



**Bases de données relationnelles :
des concepts au développement d'applications avec Oracle**

Textes des Travaux pratiques

1. Objectifs et organisation du TP	2
2. Mise en oeuvre	2
2.1. Installation préalable	2
2.2. Installations propres au TP	2
2.3. Programmes exemples	2
3. Java et SGBDR : l'impedance mismatch	5
3.1. Présentation de l'application cible	5
3.2. Différences des moteurs d'exécution	5
3.3. Mapping objet relationnel	6

Merci de communiquer toute correction ou remarque sur ce document à :

Philippe.Picouet@enst-bretagne.fr ou Philippe.Tanguy@enst-bretagne.fr

1. Objectifs et organisation du TP

Cette séance est consacrée à la mise en évidence du problème d'impedance mismatch au sein de programmes Java accédant à la base via JDBC.

Des exemples (section 2) vous sont tout d'abord fournis pour illustrer l'utilisation de l'API JDBC.

Le travail à réaliser est décrit en section 3. Vous devez implanter une fonctionnalité de calcul de commission sur un schéma très simple (EMP-DEPT). Cet exercice doit permettre de constater les problèmes posés par la coexistence de modèles d'exécution et de modèles de données différents en Java et en SQL (impedance mismatch).

2. Mise en oeuvre

2.1. Installation préalable

SDK (*Java Development Kit*) est un ensemble d'outils permettant, entre autres, la compilation (commande *javac*) et l'exécution des programme Java (commande *java*).

- Vérifiez qu'une version suffisamment récente de java (1.3 minimum) est installée sur votre machine (taper dans une invite de commande : `java -version`)
- Vérifiez que la variable *PATH* comprend le répertoire bin de l'installation java

2.2. Installations propres au TP

L'ensemble des fichiers nécessaires à la réalisation de ces exercices peut être téléchargé depuis http://perso/~picouet/french/skbd/jdbc/TP_jdbc.zip

Le répertoire ainsi créé contient un driver jdbc pour Oracle, ainsi que les sources et les corrigés des exercices des parties 3 et 4

Un driver JDBC est constitué d'un ensemble de classes Java réunies dans un archive ZIP. Lors de l'exécution d'un programme qui se connecte à une base de données, l'accès par celui-ci des classes du driver JDBC doit être possible. Pour cela, une variable d'environnement – nommée *CLASSPATH* – peut être positionnée une fois pour toute dans le système pour en indiquer le chemin d'accès.

Un fichier classes12.zip est déjà stocké dans l'archive présentée ci-dessus

Vérifiez que la variable d'environnement *CLASSPATH* est bien positionnée ou créez la :

CLASSPATH → `.;%CLASSPATH%;D:\TP_JDBC\classes12.zip`

2.3. Programmes exemples

Les différents aspects abordés dans ce TP concernent :

- la connexion/déconnexion à une base de données ;
- l'utilisation des différents types de *Statements* chargés d'exécuter des requêtes sur la base ;
- le fonctionnement d'un *ResultSet* pour la récupération et l'analyse des résultats ;

- la construction d'une requête dynamique qui n'est pas connue avant l'exécution (avant la compilation) ;
- la gestion des transactions.

Le principe général du fonctionnement de chacun des programmes à développer est le suivant :

- 1 – lancement du programme
- 2 – connexion à la base (début de la transaction)
- 3 – action(s) sur la base (requêtes, mises à jour, insertions)
- 4 – déconnexion de la base (fin de la transaction)
- 5 – arrêt du programme

2.3.1. Requêtes statiques

2.3.1.1. Statement simple

Le programme permet d'effectuer une requête affichant une partie du contenu de la table stockant les employés (*select ename,job from EMP*), afficher le résultat sur la sortie standard puis fermer la connexion à la base. L'envoi de la requête sera réalisé grâce à un *Statement* simple qui récupérera le *ResultSet* contenant le résultat. La manipulation de ce *ResultSet* permettra d'en extraire les valeurs des colonnes et de les afficher. Les deux manières de récupérer les valeurs (positionnement et nom de la colonne) seront testées.

- 1 - Appel de la méthode *connexion()*.
- 2 - Appel de la méthode *executionStatementSimple()* qui se charge de l'envoi à la base de données de la requête et réalise l'affichage du résultat.
- 3 - Appel de la méthode *deConnexion()* puis fin du programme.

