TD JAVA n°10: Persistance des Objets avec JDBC

Remarques : *Dans le cadre de ce TD, pour ne pas alourdir le code, on ne vous demande pas de logguer les méthodes, vous devrez juste propager des* `CabinetTechniqueException`*. Par contre, les logs devront être implémentés en fin de séance de TP. Vous ne devrez pas non plus écrire la classe* `CabinetTechniqueException`*, puisqu'elle vous a déjà été fournie en TP. Rappel : Cette classe permet de lever des exceptions dites de " technique". Elle est écrite sur le même principe que* `CabinetMedicalException` *(qui permet de lever des exceptions dites de " métier").*

Exercice 1 : Classe `SimpleConnection`

La classe **`SimpleConnection`** va vous permettre d'établir une connexion à la base de données en optimisant le chargement du driver (driver chargé 1 seule fois).

1. Dans un premier temps, vous n'allez coder **que les instructions relatives au chargement du Driver**
Vous utiliserez un driver de type I (pont jdbc/odbc) déjà fourni dans le JDK :
`sun.jdbc.odbc.JdbcOdbcDriver`
(voir transparent n°17 du cours Persitance des Objets avec JDBC)

En appliquant le pattern Singleton (voir transparent n°13 du cours Design Pattern) à la classe **`SimpleConnection`**, écrire le code de cette classe de manière à garantir qu'il ne puisse y avoir qu'une seule instance de `SimpleConnection` dans notre application.
C'est bien sûr lors de la création de cette instance que le chargement du driver devra être réalisé….afin de garantir que le chargement n'ait lieu qu'une seule fois…
Si une `Exception` se déclenche vous la transformerez en `CabinetTechniqueException` et vous la relancerez avec un message approprié…

2. Pour obtenir une connexion à la base de données **`cabinetMedical`** (via JDBC bien sûr), deux étapes sont nécessaires :
→ a. Le chargement du driver (c-a-d le chargement de classes implémentant l'interface `Connection` pour le driver souhaité) : ce sont les instructions que vous venez de coder à la question précédente
→ b. L'instanciation d'un objet de type **`java.sql.Connection`** réalisé par le `DriverManager`.
C'est ce que vous allez coder dans la méthode suivante qui viendra compléter la classe `SimpleConnection`

```
public Connection getConnection() throws CabinetTechniqueException
```

Au niveau de la gestion des transactions, on décide que la **`Connection`** renvoyée est définie dans la méthode `getConnection` en **désactivant le mode `AutoCommit`** c-a-d qu'on passe en mode transactionnel.
Comme précédemment et comme indiqué dans l'entête de la méthode, si une `Exception` se déclenche vous la relancerez avec un message approprié sous forme de **`CabinetTechniqueException`**

3. Le choix de conception concernant la gestion des connexions pour notre application `CabinetMedical` sera le suivant :
   a. chaque méthode du Contrôleur devra ouvrir une nouvelle connexion.
   b. La connexion sera ensuite passée en paramètre à chacune de(s) méthode(s) du DAO appelée(s) au sein de cette méthode.
   c. La transaction devra être *validée* (`commit`) en fin de méthode (une fois toutes les requêtes réalisées) c-a-d que c'est le Contrôleur qui "`commit`"…
   …ou s'il y a eu un problème durant cette transaction, le Contrôleur doit *annuler* cette transaction par un appel à `rollback` (toutes les modifications effectuées dans la base durant la transaction en cours seront alors annulées)
   d. La connexion sera ensuite fermée en fin de méthode du contrôleur, afin de respecter le principe suivant : *chaque fermeture d'une connexion est à la charge de « celui » qui l'a demandée (ouverte)…*

A partir de la classe `SimpleConnection` écrite précédemment, quelle instruction devrez-vous écrire dans chaque méthode du contrôleur pour réaliser le point a. (ouverture d'une connexion) ?

Exercice 2: *Mise en place de requête sur la base : méthode* `findAllPersonnes` *de la classe* `PersonneDAOJDBC`

L'entité principale du modèle objet est la «classe». Celle-ci sera transformée dans le SGBDR en une «table», à laquelle on pourra donner le même nom que la classe. De la même manière qu'une classe est composée de plusieurs «attributs», une table est composée de «champs», tous typés. Cela n'est rien d'autre que du **Mapping Objet/Relationnel** : toutes les entités du modèle relationnel ont leur équivalent dans le modèle objet, et inversement.

La base de données **`cabinetMedical`** (créée sous Access) sur laquelle nous travaillerons comportera deux tables :
- une table `Personne` qui regroupera à la fois les `Patient` et les `Professionnel`
- une table `Adresse` qui fera référence à la `Personne` à qui appartient cette adresse.

