ENSAH

جامعة عبد المالك السعدي
Université Abdelmalek Essaadi

# Introduction à JDBC
## Accès aux bases de données

## Langage Java

**2ème année Génie Informatique**

**Ecole Nationale des Sciences Appliquées – Al Hoceima**

Prof A. Bahri
abahri@uae.ac.ma

A.U 2019/2020

# JDBC?

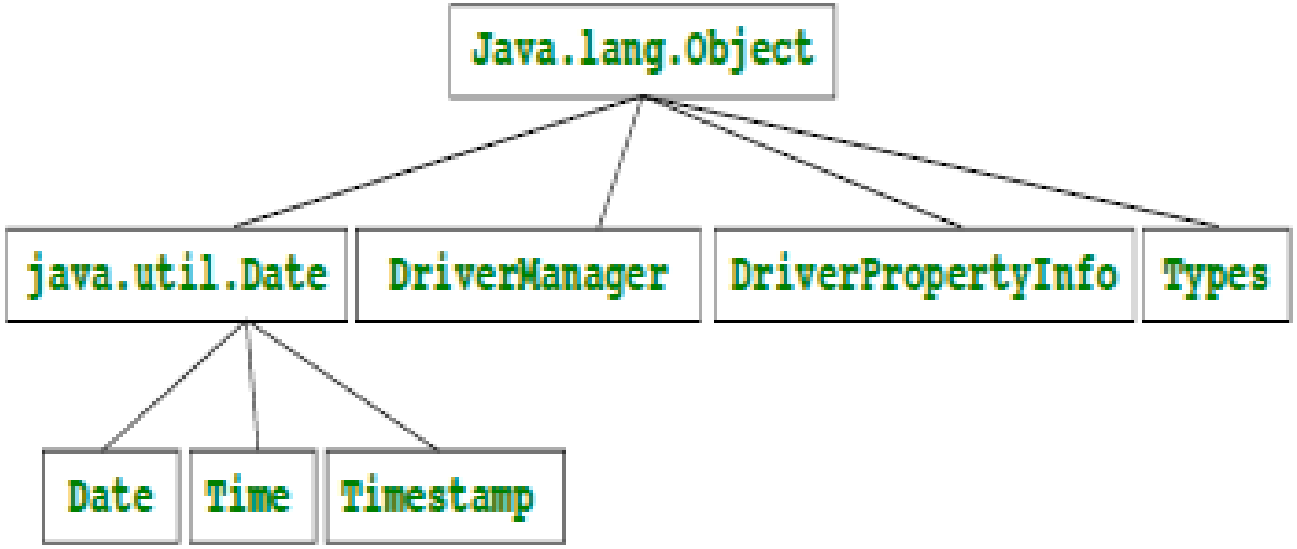
- JDBC (Java DataBase Connectivity) est une API développée par SUN pour permettre à des applications Java d'accéder à des bases de données relationnelles.

- JDBC manipule les bases de données relationnelles en utilisant le langage SQL.
  - Indépendament du type de la base utilisée (mySQL, Oracle, Postgres ...)
  - Seule la phase de connexion au SGBDR change

- Permet de faire tout type de requêtes
  - Sélection de données dans des tables
  - Création de tables et insertion d'éléments dans les tables
  - Gestion des transactions

