BASE DE DONNÉES

JDBC ?

- Java DataBase Connectivity
  - API Java (Application Programming Interface)
  - Accès aux Bases de Données Relationnelles
  - Fonctionnalités :
    - Ouvrir une connexion avec un SGBD
    - Envoyer des requêtes SQL au SGBD
    - Récupérer des données résultantes des requêtes
    - Traiter ces données (tables)
    - Gérer les erreurs associées aux requêtes

- Similaire à l'API ODBC du langage C

Ressources associées au JDBC

- L'historique de JDBC
  - JDK 1.1 → JDBC 1.0 (1997)
  - JDK 1.2 → JDBC 2.0 (1998)
  - JDK 1.4 → JDBC 3.0 (2002)
  - JDK 1.6 → JDBC 4.0 (2007)

- Les API du JDK
  - java.sql
    - Toutes les classes de base pour manipuler les BDD relationnelles
  - javax.sql
    - Classes complémentaires (introduites avec le JDK 1.4)

- Les Drivers JDBC
  - Postgres : https://jdbc.postgresql.org/download.html
    https://jdbc.postgresql.org/documentation/head/connect.html

Schéma classique

BDD ➔ Données

1. Choix du Driver de connexion
2. Connexion à la base
3. Création de la requête SQL
4. Exécution de la requête
   - Récupération du résultat
   - Traitement des erreurs éventuelles
5. Boucle de traitement des données
6. Fermeture de la connexion

Le Driver

Le Driver JDBC

- Permet d'établir la connexion et la communication entre

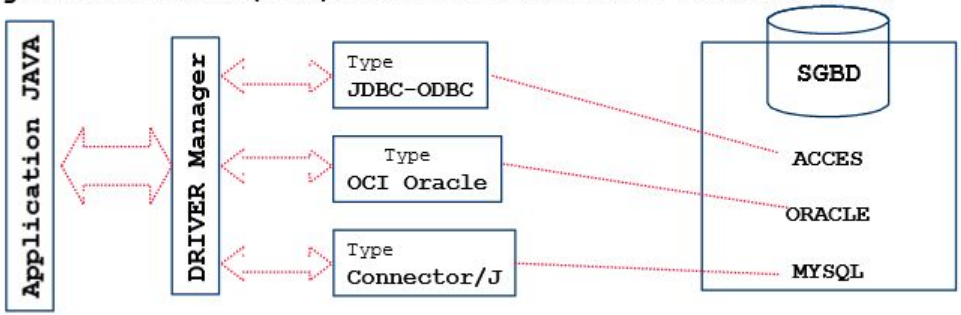
  Le programme Java ←→ Le système de gestion de bases de données

- **Chaque SGBDR possède donc son (ou ses) driver(s) spécifique(s)**

- Les classes liées aux drivers sont externes au JDK
- Elles doivent être liées à l'environnement de travail par le CLASSPATH
- Les drivers sont disponibles sur les sites des constructeurs ou sur le site de Sun

La classe **`DriverManager`** gère les drivers

- **`getConnection(xxx)`** retourne une connexion via le bon Driver



Chargement du driver

Une méthode (courante) consiste à utiliser la méthode `Class.forName`, qui aura pour effet d'enregistrer le Driver auprès du `DriverManager`.
N'oubliez pas de vérifier que le jar contenant le driver est bien dans le classpath

```
String nomDriver =
"nom_du_driver";  try{
    Class.forName(nomDriver);
}catch(ClassNotFoundException cnfe){
    System.out.println("La classe "+nomDriver+" n'a pas été trouvée");
    cnfe.printStackTrace();
}
```

En pratique, à cause d'implémentations imparfaites des spécifications, il sera parfois nécessaire d'utiliser cette syntaxe :

```
Class.forName(nomDriver).newInstance();
```

Exemple :

```
//pour le pont JDBC-ODBC
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

//pour MySQL et ConnectorJ
Class.forName("com.mysql.jdbc.Driver");
```

java.sql.Connection

class `Connection`

- Objet représentant une connexion

- **`DriverManager.getConnection(url,login,passwd)`**
  - url : « jdbc:<subprotocol>:<subname> »
    - <subprotocol> ➔ nom du driver
    - <subname> ➔ base de données (syntaxe liée au sous-protocole)
    - Exemple :
      - jdbc:mysql://monserveur.fr/mabase
      - jdbc:oracle:thin@//localhost:8000:base
      - jdbc:oracle:oci8@:base
- **`close()`**
  - ferme la connexion

