Langage Java - Chapitre 11 - Base de données MySQL

JDBC (Java DataBase Connectivity) est une API (Application Programming Interface) fournie avec Java (depuis sa version 1.1) pour assurer l'accès des applications écrites en Java à une grande variété de bases de données (SGBD : Système de Gestion de Base de Données). Le package **java.sql** fournit toutes les fonctionnalités de l'API JDBC.

L'API JDBC est constitué d'un ensemble de classes permettant la connexion et la manipulation des diverses bases de données.

Pour chaque type de base de données, l'API JDBC nécessite un connecteur-driver (pilote) spécifique qu'il faut généralement installer. Il existe normalement des pilotes JDBC pour tous les types de SGBD existants.

Ce cours présente les principales classes de l'API JDBC et son utilisation avec le SGBD MYSQL. Les exemples proposés dans ce cours traitent des classes les plus utilisées, mais les solutions présentées ici ne sont pas les seules.

1. Installation du connecteur (pilote) JAVA-MYSQL

Le connecteur **mysql-connector-java** est dit de type 4 car il est entièrement écrit en Java, il fournit l'implémentation de la connexion des applications clientes Java à la base de données.

Il faut installer le connecteur **mysql-connector-java**, pour cela :

- Aller sur le lien **https://dev.mysql.com/downloads/connector/j/** (ou faire une recherche sur **mysql-connector-java** si ce lien n'est plus valable).
- Choisir la bonne version.
- Suivre les instructions et valider le téléchargement.
- Effectuer l'installation en décompressant le fichier zip ou en exécutant éventuellement le fichier .msi.

Le N° de version indiqué dans le cours n'est peut être plus le bon.

On suppose le connecteur maintenant accessible par

«chemin»\ mysql-connector-java-5.1.34-bin.jar

Par exemple avec Windows 8:

C:\Program Files (x86)\MySQL
MySQL Connector J\mysql-connector-java-5.1.34-bin.jar

Le dossier **«chemin»\docs,** ici

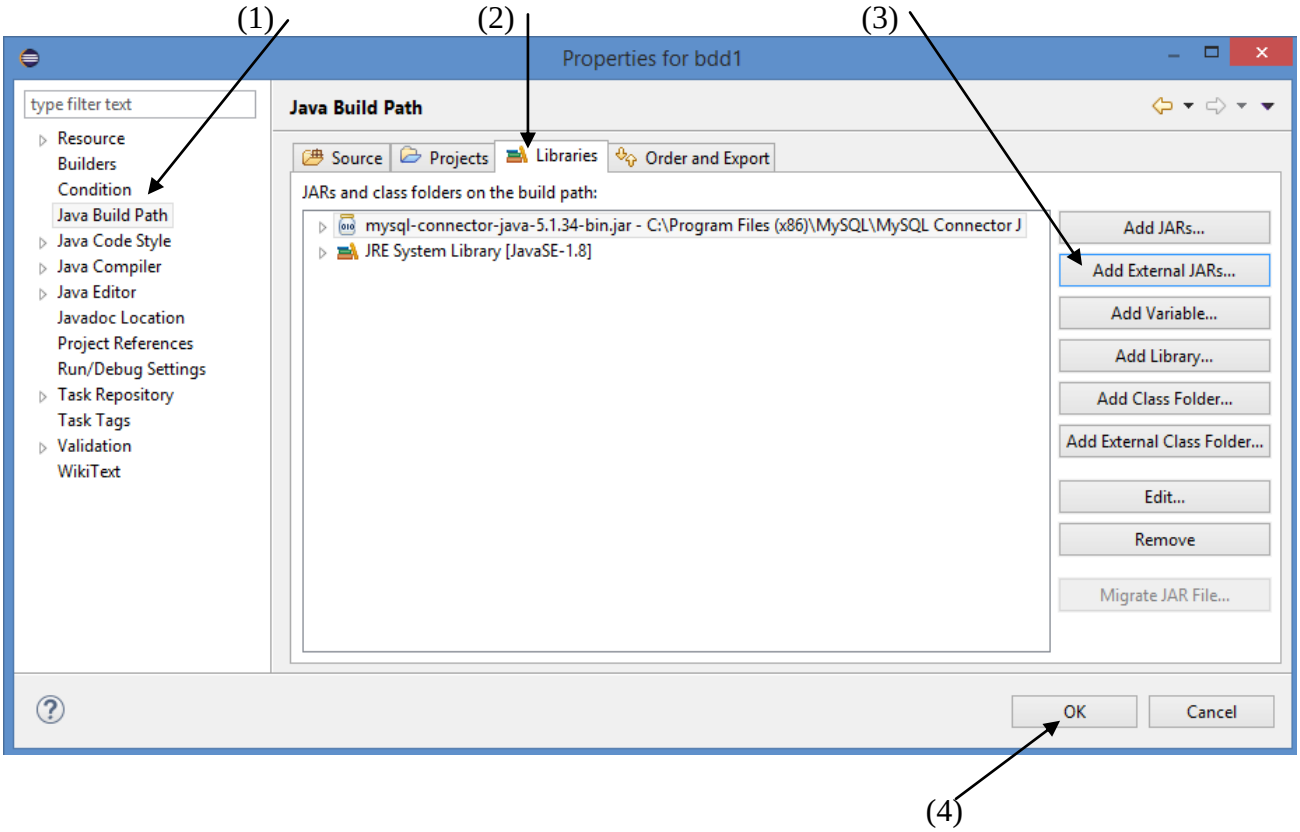
C:\Program Files (x86)\MySQL\MySQLConnectorJ\docs, contient alors la documentation nécessaire permettant l'utilisation du pilote.

