TD9: JDBC

JDBC[1] est une API Java permettant à un programme Java de communiquer avec un SGBD via le langage SQL. Familiarisons-nous avec cet API.

1 Présentation

- **JDBC** (Java Data Base Connectivity) est une API qui permet de se connecter à des bases de données et de les interroger via des requêtes SQL. Cette API nous permettra d'écrire un code indépendant du SGBD utilisé. Bien évidemment, il faudra disposer à l'exécution du driver JDBC (classe java spécifique à l'accès à un SGBD particulier) adapté au système de base de données cible.

Nous nous limiterons ici aux informations indispensables pour débuter.

2 Les drivers et leur chargement

En fonction du SGBD cible, vous devrez dynamiquement charger le driver adapté. Vous aurez à consulter la documentation de votre SGBD pour connaître la classe à référencer (n'oubliez pas de la rendre accessible à votre application).

Le chargement du driver se fera, comme vu dans le TD précédent, au travers de `Class.forName("nomCompletDeLaClasse")`.

Quelques exemples:

- **Java Db[2]:** le driver est `org.apache.derby.jdbc.ClientDriver`
- **Oracle:** le driver pour Oracle 9 est `oracle.jdbc.OracleDriver` disponible à l'ESI dans `"C:\ORAN9\jdbc\lib\classes111.jar"`.
- **ODBC** (Open Data Base Connectivity) est un protocole uniformisant le dialogue avec des systèmes de gestion de base de données. JDBC offre en standard un driver JDBC pour des SGDB que l'on peut présenter en tant que SGBD ODBC (càd pour lesquels on dispose d'un driver ODBC). Le driver est `sun.jdbc.odbc.JdbcOdbcDriver.` Ainsi, vous pourrez testez vos exemples avec une base Access, par exemple, pour autant que vous l'enregistriez dans ODBC. De manière générale, on évitera en java d'utiliser ODBC car la plupart des SGBD proposent des drivers natifs qui nous éviterons un *overhead* inutile.

Remarque: JDBC 4.0 offre l'auto-loading qui nous dispense de charger explicitement le driver. Ce chargement sera réalisé automatiquement lors de l'établissement de la connexion (cf. point suivant), le choix du driver se faisant sur base de l'url fournie.

1 Actuellement, de nouvelles API permettant de gérer la persistance des données voient le jour mais elles reprennent JDBC. Nous les aborderons l'année prochaine.
2 Java DB est la version supportée par Sun de Apache Derby ... un SGBD généralement fourni avec Netbeans

3 Établir une connexion

Pour établir une connexion vous devez fournir l'url (String) de la base de donnée à laquelle vous voulez accéder ainsi que l'utilisateur et son password:

```
Connection maConnexion = DriverManager.getConnection(url, "user", "password");
```

Remarque: nous verrons l'année prochaine une autre manière d'établir une connexion.

Quelques exemples d'url:

- ✔ **Java Db:** l'url à fournir aura la forme `jdbc:derby://host:port/nomBase` et pourra donc prendre, dans notre cas, la forme `jdbc:derby://localhost:1527/nomDb`.
- ✔ **Oracle:** l'url aura la forme `jdbc:oracle:thin:@host:port:instance`, qui sera donc ici `jdbc:oracle:thin:@oracle9:1521:esidb`.
- ✔ **ODBC** : l'url à fournir sera *`jdbc:odbc:OdbcNomDeLaBase`*.

4 Console SQL sous NetBeans

NetBeans offre la disponibilité, sous l'onglet *Services* et l'entrée *Databases*, d'une console SQL d'accès à vos BD (il est à remarquer que cette console ne vous offrira pas la notion de transaction, chaque requête étant exécutée en *AutoCommit*).

Utilisez la base *EmployeDepartement* sous Oracle et Access. Créez cette base aussi sous Postgres et JavaDb.

5 L'interface ResultSet

Les différents drivers que vous utiliserez implémentent l'interface `java.sql.ResultSet`.