2.3.1.2. PreparedStatement

L'exercice vise à afficher la liste des personnes suivant leur département. Pour cela, deux étapes sont nécessaires. La première récupérera la liste des ID des départements à l'aide d'un *Statement* simple et la seconde récupérera la liste des personnes suivant ces ID. Cette seconde étape est une succession d'envoi de plusieurs requêtes quasiment identiques à l'exception près du numéro du département. C'est pourquoi, dans un souci d'optimisation, l'emploi d'un *PreparedStatement* est indiqué dans ce cas.

- 1 - Appel de la méthode *connexion()*.
- 2 - Appel de la méthode *executionPreparedStatement()* qui se charge de l'envoi à la base de données des deux requêtes nécessaires et réalise l'affichage du résultat.
- 3 - Appel de la méthode *deConnexion()* puis fin du programme.

2.3.1.3. CallableStatement

Dans cet exemple, on insère un nouvel employé dans la base en appelant à partir du code Java une procédure stockée.

La première étape consiste à insérer dans la base la procédure stockée. Celle-ci est présente dans le fichier *insertEMP.sql* et son installation se déroule de la manière suivante :

- 1 - Ouverture d'une fenêtre « invite de commande MS-DOS » ;

- 2 - Lancement de la commande : `sqlplus fcbdXX/fcbd@ENSEIG`
- 3 - Dans l'interpréteur SQL, lancement de la commande : `@insertEMP`. Si tout se passe bien, le message suivant apparaît : *Procédure créé.*

L'embryon de la classe est fourni : `TP33.java`. Fonctionnement :

- 1 - Appel de la méthode `connexion()`.
- 2 - Appel de la méthode `executionCallableStatement ()` qui se charge de l'envoi à la base de données des paramètres nécessaires à l'exécution sur la base de la procédure puis réalise l'affichage de la table pour visualiser le résultat de l'insertion.
- 3 - Appel de la méthode `deConnexion()` puis fin du programme.

2.3.2. Requêtes dynamiques

2.3.2.1. Sans interrogation du schéma

Cet exemple crée une méthode qui visualisera le contenu d'une table inconnue avant l'exécution du programme. Le nombre de colonnes renvoyées dans le `ResultSet` étant inconnu, il faut alors interroger les métadonnées associées pour gérer correctement l'affichage.

- 1 - Appel de la méthode `connexion()`.
- 2 - Appel de la méthode `visualisationContenuTable("nomDeLaTable")` qui se charge de l'envoi à la base de données des paramètres nécessaires à l'exécution sur la base de la procédure puis réalise l'affichage de la table pour visualiser le résultat de l'insertion.
- 3 - Appel de la méthode `deConnexion()` puis fin du programme.

2.3.2.2. Interrogation du schéma

Cet exemple interroge le dictionnaire de données (classe `DatabaseMetaData`)

L'embryon de la classe est fourni : `TP36.java`. Fonctionnement :

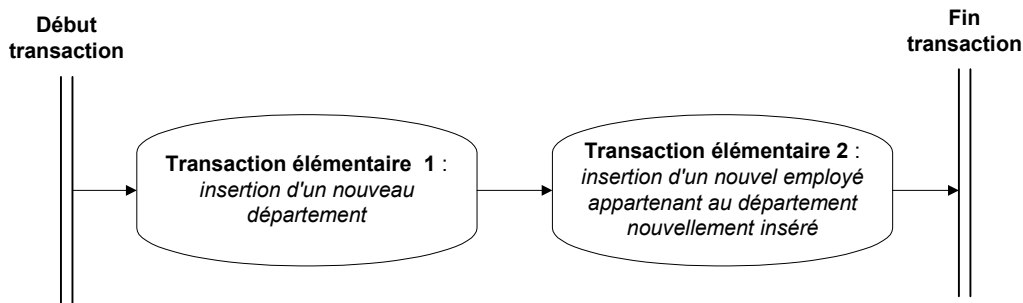
- 1 - Appel de la méthode `connexion()`.
- 2 - Appel de la méthode `exploreMetaData()` qui récupérera les renseignements :
 - liste de toutes les tables du schéma
 - liste des colonnes de la table EMP
 - liste des clés primaires de la table EMP
 - liste des clés étrangères de la table EMP
- 3 - Appel de la méthode `deConnexion()` puis fin du programme.