Intégration de l'API mysql dans un projet avec Eclipse

On opère comme d'habitude, la 1ère étape consistant à faire un projet.

La seule nouveauté dans le mode opératoire est l'intégration du driver MySQL qui s'effectue de la manière suivante :

Clic droit sur le **Nom du Projet** (fenêtre de gauche, sommet de l'arborescence du projet)
Properties (propriétés) – situé en bas du menu contextuel -
Java Build Path (1)
Onglet **Libraries** (2)
Add External JARs… (3)
Se déplacer dans l'arborescence du système pour rechercher le dossier contenant le fichier jar
Sélectionner **mysql-connector-java-5.1.34-bin.jar**
puis **Ouvrir** : on doit voir la fenêtre suivante
puis **OK** (4)



2. Le programme Java pour se connecter à une base de données

Les interactions d'un programme avec une base de données se font en 4 étapes.

1ère étape : charger le driver

Une seule instruction suffit :

```
try {
        Class.forName("com.mysql.jdbc.Driver");
 }
 catch(ClassNotFoundException err){
            System.err.println("Pilote non trouvé..");
            System.err.println(err);
            System.exit(1) ;
```

```
    }
```

Remarque

On trouve quelquefois les instructions suivantes pour charger le driver mais la documentation Oracle précise que seule l'instruction précédente est suffisante pour la création d'une instance du driver et son enregistrement par le driver manager (DriverManager).

💣 Avec le driver **mysql-connector-java-8.0.19** il faut utiliser pour le nom de la classe: Class.forName("**com.mysql.cj.jdbc.Driver**").

```
        try {
              Class.forName("com.mysql.jdbc.Driver").newInstance();
        }
        catch(ClassNotFoundException err){
             System.err.println("Pilote non trouvé..");
             System.err.println(err);
             System.exit(1) ;
        }
       catch ( InstantiationException ex) {
                        System.out.println("Erreur instanciation");
                        System.exit(1) ;
         }
        catch ( IllegalAccessException ex ){
                        System.out.println("Erreur Accès illégal");
                        System.exit(1) ;
         }
```