- **`createStatement()`**
  - crée un objet « **`Statement`** » (requête SQL)

java.sql.Statement 1/2

interface `Statement`

- Représente une requête SQL
- Propose plusieurs méthodes d'exécution
  - **`executeQuery(String query)`**
    - Pour exécuter une requête SELECT
    - Le résultat est un objet **`ResultSet`** (contenant les données)
  - **`execute/executeUpdate/executeBatch()`**
    - Pour modifier la base (INSERT, UPDATE, DELETE, CREATE)
    - Pour faire des transactions
    - Le résultat des méthodes diffère selon l'action
- **`close()`**
  - Libère la mémoire du Statement
    - *Programme la libération par le garbage collector*

java.sql.ResultSet 1/2

interface `ResultSet`

- Contient les résultats d'une requête SQL

- **`getString(String nomDeColonne)`**
  - Renvoie la valeur contenue dans la colonne nommée.
  - La colonne doit être de type chaîne de caractère
    - (`VARCHAR` par exemple en MySQL)
  - Le résultat est un objet de type String

- **`getInt(String nomDeColonne)`**
  - Renvoie la valeur contenue dans la colonne nommée.
  - La colonne doit être de type numérique entier
    - (`INT` par exemple en MySQL)
  - Le résultat est une variable de type int

- **`next()`**
  - Permet de déplacer le curseur à la ligne suivante
  - Retourne `true` si l'opération est possible (`false` sinon)

1ère connexion

```
import java.sql.*;
public class Connect1 {
  public static void main(String args[]) {
    try {
      Class.forName("com.mysql.jdbc.Driver");
    } catch(ClassNotFoundException e) {
      System.err.println("Classe Driver MySQL non trouvée: "+e.getMessage());
    }
    try {
      Connection con =
         DriverManager.getConnection("jdbc:mysql://localhost/mabase","user","pass");
      Statement st = con.createStatement();
      String query = "SELECT * FROM livres";
      System.out.println("query : " +query);
      ResultSet rs = st.executeQuery(query);
      while ( rs.next() ) {
        String auteur   = rs.getString("auteur");
        String titre    = rs.getString("titre");
        int  nbPages    = rs.getInt("pages");
        System.out.println(titre+ ":" + auteur + "(" +nbPages+ " pages)");
      }
      rs.close();
      st.close();
      con.close();
    } catch(SQLException ex) {
      System.err.println("SQLException: " + ex.getMessage());
    }
} }
```

java.sql.ResultSet 2/2

java.sql.**ResultSet** *(suite)*

Un objet ResultSet contient les tuples (lignes) issus d'une requête. Sa structure est similaire à celle d'une table de la base.

- Des lignes
  - row, tuples ou n-uplets
  - La numérotation des lignes commence à 1
- Des colonnes
  - column, attribut d'un enregistrements
  - Chaque colonne possède
    - un nom
    - un type
    - un numéro
  - La numérotation des colonnes commence à 1
- `getXXX(String nomDeColonne)`
- `getXXX(int numeroDeColonne)`

Un curseur

Pointe sur une ligne

Permet le déplacement dans le ResultSet

A la création du ResultSet → position 0 (beforeFirst)

A la fin du parcours → position N+1 (afterLast)

La manipulation du curseur en dehors des limites du ResultSet lève des exceptions (SQLException)

`next()`

Permet de déplacer le curseur à la ligne suivante
Retourne `true` si l'opération est possible (`false` sinon)

SQL	méthode	JAVA
BIT	getBoolean	boolean
DATE	getDate	Java.sql.Date
DECIMAL	getBigDecimal	Java.math.BigDecimal
DOUBLE	getDouble	double
FLOAT	getDouble	double
REAL	getFloat	float
INTEGER	getInt	int
VARCHAR	getString	String

java.sql.SQLException

```
try {
     Connection con = DriverManager.getConnection("jdbc:mysql://localhost/java_iform","root","");
     Statement st = con.createStatement();
     String query = "SELECT mauvaischamp FROM livre";
     System.out.println("query : " +query);
     ResultSet rs = st.executeQuery(query);
     rs.close();
     con.close();
} catch ( SQLException ex ) {
     System.out.println("Exception SQL :
     ");  while (ex != null) {
       System.out.println("Message = "+ ex.getMessage()      +
                          "\nSQLState = " + ex.getSQLState() +
                          "\nErrorCode = "+ ex.getErrorCode() );
        ex.printStackTrace();
        ex = ex.getNextException();
}    }
```

Une `SQLException` contient :

- Un message
- Un statut SQL
- Un code d'erreur

La plupart des méthodes de l'API JDBC peuvent lever ce type d'exception

```
query : SELECT mauvaischamp FROM livre
Exception SQL :
Message = Champ 'mauvaischamp' inconnu dans field list
SQLState = 42S22
ErrorCode = 1054
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: Champ 'mauvaischamp' inconnu dans field list  at
         com.mysql.jdbc.SQLError.createSQLException(SQLError.java:936)
         at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2941)  at
         com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
         at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)  at
         com.mysql.jdbc.Connection.execSQL(Connection.java:3243)  at
         com.mysql.jdbc.Connection.execSQL(Connection.java:3172)
         at com.mysql.jdbc.Statement.executeQuery(Statement.java:1197)
         at Connect2.main(Connect2.java:16)
```

