESCTECHAVION);
```

- Insertion

```
INSERT INTO avionsCpct VALUES (25, 250, A320,'Garches',
    (SELECT REF(d) FROM descriptionstechniques d
    WHERE d.SERIEMODELE = 3205520012));
```

- Interrogation

```
SELECT a.NUMAV FROM avionsCpct a
    WHERE a.REFDESCTECH.SERIEMODELE = 3205520012;
SELECT a.REFDESCTECH.SERIEMODELE FROM avionsCpct a
    WHERE a.NUMAV = 25;
SELECT DEREF(REFDESCTECH) FROM avionsCpct
    WHERE NUMAV = 25;
```

le cnam

Méthodes avec PL/SQL

- Spécification

```
CREATE OR REPLACE TYPE DESCTECHAVION AS OBJECT (
    SERIEMODELE NUMBER(10), …
    MEMBER PROCEDURE UpdateFicheMaintenance (…),
    STATIC FUNCTION CheckAllDesc RETURN BOOLEAN,
    CONSTRUCTOR FUNCTION DESCTECHAVION (
          …) RETURN SELF AS RESULT)
```

- Implémentation

```
CREATE TYPE BODY DESCTECHAVION AS
    MEMBER PROCEDURE … IS BEGIN … END;
    STATIC FUNCTION … IS BEGIN … END;
    CONSTRUCTOR FUNCTION DESCTECHAVION (…) RETURN
          SELF AS RESULT IS BEGIN … END;
END;
```


le cnam

Méthodes avec PL/SQL (2)

- Appel depuis PL/SQL

```
DECLARE
    …
    ficheMaj DESCTECHAVION;
    ficheCourante DESCTECHAVION;
BEGIN
    …
    IF CheckAllDesc() THEN
          ficheMaj.UpdateFicheMaintenance(…);
          ficheCourante := NEW DESCTECHAVION(…);
    END IF;
    …
END;
```

le cnam

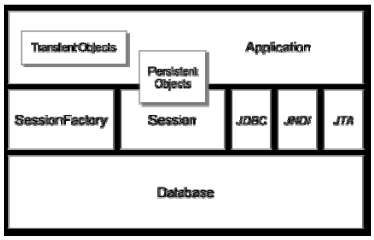
Hibernate

- *Framework* : bibliothèques, guide architectural et conventions pour faciliter le développement d'applications
- Persistance des données : conservation des données entre différentes exécution d'un programme
- *Hibernate* : *framework open source* pour le *mapping* objet-relationnel (*Object Relational Mapping*, ORM)
  - Définition déclarative de correspondances (*mapping*) entre des objets et une base de données relationnelle
  - Les échanges entre l'application et la base sont assurés grâce à la persistance des objets, précisée par le *mapping* → les programmeurs peuvent éviter le détail des échanges
  - Environnement Web léger (ex. *Apache Tomcat*) ou environnement J2EE complet (ex. *JBoss Application Server*)

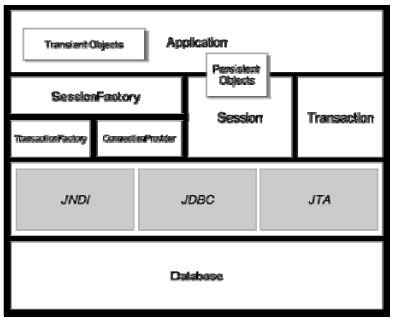
le cnam

Hibernate : architecture

Architecture légère



Architecture complète



(schémas issus de la doc. de référence Hibernate 2.1.8)

le cnam

Hibernate : exemple configuration

- Fichier de configuration XML (**`hibernate.cfg.xml`**)

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
      PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
      "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
   <hibernate-configuration>
      <session-factory> <!-- 1 SessionFactory = 1 base de données -->
         <property
            name="connection.datasource">java:comp/env/jdbc/quickstart
         </property> <!-- adresse JNDI pour le pool de connexions -->
         <property name="show_sql">false</property> <!-- sans log SQL -->
         <property
            name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect
         </property>
         <!-- fichiers de mapping -->
         <mapping resource="Cat.hbm.xml"/> <!-- mapping classe Cat -->
      </session-factory>
   </hibernate-configuration>
```

(exemple issu de la documentation de référence Hibernate 2.1.8)

le cnam

Hibernate : exemple classe persistante

```
package net.sf.hibernate.examples.quickstart;
     // classe de type Plain Old Java Objects (POJO)
public class Cat {
     private String id;
     private String name;

     public Cat() {
     }
     public String getId() {
         return id;
     }
     private void setId(String id) {
         this.id = id;
     }
     public String getName() {
         return name;
     }
     public void setName(String name) {
         this.name = name;
     }
}
```

le cnam

Hibernate : exemple fichier *mapping*

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping> <!-- fichier Cat.hbm.xml  -->

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- clé 32 caractères hexadécimaux, générée par Hibernate -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- nom qui ne doit pas être trop long -->
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

    </class>

</hibernate-mapping>
```


le cnam

Hibernate : exemple d'utilisation

- A partir des *mapping*, le schéma de la base peut être généré automatiquement (LDD SQL) par l'outil *SchemaExport* d'Hibernate
- Création objet persistant (note : **`HibernateUtil`** est ici une classe utilisateur)

```
SessionFactory sessionFactory =
     new Configuration().configure().buildSessionFactory();

Session session = HibernateUtil.currentSession();

Transaction tx = session.beginTransaction();

Cat princesse = new Cat();
princesse.setName("Princesse");

session.save(princesse);
tx.commit();

HibernateUtil.closeSession();
```

le cnam

Hibernate : exemple d'utilisation (2)

- Récupération depuis la base d'objets persistants, avec HQL

```
Transaction tx = session.beginTransaction();
nChats = 0;
catName = 'Princesse';
Query query = session.createQuery("select c from Cat as c
                                   where c.name = :nom");
query.setCharacter("nom", chatNom);
for (Iterator it = query.iterate(); it.hasNext(); nChats++) {
     Cat chat = (Cat) it.next();
     …  // autres opérations
}
out.println(nChats + " chats de nom " + chatNom);
tx.commit();
…
```

→ NSY135 « Applications orientées données - patrons, frameworks, ORM » - au second semestre, enseignée par Philippe Rigaux