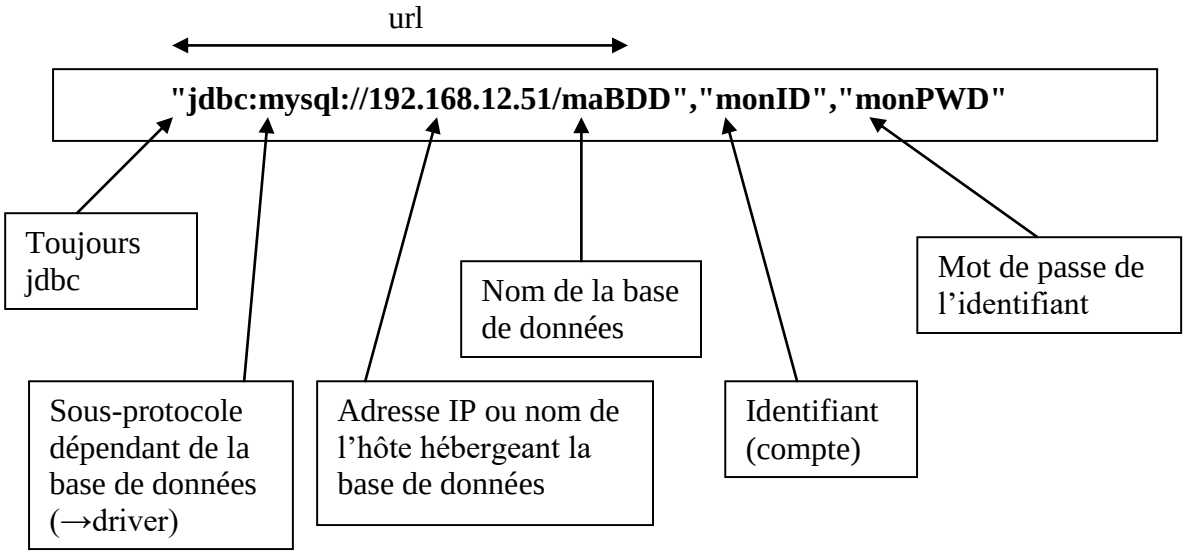
2ème étape, suite : se connecter à la base de données

```
Connection conn =null;       //objet de connexion
try {
  conn =
    DriverManager.getConnection("jdbc:mysql://192.168.12.51/bts","monty","monty");
}
catch(SQLException err){
    System.err.println("Connexion impossible");
    System.err.println(err);
    System.exit(1) ;
}
```

💣 Avec le driver **mysql-connector-java-8.0.19** il faut surement préciser le paramètre serverTimezone de l'URL:

DriverManager.getConnection("jdbc:mysql://192.168.12.51/bts?serverTimezone=UTC"
,"monty","monty");

💣 Le format des arguments de la méthode **DriverManager.getConnection()** est le suivant :



L'identifiant correspond à un compte MySQL, "monty" est à remplacer par un compte réel.

L'url est toujours de la forme : **jdbc:sous-protocole:sous-nom**

Le sous-protocole correspond au driver utilisé.

Le sous-nom permet d'identifier parfaitement la base de données concernée pour assurer la connexion.

Ces informations sont généralement données dans la documentation fournie avec le driver **(« chemin »\ mysql-connector-java-5.1.34\docs)**.

Si la base de données s'exécute sur l'hôte local :

url = **"jdbc:mysql://localhost/NomBaseDeDonnées"**

La méthode **DriverManager.getConnection(String url, String user, String passwd)** tente d'établir une connexion avec la base de données localisée à l'aide de l'argument **url** et dont les identifiants de connexion sont donnés dans **user** et **passwd**.

Le **DriverManager** doit alors sélectionner le driver approprié pour tenter d'établir cette connexion.

3ème étape, suite : exécution d'une requête SQL

L'exécution d'une requête SQL s'effectue via un objet de la classe **java.sql.Statement**.

C'est l'objet **Connection** obtenu précédemment qui fournit une référence à l'objet **Statement**.

```
try
{
Statement stat = conn.createStatement();
}
catch(SQLException ex) {
                    System.err.println("SQLException: " + ex.getMessage());
}
```

On distingue deux types de requêtes : les requêtes d'interrogation pour récupérer des données de la base et les requêtes de mise à jour (modification de la base).