Un *ResultSet* - analogue à un curseur en Embedded-SQL - nous permettra de parcourir les tuples retournés par l'exécution d'une requête de type Select.

5.1 Statement

Pour utiliser un ResultSet, il faut créer une instruction:

```
Statement stmt = connexion.createStatement()[1];
```

et ensuite éxécuter une requête de type Select:

```
ResultSet result =
      stmt.executeQuery("SELECT EmpNo, EmpNom FROM Employe");
```

1 Nous nous contenterons ici de voir les ResultSet qui ne permettent pas les mises-à-jour des tuples et qui ne permettent qu'un parcours séquentiel.

Après l'exécution de la requête, le curseur est positionné avant le premier tuple. Nous disposons alors de la méthode *next()* du ResultSet qui nous permet de parcourir les tuples obtenus:
•Tant que le curseur est positionné sur un tuple, la méthode next() permettra de le récupérer, avancera le curseur d'un tuple et nous renverra true.
•Dès qu'il n'y a plus de tuple à la position du curseur, la méthode next() nous renverra *false*.

Pour parcourir le ResultSet, nous aurons à écrire un code semblable à

```
	while (result.next()){
		String nom = result.getString("empnom");
		String num = result.getString("empno");
		System.out.println(num + " " + nom);
	}
```

Allez découvrir la javadoc de java.sql.ResultSet pour la description des méthodes getString(), getXXX, ...

5.2 Prepared Statement

Une requête préparée (*prépared statement*) est une requête SQL acceptant des paramètres. Une telle requête pourra être exécutée plusieurs fois avec des paramètres différents sans que le SGBD ne doive la recompiler ce qui apportera un gain en efficacité.

Dans l'exemple qui suit *dpt* et *sexe* reprennent des variables dont le contenu aurait pu être saisi par l'utilisateur.

```
	String query="Select empno, empnom From Employe "+
	            "Where empdpt = ? AND EmpSexe = ?";
	PreparedStatement stmt = connexion.prepareStatement(query);
	stmt.setString(1, dpt);
	stmt.setString(2, sexe);
	ResultSet result = stmt.executeQuery();
	while (result.next()){
		String nom = result.getString("empnom");
		String num = result.getString("empno");
		System.out.println(num + " " + nom);
	}
```

Attention, la correspondance entre les paramètres et les valeurs fournies se fait uniquement sur l'ordre dans lequel les paramètres apparaissent dans l'instruction!

Expliquez pourquoi il aurait été dangereux d'écrire le code suivant:

```
	String query="Select empno, empnom From Employe "+
	            "Where empdpt = '"+dpt+"' AND EmpSexe ='"+sexe+"'";
	Statement stmt = connexion.createStatement()[1];
	ResultSet result = stmt.executeQuery(query);
	while (result.next()){
		String nom = result.getString("empnom");
		String num = result.getString("empno");
		System.out.println(num + " " + nom);
	}
```

1 Nous nous contenterons ici de voir les ResultSet qui ne permettent pas les mises-à-jour des tuples et qui ne permettent qu'un parcours séquentiel.

6 Les requêtes autres que 'Select'

Pour les requêtes autres (update, delete, insert et même les requêtes du DDL) nous agirons de manière analogue en utilisant un *prepared statement* ou un *statement* en fonction du fait que nous ayons ou non des paramètres à faire passer. Nous appellerons alors la méthode *executeUpdate()* qui fera exécuter la requête et retournera – pour les instruction du DML – le nombre de lignes concernées par la modification.

7 Les transactions

Par défaut JDBC travaille en AutoCommit et donc chacune des requêtes envoyées au serveur est validée automatiquement. Pour débuter une transaction vous devrez faire exécuter:

```
connexion.setAutoCommit(false);
```

Nous verrons par la suite, lorsque nous aurons abordé cette notion en BD, que nous pouvons choisir un niveau d'isolation adapté à nos besoins.