2.3.3. Modification et transactions

Jusqu'à présent, chacune des transactions sur la base était automatiquement validée (`commit`). Le but de cet exemple vise à gérer les transactions au sein même du programme Java (`rollback` suite à une erreur de saisie par exemple...). Ceci est rendu possible grâce à la méthode `setAutoCommit(true ou false)` qui s'applique à un instance de `Connection`.

Le principe de l'exercice est de cascader un ensemble de transactions élémentaires que l'on désire considérer comme une transaction unique que l'on ne validera que si et seulement si

l'ensemble des transactions s'est correctement déroulé (c'est à dire qu'aucune erreur dans le traitement n'a été rencontré). Dans le cas contraire, tout doit être annulé.



L'embryon de la classe est le suivant (*TP35.java*) :

- 1 - Appel de la méthode *connexion()*.
- 2 - Appel de la méthode *executionInsertionNouveauDepartement()* : insertion d'un nouveau département nommé *FC* localisé à *RENNES* de n° 50. à l'aide d'un *Statement* simple comme en 3.1
- 3 - Appel de la méthode *executionInsertionNouvellePersonne()* : cette personne appartient au département nouvellement crée. Pour l'insertion, on utilisera la procédure stockée utilisée en 3.3 grâce à un *CallableStatement*.
- 4 - Appel de la méthode *deConnexion()* puis fin du programme.

Dans un premier temps, on réalise deux insertions correctes qui doivent être validées. Dans un second temps, on simule une erreur dans l'insertion de la personne et le programme doit annuler les **deux** insertions. On peut visualiser (*SQL+* ou *Cast SQL Builder*) l'annulation ou la validation de la transaction globale.

3. Java et SGBDR : l'impedance mismatch

3.1. Présentation de l'application cible

L'application à développer consiste à distribuer une commission entre les employés de la base. La valeur de la commission est globale et doit être partagée entre tous les employés de la base à proportion de son salaire : un employé dont le salaire représente 15% de la masse salariale totale se verra attribué 15% du montant total de la commission.

La valeur de la commission totale est renseignée lors du lancement du programme :

java Commission 10000. Le programme effectue le traitement et la mise à jour puis affiche en fin de processus le contenu de la table EMP pour contrôle.

3.2. Différences des moteurs d'exécution

Un embryon de la classe *Commission* est fourni : *Commission.java*. Dans celui-ci, seul le traitement concernant la récupération, le calcul de la commission puis la mise à jour sur la BD est à réaliser dans la méthode *affectationCommission*.

Le fonctionnement proposé est le suivant :

- Calcul de la masse salariale (somme des salaires de tous les employés)

- Pour chaque employé :
 - Récupération du n° d'employé et de son salaire
 - Calcul du montant individuel de la commission
 - Mise à jour dans la table du nouveau montant de commission
- Affichage du contenu de la table pour vérification

Critiquez la solution ci-dessus du point de vue des performances et proposez une autre solution qui minimise le nombre de requêtes à la base de données, en particulier la deuxième phase. Justifiez votre réponse.

3.3. Mapping objet relationnel

Les sections précédentes, bien qu'illustrant les moyens d'accéder à une base relationnelle via JDBC, n'utilisent pas la puissance de la modélisation objet telle qu'elle est possible en java. En particulier, nous n'avons pas mis en œuvre, dans ces exemples, de classe employé, ou département. Nous allons nous y intéresser dans cette section.

Il s'agit maintenant de réécrire le programme précédent en supposant que l'on n'accède non pas directement à la relation EMP mais qu'il existe une classe EMP dont les instances correspondent effectivement à celles de la relation EMP.

Créer en java la classe Employé, composée uniquement de son numéro, nom, salaire et commission ainsi que les méthodes associées, et réécrire le programme précédent en passant obligatoirement par la classe Employé.

Dans un premier temps, vous pourrez procéder en 3 temps :

- Création de toutes les instances de la classe Employé par lecture de EMP
- Exécution de la mise à jour
- Recopie des instances de la classe dans la relation EMP.

Cette solution présente toutefois l'inconvénient de devoir décharger dans la mémoire de l'ordinateur toutes les données de la base, au risque de faire exploser la mémoire. On lui préfère donc la solution qui consiste à ne considérer la classe Employé que comme une interface à la relation EMP. Pour ce faire, on implante pour chaque attribut de la classe Employé les méthodes get et set qui vont lire et écrire dans la base de données sans avoir besoin de stocker les valeurs en mémoire.