↳ Mapping de la table `Adresse` sous Access

Adresse

idAdresse	numero	rue	voie	batiment	codePostal	ville	pays	idPersonne
1	15	avenue Jean Jaure			87000	Limoges	France	1
18	3	rue de Limoges			87170	Isle	France	2
19	10	rue de Toulouse		Batiment A	87000	Limoges	France	3
20	123Bis	Boulevard d'Ici			87000	Limoges	France	4

Jusqu'à présent dans notre projet (modèle objet en JAVA) la classe `Adresse` possédait les attributs suivants :

```
private String numero;      private String rue;          private String voie;
private String batiment;    private String codePostal;   private String ville;
private String pays;
```

Dans la base Access, on retrouve ces caractéristiques dans des champs (possédant le même nom que les attributs Java). On notera deux champs supplémentaires :
`idAdresse` (clé primaire auto-incrémentée) et `idPersonne` (clé étrangère)
Le choix de la sérialisation du projet dans une base de données relationnelle, nous amènera en TP à modifier dans le programme Java la classe `Adresse` en lui rajoutant un nouvel attribut `idAdresse` et ses getteurs/setteurs correspondants. Bien sûr, on ne rajoutera pas le champ `idPersonne` dans le code Java la classe `Adresse`!!!

↳ Mapping de la table `Personne` sous Access

Intéressons nous maintenant à un concept propre au modèle objet : l'héritage.

Il existe plusieurs manières de "mapper" l'héritage dans une base de données relationnelle.
Notre choix de conception est le suivant : nous modéliserons toute une hiérarchie de classes dans une même table, chaque classe ajoutant ses attributs propres comme de nouveaux champs.
La table `Personne` sous Access devra donc contenir comme champs : tous les attributs de la classe `Personne` et tous les attributs propres à la classe `Patient` et tous les attributs propres à la classe `Professionnel` et un nouveau champ `idTypePersonne` qui permettra de différencier par une valeur numérique les enregistrements de type `Patient` des enregistrement de type `Profesionnel`
→Si on enregistre un `Patient`, les champs correspondants à un `Professionnel` seront à `null` (comme dans le cas de l'enregistrement n°1 correspondant à DUPONT Julie)
→Si on enregistre un `Professionnel`, les champs correspondants à un `Patient` seront à `null` (comme dans le cas de l'enregistrement n°2 correspondant à LEDOC Paul)

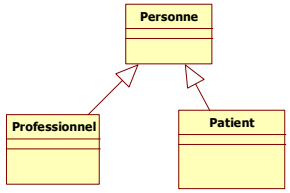
Personne

idPersonne	nom	prenom	datenaissan	male	telephone	portable	email	idAscendan	idTypePersc	nir	medecinTra	immatricula	specialite
1	DUPONT	Julie	21/05/1960	☐	0555434355	0606060606	julie.dupont@		0	2600587001123	LEDOC Paul		
2	LEDOC	Paul	10/07/1976	☑	0555434343	0612345678	paul.ledoc@le		1			871255358	généraliste
3	CHILDREN	Rose	16/02/1970	☐	0555434343	0678654321	rose.children@		1			312444555	pédiatrie
4	DURAND	Alfred	23/05/1968	☑	0512348989	0605050505	alfred.durant@		0	168072B123456	LEDOC Paul		

Afin de simplifier l'exercice, pour commencer, on travaillera à, partir de la base précédente, c-a-d une base dans laquelle, on a choisi de ne pas traiter le cas de l'Ascendant (enregistrements réalisés uniquement avec le flot de base du Use Case CréerPatient)

Pour la suite de l'exercice, on considèrera donc que les `Personnes` sont :
- soit des `Patients` (sans Ascendant),
- soit des `Professionnels`,

ce qui donne le diagramme de classes ci-contre
(c-a-d qu'on perd temporairement la réflexive sur `Personne`)



La différence dans la base de données entre un enregistrement de type `Patient` et un enregistrement de type `Professionnel` se fera en examinant la valeur du champ `idTypePersonne`.
Nous avons adopté la convention suivante :
- lorsque le champ `idTypePersonne` a pour valeur `0`, l'objet correspondant (à cet enregistrement) dans le programme Java est une instance de la classe `Patient`
- lorsque le champ `idTypePersonne` a pour valeur `1`, l'objet correspondant (à cet enregistrement) dans le programme Java est une instance de la classe `Professionnel`

