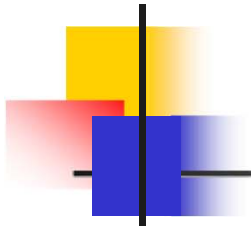
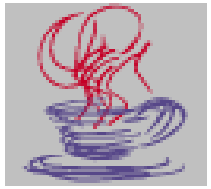
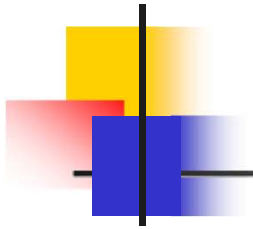




Java : Base de Données

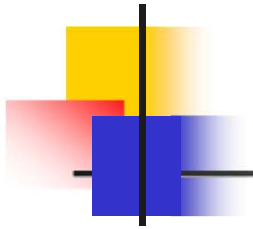
EISTI
Cergy - Pau



Sommaire

- Connexion JDBC - les DataSource
- Requêtes CRUD
- Fonctions et Procédures
- Batch et Transactions
- Exceptions

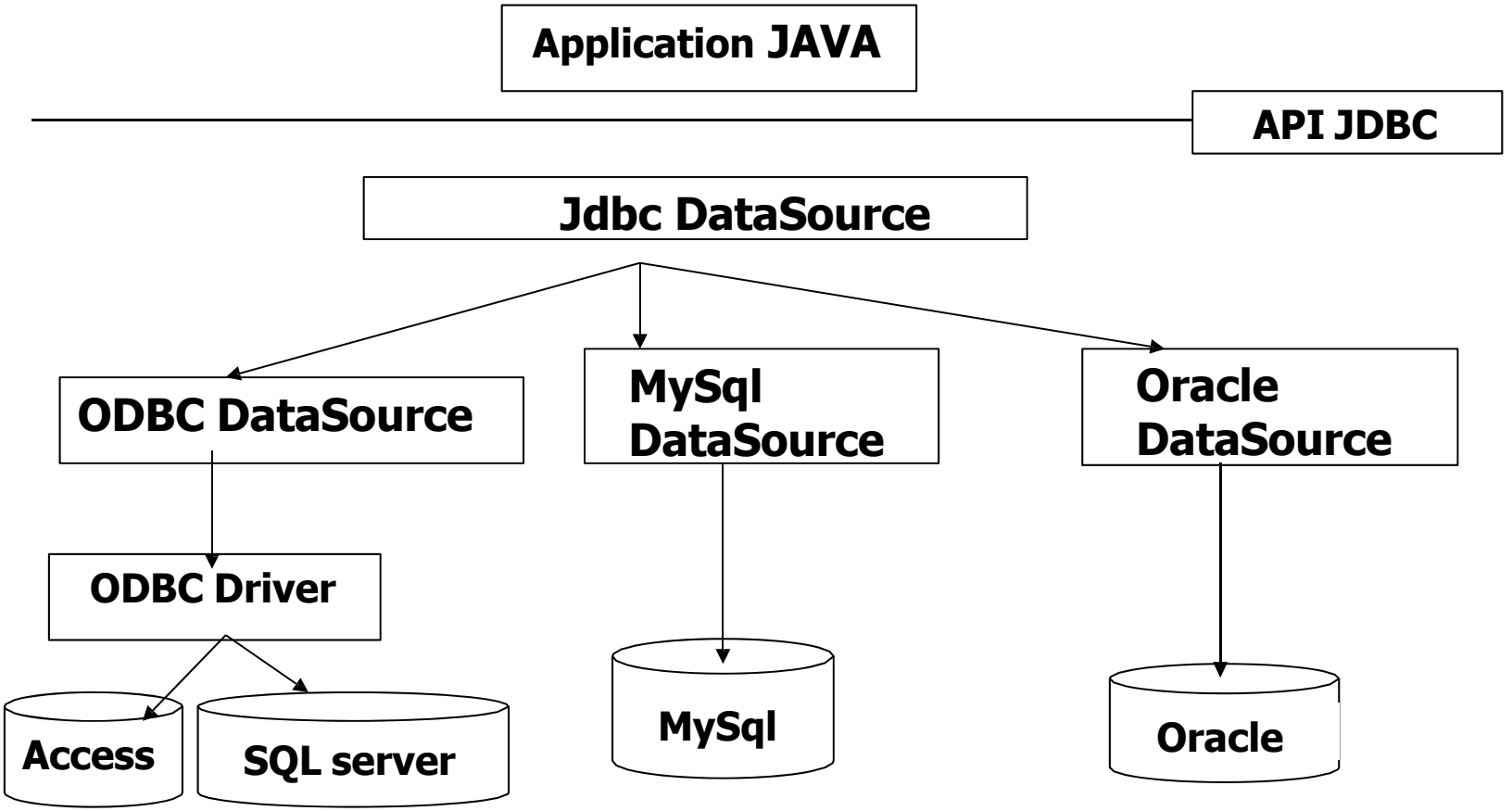
Connexion JDBC - les DataSource



Connexion JDBC - les DataSource

- JDBC : Java Database Connectivity
- Ensemble de classes Java permettant d'interfacer un programme Java avec une base de données SQL en agissant en tant que client du serveur SGBD local ou distant,
- Cette ensemble de classes permet de coder la partie Java sans connaître le SGBD auquel on est connecté (Oracle, MySQL, SQL Server, PostGreSQL, ...) sauf la connexion.
- Il existe 2 types de connexions :
  - Les **DriverManager** : ancienne version de connexion avec des drivers génériques comme l'ODBC pour Windows et d'autres spécifiques écrits tout ou partie en Java et fournis par les propriétaires des SGBD.
  - L'interface **DataSource** : apparu avec le JDK 1.4, c'est une fabrique de connexions vers une source de données. Elle améliore grandement les performances des DriverManager et permet de gérer des pools de connexion et des transactions distribuées. Les fournisseurs de pilote de SGBD doivent fournir au moins une implémentation de DataSource.

Architecture



Implémentation sous Oracle

- Inclure *ojdb6.jar* (Oracle 11 / *ojdb7.jar* Oracle 12) en tant que librairie externe du projet
- Récupération du DataSource :

```
import java.sql.SQLException;
import javax.sql.DataSource;
import java.sql.Connection;
import oracle.jdbc.pool.OracleDataSource;

public class MyConnexionOracle {

   public static Connection getConnexion () {
      try {
        OracleDataSource oracleDS = new OracleDataSource(); // Initalisation de la source
        oracleDS.setURL("jdbc:oracle:thin:@localhost:1521:xe"); // @server:port:instanceBDD