Pour terminer la transaction, soit nous la validerons par

```
connexion.commit();
```

soit nous l'annulerons par

```
connexion.rollback();
```

8 Exemples - Exercices

Un tutoriel d'apprentissage est disponible chez SUN:
http://java.sun.com/docs/books/tutorial/jdbc/basics/index.html

1. Trouvez sur le net le nom des drivers à utiliser [ainsi qu'une adresse d'un site à partir duquel ces drivers sont téléchargeables] et le format de l'url à fournir pour les SGBD suivants:

   MySQL, PostgreSQL et Ms-SQL.

2. Pour mettre en œuvre de manière simple les éléments découverts dans le tutoriel précédent, il vous est demandé de réaliser une application 'console':
   - fournissant la liste des n° et nom de tous les employés;
   - fournissant la liste des n° et nom de tous les managers;
   - fournissant la liste des n°, libellé et masse salariale de chacun des départements;
   - augmentant le salaire de tous les employés d'un département dans une transaction (listez les noms et salaires de chacun de ces employés avant et après avoir réalisé la validation de la transaction);
   - idem que la précédente avec annulation de la transaction.

Testez votre programme en utilisant plusieurs des SGBD *Oracle*, *Java DB, PostgreSQL, Access.*

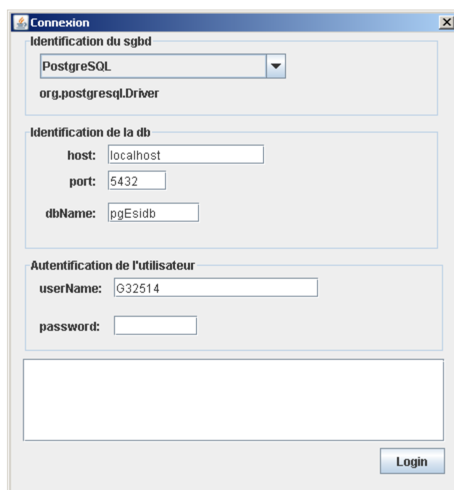
3. Écrivez une interface graphique permettant de saisir [au travers de JTextField] un département et un sexe et exécutez les 2 exemples du 5.2. Modifiez ensuite votre interface pour saisir le département au travers d'une comboBox et le sexe au travers de radiobuttons.

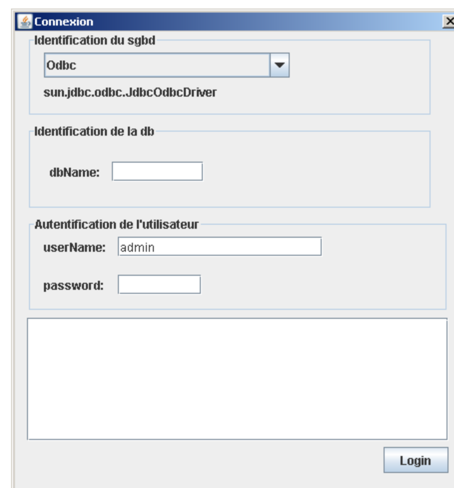
9 Fichier properties

Nous allons placer dans un fichier *Properties* les informations liées au choix du SGBD (nom du driver, format de l'url, et des valeurs par défaut comme le host, le port, l'utilisateur, ...). Il est clair qu'en pratique nous ne fournirons pas dans ce fichier le password. Veillez à utiliser le format XML pour le stockage.

Consultez la javadoc de `java.util.Properties`.

En exploitant ces '*properties*', nous pourrons créer un JDialog réutilisable pour établir une connexion:





Une combobox présentera la liste des SGBD prévus et, à chaque sélection, le nom du driver correspondant apparaîtra en-dessous.

Les informations relatives à la BD [variable en fonction du SGBD] proposent des valeurs par défaut modifiables.

Les informations liées à l'utilisateur apparaissent en-dessous, proposant déjà un nom d'utilisateur.

Si la connexion a pu être établie, le JDialog se fermera. Sinon, le message de l'exception apparaîtra en rouge dans la zone de texte du bas.