• **Requête d'interrogation** typiquement utilisée avec l'ordre **SELECT**

On utilise la méthode de Statement **executeQuery( )** qui retourne un objet **java.sql.ResultSet**.

Pour lire la table 2tssnir :

```
try
{
ResultSet rs = stat.executeQuery("SELECT * FROM 2tssnir");
}
catch(SQLException ex) {
                    System.err.println("SQLException: " + ex.getMessage());
}
```

• **Requête de mise à jour** (modification de la base) avec les ordres **UPDATE**, **INSERT**, **DELETE.**
On utilise la méthode **executeUpdate( )** de **Statement** qui retourne un entier égal au nombre de lignes modifiées.
Pour supprimer l'enregistrement de l'étudiant "Maifaitrien" de la table 2tssnir :

```
try
{
int ret ;
ret = stat.executeUpdate( "DELETE FROM 2tssnir WHERE nom='Maifaitrien'");
}
catch(SQLException ex) {
                    System.err.println("SQLException: " + ex.getMessage());
}
```

4ème étape, suite : exploitation des résultats

►Cas d'une requête d'interrogation avec SELECT

L'objet **ResultSet** retourné par **executeQuery()** contient le résultat sous la forme d'un **tableau** (qui contient donc les données extraites par la requête **SELECT**).
On associe un **curseur** à ce tableau afin de le balayer ligne par ligne.
L'appel une 1ère fois à la méthode **next()** de cet objet **ResultSet** positionne le curseur sur la 1ère ligne de ce tableau, un 2ème appel à cette méthode le positionne sur la 2ème ligne et ainsi de suite.
Après avoir mis le curseur sur la dernière ligne, un nouvel appel à la méthode **next()** retourne **false**.

```
try
{
while (rs.next()) {
        String nom = rs.getString("nom");
        String prenom = rs.getString("prenom");
        int age = rs.getInt(3);
        System.out.println(nom + " " + prenom + " " + age );
        }
}
catch(SQLException ex) {
                    System.err.println("SQLException: " + ex.getMessage());
}
```

Chaque colonne de la table est identifiée par son nom ou par son rang qui commence à 1.

►Cas d'une requête de modification

Il faut exploiter la valeur retournée par **executeUpdate()** qui indique le nombre de lignes modifiées.
Ajout d'un enregistrement dans la table :

```
Statement stat = connection.createStatement() ;
 int ret = stat.executeUpdate("insert into 2tsiris(nom, prenom) " +
                        "values ('Aimar', 'Jean')") ;
 // exploitation du resultat
 if (ret >  0) {
   ...
}
```

Attention : ne pas oublier la fermeture de la connexion

Cette action est effectuée par l'appel de la méthode **close()** des objets **ResultSet**, **Statement** et **Connection.**

3. Un programme Java complet

```
import java.sql.*;

public class LireTable {

public static void main(String[] args){

Connection connection =null;
try{
      Class.forName("com.mysql.jdbc.Driver");
      connection =
         DriverManager.getConnection("jdbc:mysql://192.168.12.51/bts","monty","monty");
}
catch(ClassNotFoundException err){
         System.err.println("Pilote non trouvé..");
         System.err.println(err);
         System.exit(1) ;
}
catch(SQLException err){
         System.err.println("Connexion impossible");
         System.err.println(err);
         System.exit(1) ;
}
Statement stmt = null;
ResultSet rs = null;
try {
         stmt = connection.createStatement();
```

```
        rs = stmt.executeQuery("SELECT * FROM 2tsiris");
        while (rs.next())  {
                String nom = rs.getString("nom");
                String prenom = rs.getString("prenom");
                int age = rs.getInt(3);
                System.out.println(nom + " " + prenom + " " + age );
        }
}
catch (SQLException ex){
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
        if (rs != null) {
                try {
                rs.close();
                } catch (SQLException sqlEx) { }
        rs = null;
        }
        if (stmt != null) {
                try {
                stmt.close();
                } catch (SQLException sqlEx) { }
        stmt = null;
        }
        if (connection != null ){
                try {
                connection.close();
                } catch(SQLException sqlEx) { }
        }
  }
 }
}
```