        oracleDS.setUser("user");
        oracleDS.setPassword("password");
        Connection connexion = oracleDS.getConnection(); // Récupération de la connexion
        return connexion;
      } catch (SQLException e) {
          e.printStackTrace();
          return null;
      }
    }
}
```

Implémentation sous MySQL

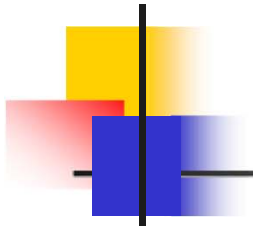
- Inclure *com.mysql.jdbc_5.1.5.jar* en tant que librairie externe du projet
- Récupération du DataSource :

```
import java.sql.SQLException;
import javax.sql.DataSource;
import java.sql.Connection;
import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;

public class MyConnexionOracle {

   public static Connection getConnexion () {
      try {
        MysqlDataSource mysqlDS = new MysqlDataSource(); // Initalisation de la source
        mysqlDS.setURL("jdbc:mysql://localhost:3306/cinema"); // @server:port:base
        mysqlDS.setUser("user");
        mysqlDS.setPassword("password");
        Connection connexion = mysqlDS.getConnection(); // Récupération de la connexion
        return connexion;
      } catch (SQLException e) {
          e.printStackTrace();
          return null;
      }
    }
}
```

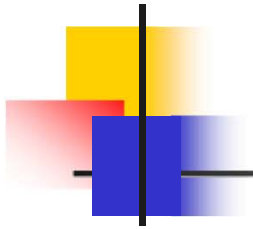
Requêtes CRUD



Requêtes simples

- Les requêtes simples sont :
  - Non optimisées
  - Sans paramètres dans la requête
  - Faites pour une utilisation unique ou faire des batch.
- On récupère un curseur de type ResultSet positionné AVANT le 1er enregistrement, qui doit être parcouru avec la méthode next().
  - ATTENTION : même s'il n'y a qu'un seul résultat, il faut quand même exécuter une fois next().
  - Les champs, numérotés à partir de 1, sont récupérés par des getter du type de la données:

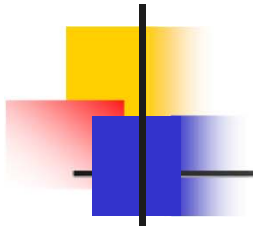
    `getTypeDeLaDonnée(N°champ)`
  - ATTENTION au type java.sql.Date différent de java.util.Date
- Les INSERT, UPDATE, DELETE et autres CRUD de LDD sont exécutés par la méthode executeUpdate et non un executeQuery.