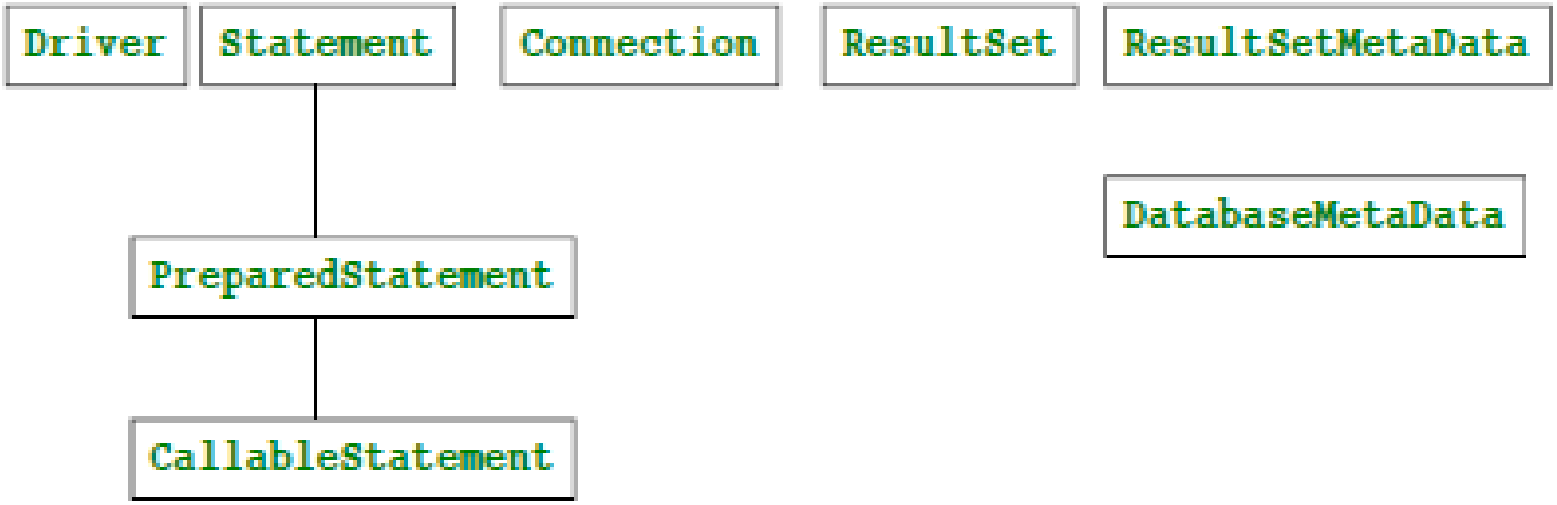
- **Packages :** `java.sql` **et** `javax.sql`

# Les classes du package « java.sql »




- **DriverManager** : gère les drivers, lance les connexions aux BD
- **Date** : date SQL
- **Time** : heures, minutes, secondes SQL
- **TimeStamp** : comme Time, avec une précision à la microseconde
- **Types** : constantes pour désigner les types SQL (conversions)

# Les interfaces du package « java.sql »




- **Driver** : renvoie une instance de Connection
- **Connection** : connexion à une BD
- **Statement** : instruction SQL
- **PreparedStatement** : instruction SQL paramétrée
- **CallableStatement** : procédure stockée dans la BD
- **ResultSet** : n-uplets récupérés par une instruction SQL
- **ResultSetMetaData** : description des n-uplets récupérés
- **DatabaseMetaData** : informations sur la BD

# Principes généraux d'accès à une BDD

## ❑ Première étape

- Préciser le type de driver que l'on veut utiliser
  - Driver permet de gérer l'accès à un type particulier de SGBD

## ❑ Deuxième étape

- Récupérer un objet « Connection » en s'identifiant auprès du SGBD et en précisant la base utilisée

## ❑ Etapes suivantes

- A partir de la connexion, créer un « Statement » (état) correspondant à une requête particulière
- Exécuter ce Statement au niveau du SGBD
- Fermer le Statement

## ❑ Dernière étape

- Se déconnecter de la base en fermant la connexion

# Utilisation du pont JDBC/ODBC

- ODBC permet d'accéder de manière identique à plusieurs types de bases de données distinctes (Access, SQL Server, Oracle, …).

- Pour ce faire, il utilise le concept de DSN (Data Source Name).

- Un DSN contient en fait les éléments nécessaires à la connexion à la base de données : type de la base, localisation du serveur, nom de la base, login, password, ...