4. Exercices

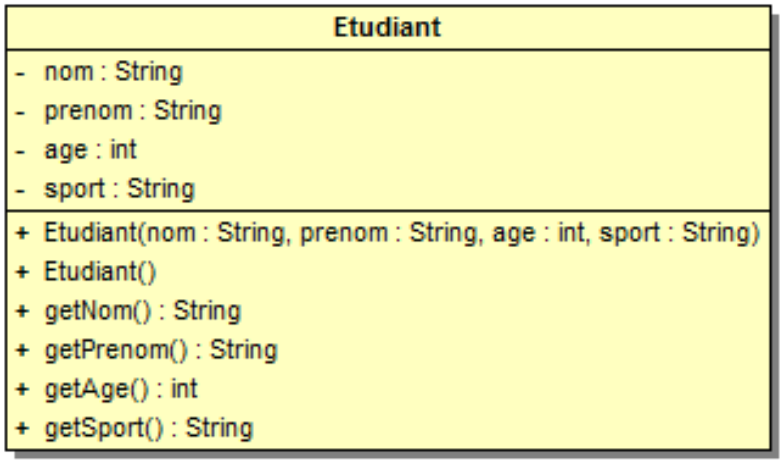
4.1 Etudier le programme donné au paragraphe 3 et, notamment, le bloc **finally**.

4.2 Taper et tester le programme donné au paragraphe 3 en changeant, si cela est nécessaire, la base de données **bts** par toute autre base et la table **2tssnir** par toute autre table de cette nouvelle base. La table 2tssnir est donnée à la fin de ce cours.

4.3 Modifier le programme afin d'ajouter l'enregistrement de l'étudiant de nom "NEFAIRIEN", de prénom "Jean", d'âge 25 ans et amateur de football. Il faut vérifier le fonctionnement en affichant la table résultante de cette opération.

4.4 Modifier le programme afin de supprimer l'enregistrement le l'étudiant ajouté en 4.3.

4.5 Ré organiser le code de l'application en créant une classe métier de nom **Etudiant** dont le modèle est donné ci-dessous :



4.6 Ecrire un programme permettant de remplir la table **2tssnir** avec des enregistrements.

Pour ceux qui ont besoin de créer la base bts avec la table 2tssnir

nom	prenom	age	sport
ADJEMI	Mohamed	20	football
AIT AAMEL	Brahim	20	football
AKA	Gnyanman	22	football
AQAJJEF	Soufiane	22	saut en hauteur
BENALDJIA	Nabil	20	handball
CISSOKHO	Hamidou	21	handball
DALIGANT	David	23	rugby
DAMANY	Arnaud	22	rugby
DIALLO	Mamadou	21	football
FARNER	Benjamin	19	tennis
GOMIS	Dendhioume	22	football
GONCALVES	Gael	21	football
HADJ-ARAB	Hakim	19	course
HAMON	Kevin	19	tennis
HELFER	Nicolas	21	tennis
ID AMHANE	Oussama	20	basket
LEGROS	Stephen	19	rugby
MARTIAS	Thomas	19	lutte
MBAPPE	Jean-Cyrile	21	course
MOLONGO	Benjamin	19	karaté
MOUMINI	Youssouf	20	basket
PIERRONNET	Frédéric	21	tennis
POPOTTE	Lamine	18	karaté
ROUSSET	Xavier	20	rugby
SIDIBE	Boubou	20	basket
SRIDAR	Johnson	20	football
YOUSSEF	Rami	20	football

5. L'interface PreparedStatement

Cette interface permet de créer des requêtes pré compilées et paramétrables. Un tel objet peut être ainsi utilisé plusieurs fois et l'application gagne en rapidité.
Chaque paramètre est indiqué dans la requête par un point d'interrogation puis est repéré par son rang numéroté à partir de 1.