Le but de cet exercice est de coder la ***méthode* `findAllPersonnes` *de la classe* `PersonneDAOJDBC` :**

```
public static Collection<Personne> findAllPersonnes(Connection c)
                                   throws CabinetTechniqueException
```

→ Cette méthode nous permettra de récupérer la liste de toutes les personnes (sans disctinction) mémorisées dans la table `Personne` de la base `CabinetMedical` et retournera cette liste sous forme d'une `Collection.`
→ Cette méthode sera pour l'instant une méthode statique.
→ Seule une `CabinetTechniqueException` pourra être lancée par les méthodes de la classe **`PersonneDAOJDBC.`**
→ Conformément au point 3.b de l'exercice précédent, notre choix de conception nous impose que la connexion soit passée en paramètre à chacune de(s) méthode(s) du DAO.

L'implémentation de la méthode **`findAllPersonnes`** étant un peu complexe de prime abord, nous allons décomposer le problème en 2 étapes afin d'arriver à son implémentation définitive.
→ Etape 1 : **Récupération du `Résultet`**
→ Etape 2 : **Création de la collection de `Personne` à partir du `ResultSet`**

Pour coder cette méthode, compléter le code ci-après en suivant les pas à pas les questions 2.1 et 2.2 …
…. et en vous aidant des transparents de la partie IV du cours n°10 : Persistance des Objets avec JDBC


```
// Ne pas oublier les import …

public class PersonneDAOJDBC {

public static Collection<Personne> findAllPersonnes(Connection c)
                                                    throws CabinetTechniqueException
    {
        Collection<Personne> listPers =new ArrayList<Personne>();

        ////////////////////////////////////////
        /// Etape 1 : Recuperation du ResulSet
        /// à l'aide d'une requête précompilée
        ////////////////////////////////////////
        //… à vous de coder : cf question 2.1









        ////////////////////////////////////////////////////////
        /// Etape  2 : Création de la collection de Personne
        /// à partir du RésultSet
        ////////////////////////////////////////////////////////

        //… le code vous sera donné à la question 2.2



        ////////////////////////////////////
        /// Fermeture des ressources
        /// ouvertes dans la méthode
        ////////////////////////////////////
        // dans un bloc finally => cf cours Exceptions ...
        //… à vous de coder
        finally
        {




        }


        ////////////////////////////////
        /// renvoi de la collection
        ////////////////////////////////
        return listPers;
    }
}// fin classe PersonneDAOJDBC
```

2.1 Etape 1 : Récupération du contenu de la table `Personne` dans un objet de type `Resultet`

Dans un premier temps, implémenter l'étape 1 de la méthode `findAllPersonnes` c-a-d écrire les instructions qui permettent de récupérer dans un objet de type `ResultSet` (que vous appelerez `rs`) tous les enregistrements de la table `Personne`.

→ Pour coder vos requêtes, prenez l'habitude d'utiliser un **`PreparedStatement`**

2.2 Etape 2: Parcours du `ResultSet` pour la création de la collection de `Personne`.

Une fois, l'objet de type `ResultSet` obtenu, il est possible de parcourir enregistrement par enregistrement cet objet afin de construire pas à pas la collection de `Personne(s)` à l'aide des instructions suivantes.

```
        ////////////////////////////////////////////////////
        /// Etape  2 : Création de la collection de Personnes
        /// à partir du RésultSet
        ////////////////////////////////////////////////////
        try {
            while (rs.next()){
                        Personne unePersonne  = getBonnePersonne(rs,c);
                        listPers.add(unePersonne);
                    }
        }
        catch (SQLException e) {
            throw new CabinetTechniqueException("Erreur parcours du resultSet de
findAllPersonnes"+e.getMessage());
        }
```

Pour cette question, on vous demande donc d'écrire la méthode statique **`getBonnePersonne`** qui vous permettra à partir des données de `rs` de reconstruire soit un objet de type `Patient`, soit un objet de type `Professionnel` :

```
private static Personne getBonnePersonne(ResultSet rs,Connection c)
                                            throws CabinetTechniqueException
```

Remarque :
Pour récupérer l'adresse de la `Personne`, vous utiliserez directement la méthode statique suivante :

```
public static Adresse findAdresseByIdPersonne(Integer idPersonne, Connection c)
                                            throws CabinetTechniqueException
```

de la classe `AdresseDAOJDBC` qui vous sera donnée en TP …

Exercice 1 : Classe `SimpleConnection`

La classe **SimpleConnection** va nous permettre d'établir une connection à la base de données en optimisant le chargement du driver .