# Utilisation du pont JDBC/ODBC

Il existe trois types de DSN la différence principale réside en la localisation du DSN

- **DSN utilisateur** : stocke ses informations dans la base de registre Windows sous l'entrée HKEY_CURENT_USER. Les données sont donc accessibles uniquement pour l'utilisateur en cours.
- **DSN système** : stocke ses informations dans la base de registre sous l'entrée HKEY_LOCAL_MACHINE. Tous les utilisateurs de la machine peuvent donc avoir accès à ces informations et donc se connecter à la base considérée.
- **DSN fichier** : stocke ses informations dans un fichier. Vous pourrez apposer une sécurité adaptée pour ne laisser qu'un certain nombre de personnes avoir accès à ces données.

# 1.Chargement du pilote de base de données

- La première chose que doit faire votre code est de charger le pilote en mémoire.

- Pour ce faire, il faut utiliser la méthode statique **forName** de la classe **Class**.

- Cette méthode possède un argument de type String contenant le nom de la classe du pilote

# 1.Chargement du pilote de base de données

- Syntaxe (SGBD Access) :

**Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**

- **Cette méthode génère une exception ClassNotFoundException**

```
try {
        Class.forName("com.mysql.jdbc.Driver");//sgbd MysQL
} catch(java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
}
```

- Access:
  - Class.forName("`sun.jdbc.odbc.JdbcOdbcDriver`");
- Oracle:
  - Class.forName("`oracle.jdbc.OracleDriver`");

- L'étape suivante est d'établir et d'ouvrir une connexion à la base de données
- La connexion peut échouer → gérer l'exception SQLException

# 2. Mettre en place une connexion JDBC

- **Classe `java.sql.DriverManager`**
  - Gestion du contrôle et de la connexion au SGBD

- Pour l ’ouverture d ’une connexion JDBC les méthodes Java doivent spécifier :
  - le nom du pilote JDBC à charger (étape 1)
  - les paramètres pour localiser la base de données.
  - le nom de l’utilisateur et son mot de passe (optionnels)

- La méthode statique **getConnection** de la classe **DriverManager** permet de créer un objet de connexion, en fonction du pilote chargé.

  - `static Connection getConnection(String url,String user,String password)`

```
String url = "jdbc:mysql://localhost:3306/myDataBase";
Connection con = DriverManager.getConnection(url[,
"myLogin", "myPassword" ] );
```

# 2. Mettre en place une connexion JDBC

```
Connection con = null;
try {
      String url="jdbc:mysql://localhost:3306/gestemployes";
       Class.forName("com.mysql.jdbc.Driver");
       con=DriverManager.getConnection(url,"root","");
       // . . .
       con.close();
} catch(ClassNotFoundException e) {
       System.out.println("Driver non chargé !");
 } catch(SQLException e) {
       System.out.println("Erreur dans la requête SQL");
}
```

# Exécution d'instructions sur la base

- Préparation de l'exécution d'instructions sur la base, 2 types
  - Instruction simple : classe **`Statement`**
    - On exécute directement et une fois l'action sur la base
  - Instruction paramétrée : classe **`PreparedStatement`**
    - L'instruction est générique, des champs sont non remplis
    - Permet une pré-compilation de l'instruction optimisant les performances
    - Pour chaque exécution, on précise les champs manquants
  - Pour ces 2 instructions, 2 types d'ordres possibles
    - Update : mise à jour du contenu de la base
    - Query : consulation (avec un select) des données de la base

# 3. Créer un objet Statement

- L'objet JDBC Statement envoie des instructions SQL pour la base de données.

- Les objets Statement sont créés à partir de l'objet de connexion active.

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

- Avec un objet Statement, vous pouvez émettre des appels SQL directement à la base de données.