Exemple : recherche de tous les étudiant qui ont 20 ans et qui aiment le football.

```
PreparedStatement preparee   =
    conn.prepareStatement("SELECT * FROM 2tssnir WHERE age=?  AND sport=?");

//On positionne le 1er paramètre
preparee.setInt(1,20);

//On positionne le 2ème paramètre
preparee.setString(2, "football");

//On exécute la requête
rs= preparee.executeQuery();

//On récupère le résultat
while (rs.next())  {
        String nom = rs.getString("nom");
        String prenom = rs.getString("prenom");
        System.out.println(nom + " " + email );
}
```

Pour une requête de modification, utiliser la méthode executeUpdate(…) de PreparedStatement.

Exercice

5.1 Tester l'exemple de code précédent.

5.2 Réalisation d'une classe **DaoEtudiant**

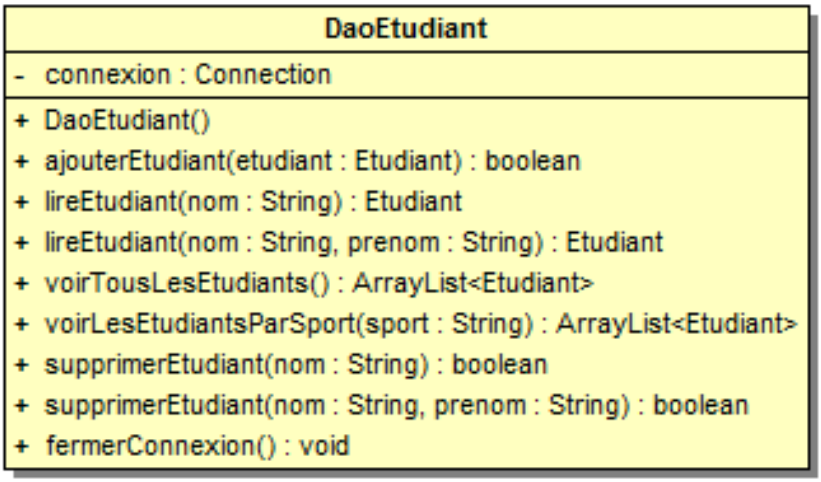
On organise le développement en 2 parties: une équipe s'occupe de l'accès à la base de données et du modèle de données; l'autre équipe s'occupe de la vue, c'est-à-dire de la présentation des données aux utilisateurs, cette partie n'est pas traitée ici.

Que demande le cahier des charges?

- ajouter un étudiant,
- voir un étudiant sélectionné à partir de son nom, et peut-être aussi de son prénom,
- voir tous les étudiants,
- voir tous les étudiants qui pratiquent un sport donné,
- supprimer un étudiant à partir de son nom, et peut-être aussi de son prénom.

Ces opérations sont appelées les opérations **CRUD**: Create, Read, Update et Delete.

On donne ci-dessous le modèle UML de la classe **DaoEtudiant** (dao: Data Access Object):



- Le constructeur établit la connexion à la base de données.
- La méthode **fermerConnexion** ferme la connexion à la base de données.
- Les méthodes **lireEtudiant** retournent null s'il n'y a pas d'étudiants correspondants aux critères de recherche.
- Les méthodes **voir..Etudiants..** retournent une ArrayList vide s'il n'y a pas d'étudiants correspondants aux critères de recherche.
- Les méthodes **ajouterEtudiant** et **supprimerEtudiant** retournent false si l'action ne peut pas se faire.

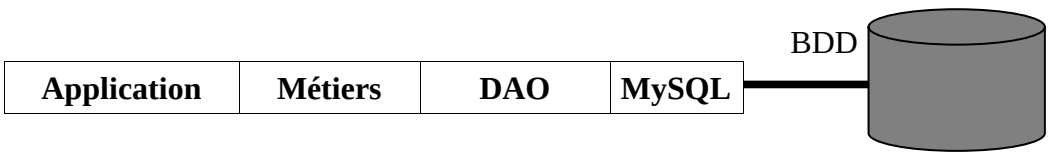
5.2.1 Coder la classe **DaoEtudiant**. Ecrire un programme de test validant les diverses méthodes.