1. Pattern singleton pour garantir un chargement unique du driver : jdbc-odbc

Rappel de la forme du singleton…
Le singleton est une **classe** qui permet de **s'assurer qu'il n'existera qu'une et une seule instance d'un objet de cette classe dans l'espace et dans le temps** d'un bout à l'autre du cycle de vie de l'application

→ **Le problème.** Certaines classes ne doivent pas avoir plus d'une instance lors de l'exécution du programme auquel elles appartiennent. Cela se justifie soit par la nature de la classe (elle modélise un objet unique, comme un ensemble de variables globales à l'application), soit par souci d'économie de ***ressource mémoire*** (une instance unique fournit le même niveau de service que de multiples instances).

→ **La solution du problème.** La classe doit comporter :
- un **attribut statique**, généralement appelé **instance**, destiné à recevoir la référence de l'instance unique pour l'ensemble du logiciel,
- et une **méthode**, généralement appelée **`getInstance`**, renvoyant la valeur d'instance. Si instance est vide, `getInstance` crée une nouvelle instance de la classe en la stockant dans l'attribut instance et la renvoie à l'appelant.
- bloquer le **constructeur** en le mettant en **private**. Sinon ce n'est pas vraiment un singleton puisqu'il pourra être instancié comme n'importe quel autre objet.

Remarque :
D'une manière générale lorsque l'on veut partager une ressource ou une variable entre plusieurs instances de classes et être sûre que la ressource ne sera initialisée qu'une fois (c'est souvent le cas avec les connexions aux bases de données) il faut utiliser le pattern singleton. Ce pattern garanti la création unique d'une classe et son partage parmi les autres.

```
// Modèle du singleton du cours
…auquel ne rajoutera aucun attribut …
public class SimpleConnection {
   private static SimpleConnection instance = null;

   private SimpleConnection () { // constructeur privé }

   public static synchronized SimpleConnection getInstance() {
      if (instance == null)
         { instance = new SimpleConnection () ;}
      return instance ;
      }

   }
```

simpleConnection

Remarque : par convention, on remplace « `instance` » par une « **`simpleConnection`** »…
Quand on écrit une classe Singleton, l'instance porte le nom de la classe…

⇒ La **chargement du driver** va se faire **une fois** lors de l'instanciation …

Etape1 : chargement du driver (transparent 11)

```
try{
    Class.forName(driver);
    }
catch(ClassNotFoundException cnfe){
   System.out.println("Driver introuvable : ");
   cnfe.printStackTrace();  }
```

que nous transformerons en :

```
try{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch (ClassNotFoundException e) {
        throw new CabinetTechniqueException("Erreur Driver : Classe
Introuvable"+e.getMessage());
    }
```

Ce code peut être écrit :
- soit dans la méthode `getInstance` …
- soit dans le `constructeur` …

Version n°1 : Chargement du driver dans le constructeur

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.iut.cabinet.metier.CabinetTechniqueException;

public class SimpleConnection {

    // attribut statique destiné à recevoir la référence de l'instance unique
    private static SimpleConnection simpleConnection = null;


    //Constructeur privé
    private SimpleConnection() throws CabinetTechniqueException {
        // chargement du driver
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            }
        catch (ClassNotFoundException e) {
            throw new CabinetTechniqueException("Erreur Driver : Classe Introuvable"+
                                                e.getMessage());
    }

    }

    // renvoie une instanciation unique d' un objet  static Singleton
    //en  vérifiant si cette instance existe déjà ou si elle doit vraiment être créé
    // méthode renvoyant la valeur d'instance : point accès unique et global
    public static synchronized SimpleConnection getInstance()
                                            throws CabinetTechniqueException
    {   if (simpleConnection == null)
        {simpleConnection = new SimpleConnection();}
    return simpleConnection; // retourne référence de l 'instance créée ou existante
    }// fin getInstance

}// fin classe SimpleConnection
```

↳ Version n°2 : Chargement du driver dans la méthode `getInstance`

```
import java.sql.DriverManager;
import java.sql.SQLException;
import com.iut.cabinet.metier.CabinetTechniqueException;

public class SimpleConnection {
    // attribut statique destiné à recevoir la référence de l'instance unique
    private static SimpleConnection simpleConnection = null;


    //Constructeur privé
    private SimpleConnection(){
    }

    // renvoie une instanciation unique d' un objet  static Singleton
    //en  vérifiant si cette instance existe déjà ou si elle doit vraiment être créé
    // méthode renvoyant la valeur d'instance : point accès unique et global
    public static synchronized SimpleConnection getInstance()
                                                    throws CabinetTechniqueException
    {   if (simpleConnection == null)
        {
        simpleConnection = new SimpleConnection();