# 3. Exécuter un objet Statement

- Classe **`Statement`**

  - `ResultSet executeQuery(String ordre)`
    - Exécute un `ordre` de type SELECT sur la base
    - Retourne un objet de type `ResultSet` contenant tous les résultats de la requête

```
String sql = "SELECT * FROM Employes";
ResultSet rs = stmt.executeQuery(sql);
```

  - `int executeUpdate(String ordre)`
    - Exécute un ordre de type INSERT, UPDATE, ou DELETE

```
String sql = "INSERT into Emplyes(id, nom, dateNaiss)
values (1, 'Ali Madani',#12/11/1975#)";
int rs = stmt.executeUpdate(sql);
```

  - `void close()`
    - Ferme l'état

# 3. Exécuter un objet Statement

## Établir une requête d'insertion

```
try {
    String url="jdbc:mysql://localhost:3306/gestemployes";
    Class.forName("com.mysql.jdbc.Driver"); //1:driver
    Connection con=DriverManager.getConnection(url,"root",""); //2:connection

    String strInsert = "INSERT INTO Employes VALUES (13, 'Titi', #1/5/60#);";

    Statement st = conn.createStatement(); //3: creation d'un objet Statement
    st.executeUpdate(strInsert);
    st,close();
    con.close();
} catch(ClassNotFoundException e) {
  // . . .
} catch(SQLException e) {
  // . . .
}
```

# 3. Exécuter un objet Statement

- Nous cherchons maintenant à récupérer l'ensemble des enregistrements de la table Employés.

- Il nous faut donc exécuter l'ordre SQL "SELECT * FROM Employés;".

- L'appel à "executeQuery" renvoie un objet de type **ResultSet** chargé avec les valeurs considérées

# 3. Exécuter un objet Statement

## Établir une requête de sélection

```
try {
   String url="jdbc:mysql://localhost:3306/gestemployes";
   Class.forName("com.mysql.jdbc.Driver");
   Connection con=DriverManager.getConnection(url,"root","");

   String strReq = "SELECT * FROM Employés;";
   Statement st = con.createStatement();
   ResultSet rs = st.executeQuery(strReq);
  // . . . Utilisation du ResultSet . . .
  con.close();
} catch(ClassNotFoundException e) {
  // . . .
} catch(SQLException e) {
  // . . .
}
```

# 3. Exécuter un objet Statement

## Parcours d'un ResultSet

```
String Req = "SELECT * FROM Employés;";
ResultSet rs = st.exexcuteQuery(Req);
while(rs.next()){
     System.out.println(rs.getInt(1)+" "+   rs.getString(2)+"
   "+rs.getString("DatNaiss"));
}
rs.close();
```

# 3. Exécuter un objet Statement

## Quelques méthodes de récupération de données

```
boolean getBoolean(int);
boolean getBoolean(String);
byte getByte(int);
byte getByte(String);
Date getDate(int);
Date getDate(String);
double getDouble(int);
double getDouble(String);
float getFloat(int);
```

```
float getFloat(String);
int getInt(int);
int getInt(String);
long getLong(int);
long getLong(String);
short getShort(int);
short getShort(String);
String getString(int);
String getString(String);
```

# Utilisation d'instructions préparées

- Jusqu'à présent, nous savons comment utiliser les objets JDBC Statement pour interroger ou mettre à jour les tables.
- L'objet *PreparedStatement* fournit des fonctionnalités similaires et offre deux avantages supplémentaires en plus:
  - Instructions SQL paramétrées
  - une exécution plus rapide
- La classe *PreparedStatement* est utilisée pour pouvoir envoyer au gestionnaire de base de données une requête SQL pour interprétation mais non pour exécution. Cette requête peut contenir des paramètres qui seront renseignés ultérieurement.

# Utilisation d'instructions préparées

## PreparedStatements sont plus rapides

La différence entre PreparedStatement et Statement:

- *PreparedStatement* est précompilée. Elle est compilée une et une seule fois. Donc, Elle présente de meilleure performance.

- En conséquence, *PreparedStatements* sont généralement plus efficaces, surtout si vous exécutez les mêmes *PreparedStatement* plusieurs fois.