5.2.2 Ajouter la possibilité de faire un Update (mise à jour) pour modifier le sport pratiqué par un étudiant.

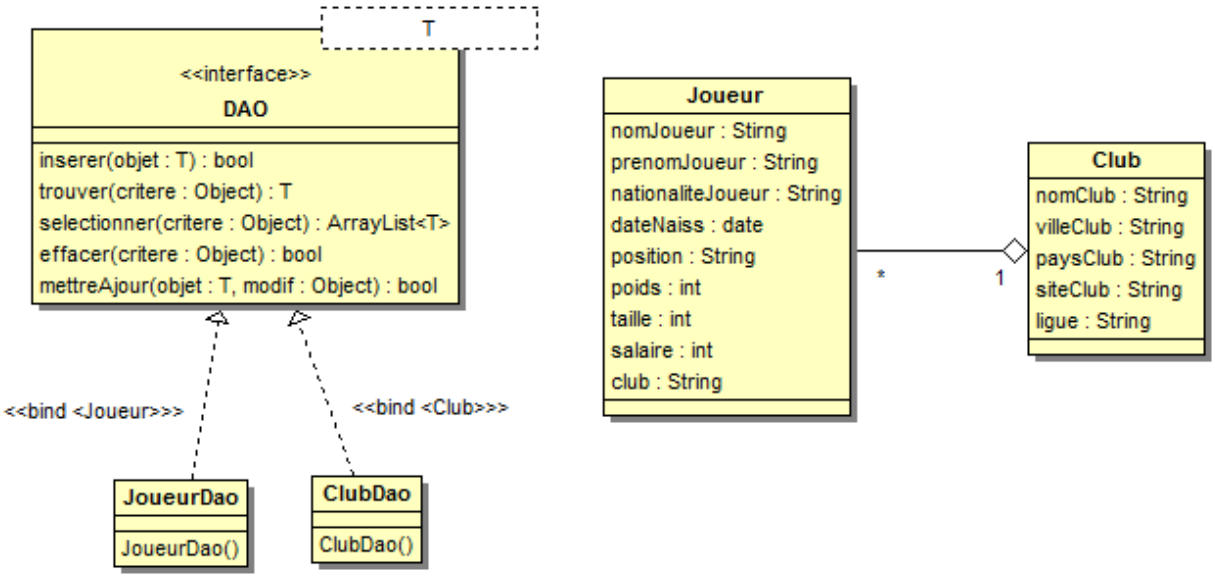
6. Gestion évoluée de la base de données avec une classe DAO

Un des objectifs est de diviser le développement en au moins 2 parties:

- une 1ère partie qui s'occupe de la gestion de la bdd, elle cache cette gestion à l'application en présentant une interface paramétrée DAO<T>,
- une 2ème partie qui s'occupe de l'application à proprement parler, elle accède à la bdd à l'aide des méthodes de l'interface proposée.



Modèle UML proposé



Une classe DAO doit fournir:
- les opérations **CRUD** : Create, Read, Update et Delete sur la base,
- éventuellement toutes sortes de requête spécifiques à l'application.

Les développeurs "application" et "dao" passent un contrat sur l'interface :

- les développeurs "application" utilisent l'interface,
- les développeurs "dao" la conçoivent (l'implémentent).

Les constructeurs des classes **JoueurDao** et **ClubDao** sont chargés de créer la connexion à la base. Il est possible de créer une nouvelle classe, **ConnexionBDD**, de type **singleton** pour ne pas avoir plusieurs connexions simultanées.
Les classes **Joueur** et **Club** doivent être munis des constructeurs et des accesseurs nécessaires.

Développer les classes du diagramme fourni.
Ecrire une application de test.