        // chargement du driver
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            }
        catch (ClassNotFoundException e) {
            throw new CabinetTechniqueException ("Erreur Driver :
                                        Classe Introuvable"+e.getMessage());
            }
        }//fin if
        return simpleConnection; // retourne référence de l 'instance créée ou existante
    } // fin getInstance
}// fin classe SimpleConnection
```


Remarques :

↳ Dans la "vraie vie", plusieurs requêtes peuvent être réalisées au sein d'une même méthode du Contrôleur (c'est la cas lorsqu'on on effectue des transcations bancaires : un compte est crédité, l'autre est débité….)
on souhaite donc que la ***transaction finale*** soit "validée" une fois toutes les requêtes ont été correctement exécutées (pas de débit sans crédit et vice-versa!!!) : on choisit donc de passer la transaction **en `NON Autocommit` (`setAutocommit(false)`)**et de valider la transaction dans chaque méthode du Ctrl (`co.commit()`;…

↳ La méthode `getConnection` ne doit pas être `static`,
Le singleton est une classe qui permet de s'assurer qu'il n'existera qu'une et une seule instance d'un objet de cette classe dans l'espace et dans le temps d'un bout à l'autre du cycle de vie de l'application
Rappelons que chaque méthode du Contrôleur va ouvrir une connection et la fermer après utilisation (appel à un ou plusieurs DAO) : ***la fermeture de la connexion est à la charge de celui qui la demande …***
Donc si on ferme la connexion avec un appel à la méthode close que se passerait-il ensuite ?
Comme la classe serait un singleton, on ne recréera pas d'instance de la connection !!! … donc on serait bloqué si on perdait la connexion… Ce n'est donc pas une bonne idée de passer par un singleton !!!
Donc PAS de STATIC devant `getConnection` !!!
Autre remarque : *pour les accès aux bases de données il vaut mieux utiliser un gestionnaire de connexion qui créée et libère les connexions d'une manière plus transparente => `Datasource` de JDBC 2.0 (obligatoire pour JEEE) est désormais préférée au `DriverManager`…*

↳ ***En ce qui concerne les accès aux BD (connection) :*** il vaut mieux utiliser un gestionnaire de connexion qui créée et libère les connexions d'une manière plus transparente => `Datasource` de JDBC 2.0 (obligatoire pour J2E) est désormais préférée au `DriverManager`…
En entreprise, on n'utilise pas des connexions simples mais des `Connection Pool` (réserve de connections)
→ **`Connection Pool`** : réserve de connexions : Ensemble limité de connexions actives et réutilisables, mises à la disposition des clients, et permettant à un certain nombre d'entre eux d'accéder rapidement et simultanément à un serveur d'application.
Note(s) : En gardant actives un certain nombre de connexions, la réserve de connexions élimine les longs délais inhérents à l'établissement de toute nouvelle connexion.
La réserve de connexions est très utilisée dans les serveurs de bases de données
→ Dans notre application `CabinetMedical`, on aurait pu choisir d'implémenter avec un tableau de connections ou d'utiliser un `PooledConnection` interface de `javax.sql` qui s'utilise avec un datasource
… mais pour commencer notre cas est simple et une simple `Connection` obtenue avec un `DriverManager` suffit

↳ ***Pour info :*** On doit vraiment passer par une classe qui propose une méthode `getConnection`…
En effet, on ne peut pas écrire une classe du type
`public class CabinetConnection extends Connection{…}`
… et écrire le code dans le constructeur
… car `java.sql.Connection` est une interface qui nécessiterait l'implémentation de toutes les méthodes de cette interface (voir Annexe I) c-a-d 36 méthodes et surtout il faudrait coder les méthodes telles que :
`close(), commit(), createStatement(), prepareCall(String arg0), rollback() , setAutoCommit` etc …
Rappelons que l'objet de type `java.sql.Connection` est renvoyer par la méthode `getConnection` du DriverManager. L'objet renvoyé appartient donc à une sous-classe de Connection qui contient des méthodes `close(), commit(), createStatement(), prepareCall(String arg0), rollback() , setAutoCommit` etc … codées… C'est le driver manager qui se charge d'instancier avec la bonne classe !!!
(même principe que `Graphics g` de `paintComponent` => c 'est la JVM qui instancie, ici, c'est le `DriverManager` qui instancie, mais surtout pas nous directement !!!)….

2. Programme complet avec la méthode : `public Connection getConnection() throws SQLException`
Sachant que la base s'appelle `cabinetMedical`, et qu'on utilise un driver de type I, on peut construire l'URL de connexion (voir transparent 12): `jdbc:odbc:cabinetMedical`

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import com.iut.cabinet.metier.CabinetTechniqueException;
public class SimpleConnection {

   // attribut statique destiné à recevoir la référence de l'instance unique
   private static SimpleConnection simpleConnection = null;


   //Constructeur privé
   private SimpleConnection() throws CabinetTechniqueException {
      // chargement du driver
      try{
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         }
      catch (ClassNotFoundException e) {
      throw new CabinetTechniqueException ("Erreur Driver : Classe
Introuvable"+e.getMessage());
   }

   }

// renvoie une instanciation unique d' un objet  static Singleton
//en  vérifiant si cette instance existe déjà ou si elle doit vraiment être créé
// méthode renvoyant la valeur d'instance : point accès unique et global
   public static synchronized SimpleConnection getInstance()
                                          throws CabinetTechniqueException
   {  if (simpleConnection == null)
      {simpleConnection = new SimpleConnection();}
   return simpleConnection; // retourne référence de l'instance créée ou existante
   }


   //Méthode permettant de récupérer la connection instanciée par le driver
Manager
      public Connection getConnection() throws CabinetTechniqueException {
       String url = "jdbc:odbc:cabinetMedical";
       String login = "";
       String password = "";
       Connection connection = null;//Initialisation variable obligatoire car
catch ...
      try {
         connection = DriverManager.getConnection(url,login,password);
         connection.setAutoCommit(false);
         }
      catch (SQLException e) {
         throw new CabinetTechniqueException ("Erreur SQL : Connexion
Impossible" +e.getMessage());
      }
      return connection;
   } // fin getConnection
}// fin classe SimpleConnection
```