# Création d'un objet PreparedStatement

- Comme pour les objets Statement, vous devez créer un objet PreparedStatement avec une méthode de connexion.

  - Par exemple:
  PreparedStatement ps = *con.preparedStatement ("UPDATE comptes SET solde = ? Where id = ?") ;*
  - Dans cet exemple, l '? indique un espace réservé de paramètre qui peut être réglé via l'API JDBC.

```
for (int i = 0 ; i < comptes.length ; i++){
    ps.setFloat (1,comptes [i].extraitSolde ()) ;
    ps.setInt (2,comptes [i].extraitIdf ()) ;
    ps.executeUpdate ()
}
```

# Exécution d'un PreparedStatement

- Classe **`PreparedStatement`**

  - Avant d'exécuter l'ordre, on remplit les champs avec
    - Exécute un `ordre` de type SELECT sur la base
    - `void set[Type](int index, [Type] val)`
      - Remplit le champ en i$^{ème}$ position définie par index avec la valeur val de type [`Type`]
      - [`Type`] peut être : `String, int, float, long` ...
      - Ex : `void setString(int index, String val)`

  - `ResultSet executeQuery()`
    - Exécute un `ordre` de type SELECT sur la base
    - Retourne un objet de type `ResultSet` contenant tous les résultats de la requête

  - `int executeUpdate()`
    - Exécute un `ordre` de type INSERT, UPDATE, ou DELETE

# Création d'un objet PreparedStatement

## Réglage des paramètres

- Une fois que vous avez votre ***PreparedStatement***, vous devez fournir les valeurs des paramètres pour chacun des points d'interrogation.

- Pour ce faire, en appelant l'une des méthodes de ***setXXX*** définies dans l'API ***PreparedStatement***.

- Si la valeur que vous voulez substituer à un point d'interrogation est un int Java, vous appelez la méthode ***setInt()***.

- Si la valeur que vous voulez substituer à un point d'interrogation est une String Java, vous appelez la méthode ***setString ().***

- En général, il existe une méthode de ***setXXX*** de chaque type dans le langage de programmation Java.

# Création d'un objet PreparedStatement

## Réglage des paramètres

❑ Arguments de setXXX:

- Le premier argument indique le point d'interrogation qui doit être réglé.
- Le deuxième argument indique la valeur de remplacement.

```
public static void main(String[] args) {
    java.util.Scanner scanner = new Scanner(System.in);
    System.out.print(« Saisir un l identifiant d un employé et son nom:");
    Int id= scanner.nextInt();
    String nom= scanner.nextLine();
    try {
      String url="jdbc:mysql://localhost:3306/gestemployes";
      Class.forName("com.mysql.jdbc.Driver");
      Connection con=DriverManager.getConnection(url,"root","");
       String sql = "select * from employes where cin=? and nom = ?";
       PreparedStatement ps = conn.prepareStatement(sql);
       ps.setString(1, id);
       ps.setString(2, nom);
       ResultSet rs = ps.executeQuery();
       while (rs.next()) {
          System.out.println(rs.getInt("cin") +
          "\t "+ rs.getString("nom") +
          "\t " + rs.getString("dateNaiss"));
       }
       rs.close();
    } catch (Exception e) { e.printStackTrace();}
}
```

# Aspect transactionnel

- Par défaut, les opérations sur la base de données sont en mode *auto-commit*. Dans ce mode, chaque opération est validée unitairement pour former la transaction.

- Pour rassembler plusieurs opérations en une seule transaction:
  - *connection.setAutoCommit(false);*
  - *connection.commit () ;*

- Retour en arrière:
  - *connection.rollback ();*

# Autres sujets

Si vous êtes curieux d'en apprendre plus sur JDBC, consultez la 2e partie du tutoriel sur Sun JDBC:

http://java.sun.com/docs/books/tutorial/jdbc/jdbc2dot0/index.html

Elle couvre des sujets tels que: les curseurs, les pools de connexions, etc.