Arborescence des Exceptions

SQLTransientException

- **SQLTransientConnectionException**, **SQLTimeoutException**, **SQLTransactionRollbackException**
- Erreurs passagères
  - *(la requête pourrait éventuellement fonctionner si elle était relancée)*

SQLNonTransientException

- **SQLDataException** *(arguments non valides)*, **SQLSyntaxErrorException**, **SQLFeatureNotSupportedException**, **SQLIntegrityConstraintViolationException**, **SQLInvalidAuthorizationSpecException**, **SQLNonTransientConnectionException**
- Erreurs permanentes *(ne fonctionnera jamais tant que le problème n'est pas résolu)*

java.sql.Statement 2/2

interface `Statement`

- A chaque exécution de requête, le `Statement` retourne un `ResultSet`
  - On ne peut utiliser qu'un seul `ResultSet` par `Statement`
  - Pour parcourir en parallèle le résultat de 2 requêtes
    - Il faut 2 `Statement` distincts
- **`execute(query)`**
  - Méthode généraliste
  - Elle retourne un booléen (vrai si la requête retourne un `ResultSet`)
  - La méthode `getResultSet()` pourra être utilisée pour obtenir le `ResultSet`
- **`executeQuery(query)`**
  - Pour exécuter une requête SELECT
  - Le résultat est un objet `ResultSet` (contenant les données)
  - NB: Le résultat n'est jamais `null`
- **`executeUpdate(query)`**
  - Pour exécuter une requête de modification (INSERT, UPDATE, DELETE, CREATE)
  - Le résultat est un entier indiquant le nombre de lignes modifiées
  - Dans le cas d'une requête CREATE, le résultat vaut toujours 0
  - Dans le cas d'une requête INSERT, le statement peut fournir les références de la clé primaire créé Cf. **`getGeneratedKeys()`**

Requêtes de mise à jour (exemple)

```
. . .
Statement st = con.createStatement();
String query = "SELECT COUNT(*) FROM livres";
boolean etat = st.execute(query);
System.out.println("query : " +query+" boolean resultat="+etat);
if (etat) {
   ResultSet resultSet = st.getResultSet();
   // ResultSet resultSet = st.executeQuery(query); etait plus direct !
   resultSet.next();
   System.out.println("count ="+resultSet.getInt(1));
}
query = "INSERT INTO livres VALUES(null,'Prohibition','AlCapone',758)";
int nombre = st.executeUpdate(query);
System.out.println("query : " +query+" nb lignes ajoutees = "+nombre);
query = "DELETE FROM livres WHERE titre='Prohibition' AND auteur='AlCapone'";
nombre = st.executeUpdate(query);
System.out.println("query : " +query+" nb lignes supprimees="+nombre);
. . .
```

```
query : SELECT COUNT(*) FROM personne boolean resultat=true
count =4
query : INSERT INTO personne VALUES(null,'Ita','AlCapone',99) nb lignes ajoutees = 1  query :
DELETE FROM personne WHERE nom='Ita' AND prenom='AlCapone' nb lignes supprimees=1
```

java.sql.PreparedStatement

L'interface `PreparedStatement` permet de gérer des requêtes précompilées

- Utile lorsqu'une requête est renouvelée plusieurs fois avec des paramètres différents
- Chaque paramètre est représenté par un « ? » dans la requête

Un objet `Connection` permet d'obtenir un `PreparedStatement` à partir d'une requête SQL

- *connexion*.**prepareStatement**(query)
- *connexion*.**prepareStatement**(query, rsType, rsConcurrency)
  - *Comme pour la méthode `createstatement`, `rsType` et `rsconcurrency` paramètrent éventuellement le `ResultSet`*

Les principales méthodes d'un `PreparedStatement` sont :

- **setXXX**(indexParam, valeurParam)
  - `indexParam` est le numéro du paramètre variable dans la requête (le 1er porte le n°1)
  - `valeurParam` est la valeur donnée au paramètre
    - Exemple `setString(2,"ma chaine"); setInt(2, 234);`
- **clearParameters**()
  - Réinitialise tous les paramètres
  - Un paramètre en état « initial » ne permet pas d'exécuter la requête
- **execute**()
  - Retourne un booléen (true si le résultat de la requête est un ResultSet)
- **executeQuery**()
  - Dédié aux requêtes de type SELECT
  - Retourne un ResultSet contenant les résultats de la requête
- **executeUpdate**()
  - Dédié aux requêtes de mise à jour
  - Retourne le nombre d'enregistrements modifiés

Exercices

EXERCICES :

- Construire l’application de gestion d’une médiathèque