3. Pour ouvrir une nouvelle connexion :

```
Connection co1 =null; // initialisation variable
      try {
         co1 = SimpleConnection.getInstance().getConnection();
      } catch (CabinetTechniqueException e) {
         e.printStackTrace();
      }
```

on pourrait relancer une CabinetTechniqueException dans l'IHM avec un message approprié banalisé :
Système en erreur : contacter l'administrateur
Pour cette année, on dispose d'une CabinetApplicationException que l'on lance avec un message approprié

Exercice 2: *Mise en place de requête sur la base :*
méthode `findAllPersonnes` *de la classe* `PersonneDAOJDBC`

2.1 Etape 1 pour coder `findAllPersonnes` : Récupération du contenu de la table `Personne` dans un objet de type `Resultet`

```
package com.iut.cabinet.metier;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;

public class PersonneDAOJDBC {

    public static Collection<Personne> findAllPersonnes(Connection c)
                                            throws CabinetTechniqueException
    {
        Collection<Personne> listPers =new ArrayList<Personne>();

        ////////////////////////////////////////
        /// Etape 1 : Recuperation du ResulSet
        ////////////////////////////////////////
        ResultSet rs= null;
        PreparedStatement pst = null;

        try {
            pst = c.prepareStatement("select * from Personne");
            rs = pst.executeQuery();
        } catch (SQLException e) {
                throw new CabinetTechniqueException("Pb avec la requête : select * from
Personne"+e.getMessage());

        }



        ////////////////////////////////////
        /// Fermeture des ressources
        /// ouvertes dans la méthode
        ////////////////////////////////////
        finally
        {
            try
                {
                if (rs !=null)    rs.close();
                if (pst !=null)  pst.close();
                } catch (SQLException e) {
                    throw new CabinetTechniqueException("Probleme lors de la fermeture des
ressources de la méthode findAllPersonnes"+e.getMessage());
                }
        }
            ////////////////////////////////
        /// renvoi de la collection
        ////////////////////////////////
        return listPers;
    }
} // fin classe PersonneDAOJDBC
```


Remarque : Pour une maintenance plus facile, il est d'usage de regrouper toutes les requêtes sous forme de constante de classes (`final`) , statiques ici (car on a choisi d'implémenter le DAO en classe tout `static` donc méthode `static` ne peut utiliser que des variable `static`.

```
package com.iut.cabinet.metier;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;

public class PersonneDAOJDBC {

private static final String reqAllPersonnes = "select * from Personne";

    public static Collection<Personne> findAllPersonnes(Connection c)
                                            throws CabinetTechniqueException

    {
        Collection<Personne> listPers =new ArrayList<Personne>();

        ////////////////////////////////////////
        /// Etape 1 : Recuperation du ResulSet
        ////////////////////////////////////////
        ResultSet rs= null;
        PreparedStatement pst = null;

        try {
            pst = c.prepareStatement(reqAllPersonnes);
            rs = pst.executeQuery();
        } catch (SQLException e) {
         throw new CabinetTechniqueException("Pb avec la requête : "+reqAllPersonnes
+e.getMessage());

        }



        ////////////////////////////////////
        /// Fermeture des ressources
        /// ouvertes dans la méthode
        ////////////////////////////////////
        finally
        {
            try
                {
                if (rs !=null)    rs.close();
                if (pst !=null)  pst.close();
                } catch (SQLException e) {
                    throw new CabinetTechniqueException("Probleme lors de la fermeture des
ressources de la méthode findAllPersonnes"+e.getMessage());
                }
        }
            ////////////////////////////////
        /// renvoi de la collection
        ////////////////////////////////
        return listPers;
    }
} // fin classe PersonneDAOJDBC
```

2.2 Etape 2: Parcours du ResultSet pour la création de la collection de `Personne`.

```
private static Personne getBonnePersonne(ResultSet rs,Connection c)
                              throws CabinetTechniqueException{


   Personne pers = null;
   try {
       switch (rs.getInt("idTypePersonne")) {

      case 0 : // Patient
          pers=new Patient();
          ((Patient) pers) .setNir(rs.getString("nir"));
          ((Patient) pers) .setMedecinTraitant(rs.getString("medecinTraitant"));
          break;

      case 1 : // Professionnel
           pers=new Professionnel();
    ((Professionnel) pers) .setImmatriculation(rs.getString("immatriculation"));
    ((Professionnel) pers) .setSpecialite(rs.getString("specialite"));
            break;

     default : //Autre
               break;
      }

    pers.setDateNaissance(rs.getDate("datenaissance"));
    pers.setEmail(rs.getString("email"));
    pers.setIdPersonne(rs.getInt("idPersonne"));
    pers.setMale(rs.getBoolean("male"));
    pers.setNom(rs.getString("nom"));
    pers.setPortable(rs.getString("portable"));
    pers.setPrenom(rs.getString("prenom"));

pers.setAdresse(AdresseDAOJDBC.findAdresseByIdPersonne(pers.getIdPersonne(),c));


  }
catch (SQLException e) {
     throw new CabinetTechniqueException("Probleme dans la méthode
getBonnePersonne "+e.getMessage());

   }

catch(CabinetMedicalException e)
   {
 throw new CabinetTechniqueException("Probleme dans la méthode getBonnePersonne
"+e.getMessage());

    }

 return pers;
}
```

Remarque : **catch**(CabinetMedicalException e)
car `setNir` ou `setAdresse` peuvent déclencher une `CabinetMedicalException`

Annexe 1: Méthode de l'interface `java.sql.Connection`

```
public void clearWarnings() throws SQLException
public void close() throws SQLException
public void commit() throws SQLException
public Statement createStatement() throws SQLException
public Statement createStatement(int arg0, int arg1) throws SQLException
public Statement createStatement(int arg0, int arg1, int arg2)throws SQLException
public boolean getAutoCommit() throws SQLException
public String getCatalog() throws SQLException
public int getHoldability() throws SQLException
public DatabaseMetaData getMetaData() throws SQLException
public int getTransactionIsolation() throws SQLException
public Map<String, Class<?>> getTypeMap() throws SQLException
public SQLWarning getWarnings() throws SQLException
public boolean isClosed() throws SQLException
public boolean isReadOnly() throws SQLException
public String nativeSQL(String arg0) throws SQLException
public CallableStatement prepareCall(String arg0) throws SQLException
public CallableStatement prepareCall(String arg0, int arg1, int arg2)
public CallableStatement prepareCall(String arg0, int arg1, int arg2,
public PreparedStatement prepareStatement(String arg0) throws SQLException
public PreparedStatement prepareStatement(String arg0, int arg1)
public PreparedStatement prepareStatement(String arg0, int[] arg1)
public PreparedStatement prepareStatement(String arg0, String[] arg1)
public PreparedStatement prepareStatement(String arg0, int arg1, int arg2)
public PreparedStatement prepareStatement(String arg0, int arg1, int arg2,
       int arg3) throws SQLException
public void releaseSavepoint(Savepoint arg0) throws SQLException
public void rollback() throws SQLException
public void rollback(Savepoint arg0) throws SQLException
public void setAutoCommit(boolean arg0) throws SQLException
public void setCatalog(String arg0) throws SQLException
public void setHoldability(int arg0) throws SQLException
public void setReadOnly(boolean arg0) throws SQLException
public Savepoint setSavepoint() throws SQLException
public Savepoint setSavepoint(String arg0) throws SQLException
public void setTransactionIsolation(int arg0) throws SQLException
public void setTypeMap(Map<String, Class<?>> arg0) throws SQLException
```

Pour info : On doit vraiment passer par une classe qui propose une méthode `getConnection`…
En effet, on ne peut pas écrire une classe du type
`public class CabinetConnection extends Connection{…}`
… et écrire le code dans le constructeur
… car `java.sql.Connection` est une interface qui nécessiterait l'implémentation de toutes les méthodes de cette interface (voir Annexe I) c-a-d 36 méthodes et surtout il faudrait coder les méthodes telles que :

`close()`, `commit()`, `createStatement()`, `prepareCall(String arg0)`, `rollback()` , `setAutoCommit` etc …
Rappelons que l'objet de type `java.sql.Connection` est renvoyer par la méthode `getConnection` du DriverManager. L'objet renvoyé appartient donc à une sous-classe de Connection qui contient des méthodes `close()`, `commit()`, `createStatement()`, `prepareCall(String arg0)`, `rollback()` , `setAutoCommit` etc … codées… C'est le driver manager qui se charge d'instancier avec la bonne classe !!! (même principe que `Graphics g` de `paintComponent` => c 'est la JVM qui instancie, ici, c'est le `DriverManager` qui instancie, mais surtout pas nous directement !!!)….

Problèmes avancées sur les DAO

(extrait du cours de Grin http://deptinfo.unice.fr/~grin/mescours/minfo/bdavancees/supports/patternspersistance-dao6.pdf)

DAO et exceptions

→ Les méthodes des DAO peuvent lancer des exceptions puisqu'elles effectuent des opérations d'entrées-sorties
→ Les exceptions ne doivent pas être liées à un type de DAO particulier si on veut pouvoir changer facilement de type de DAO
→ Pour cela, on crée une ou plusieurs classes d'exception indépendantes du support de persistance, désignons-les par **`DAException`** (ou **`DataAccessException`** ou **`DaoException`**)
→Les méthodes des DAO attrapent les exceptions particulières, par exemple les **`SQLException`**, et relancent des **`DAException`** (auxquels sont chaînées les exceptions d'origine pour faciliter la mise au point)

DAO et connexions

→ Une connexion peut être ouverte au début des méthodes du DAO, et fermée à la fin des méthodes
Cette stratégie va coûter cher si un pool de connexions n'est pas utilisé
→ Il est préférable que les connexions soient ouvertes par les clients du DAO
En ce cas, les connexions ouvertes doivent être passées au DAO.
Pour cela le DAO peut comporter une méthode **`setConnection(Connection c)`**
(la façon de faire dépend de l'API de persistance que l'on utilise ; avec JPA on passera le manager d'entité et avec Hibernate la session)

Qui gère les transactions ?

→ Un DAO pourrait démarrer et terminer lui même les transactions à chaque méthode.
Cependant il n'est pas rare de vouloir inclure un ou plusieurs appels de méthodes de DAOs dans une seule transaction
L'implémentation des DAOs doit donc permettre cette dernière possibilité : ce sont les clients du DAO qui vont gérer les transactions

Transactions gérées par les clients

→ C'est le client du DAO, et pas le DAO qui va indiquer quand une transaction doit être validée ou invalidée
Le DAO utilise la transaction en cours si elle existe

Exemple schématique JDBC - DAO

```
public class StyloDao {
private Connection conn;
public void setConnection(Connection c) {
this.conn = c;
}
public long create(...) {
PreparedStatement pstmt =
conn.prepareStatement(...);
pstmt.setString(...);
...
pstmt.executeUpdate();
}
```

Exemple schématique JDBC - client

```
Connection conn = ... ;
daoStylo.setConnection(conn);
daoRamette.setConnection(conn);
...
daoStylo.create(...);
daoFacture.update(...);
conn.commit();
```

Tout n'est pas parfait !

→ On vient de voir qu'avec un DAO JDBC, il faut passer une connexion ; avec un DAO JPA il faut passer un gestionnaire d'entités
Il est donc difficile de rendre l'utilisation des DAOs totalement indépendante du type de persistance si on veut gérer des types de persistance très différents
Malgré tout, l'utilisation des DAOs diminue fortement la dépendance vis-à-vis des types de persistance

Solution partielle

Par exemple, pour la gestion des transactions, le code différent concernera l'initialisation des DAOs (avec une connexion ou avec un autre objet)
La solution est de ne pas mettre la méthode **`setConnection`** dans l'interface du DAO et de *caster* le DAO dans un type concret, le temps de l'initialiser