Requêtes simples : exemple

- Exemple avec un try-with-resource (gère l'ouverture/fermeture des connexions) :

```
String req = "SELECT * FROM individu WHERE num_ind=5"; // Attention : pas de ";" à la fin
try (Statement selectUnIndividu = connexion.createStatement()) {
      ResultSet result = selectUnIndividu.executeQuery(req);
      if (result.next()) {  // positionnement sur le 1er enregistrement
          num_ind = result.getInt(1); // Récupération des champs par getTypeDeLaDonnée(N°champ)
          String nom = result.getString(2); // Attention : champs numérotés à partir de 1
          String prenom = result.getString(3);
          Individu individu = new Individu(num_ind, nom, prenom);
          return individu;
      }
      else
          return null;
} catch (SQLException ex) {
      System.err.println("Echec de lecture" + ex);
      // Message, code d'erreur SQL et code d'erreur du pilote
      System.err.println( ex.getMessage() + ex.getSQLState() + ex.getErrorCode );
      return null;
}
```



Requêtes préparées

- Les requêtes préparées sont :
  - **Précompilées** donc optimisées et plus performantes
  - On peut insérer des **paramètres** dans la requête avec un **"?"**. Ils peuvent être modifiés à chaque exécution, sans recompilation de la requête. On leur affecte une valeur par des setter du type de la données :

```
setTypeDeLaDonnée(N°champ, valeur)
```

  - ATTENTION : la numérotation des champs commencent à 1
- On récupère aussi un curseur de type ResultSet positionné AVANT le 1er enregistrement, qui doit être parcouru avec la méthode next()
- Les INSERT, UPDATE, DELETE et autres CRUD de LDD sont exécutés par la méthode executeUpdate et non un executeQuery.

Requêtes préparées : exemple

- Exemple avec un try-with-resource :

```
// Les paramètres sont signifiés par un "?" dans la requête, ordonnés à partir de 1
String req = "SELECT * FROM individu WHERE num_ind=? AND nom=?";
ArrayList<Individu> listeIndividus = new ArrayList<Individu>();
try (PreparedStatement selectUnIndividu = connexion.PreparedStatement(req)) {
   selectUnIndividu.setInt(1, 5);         // Affectation paramètres par :
   selectUnIndividu.setString(2,"Yoda"); // setTypeDeLaDonnée(N°champ, valeur)
   ResultSet result = selectUnIndividu.executeQuery();
   while (result.next()) {  // parcours du curseurs jusqu'à next() == null
     num_ind = result.getInt(1); // récupération des champs par getTypeDeLaDonnée(N°champ)
     String nom = result.getString(2); // Attention : champs numérotés à partir de 1
     String prenom = result.getString(3);
     Individu individu = new Individu(num_ind, nom, prenom);
     listeIndividus.add(individu);
   }
} catch (SQLException ex) {
   System.err.println("Echec de lecture : " + ex);
   System.err.println(ex.getMessage() + ex.getSQLState() + ex.getErrorCode );
}
```

Options du ResultSet

- L'interface ResultSetMetaData donne des informations sur le ResultSet. Exemple :

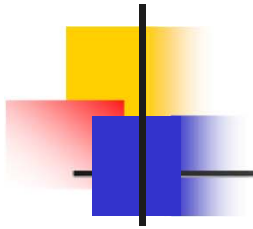
```
ResultSetMetaData rsmd = resultSet.getMetaData();
int nbCols = rsmd.getColumnCount();
```

- Remarque : l'interface DataBaseMetaData donne des informations sur la base de données dans son ensemble.
- Par défaut, un ResultSet est en avance seule et en lecture seule. Mais :
- CreateStatement peut prendre 2 arguments :

```
createStatement(int direction, int maj)
```

  - Les directions :
    - ✓ ResultSet.TYPE_FORWARD_ONLY ➔ parcours séquentiel en avance seule (par défaut)
    - ✓ ResultSet.TYPE_SCROLL_INSENSITIVE ➔Les occurrences ne reflètent pas les MAJ durant le parcours
    - ✓ ResultSet.TYPE_SCROLL_SENSITIVE ➔Les occurrences reflètent les MAJ faites durant le parcours
  - Les types de mise à jour :
    - ✓ ResultSet.CONCUR_READ_ONLY ➔ lecture seule (par défaut)
    - ✓ ResultSet.CONCUR_READ_UPDATABLE ➔ mise à jour possible, à valider par un rs.updateRow();

Fonctions et Procédures



Procédures stockées

- Pour exécuter une procédure stockée, on opère en deux ou trois temps :

  - Utilisation d'un objet de la classe CallableStatement, dérivée de PreparedStatement :

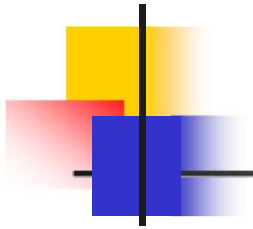
```
CallableStatement cs = req.prepareCall("{ call maProcédure(1, ?)}");
```

  - Si un paramètre de la procédure est inconnu, on le mentionne avec le symbole "?". Avant d'exécuter le call, on donne les valeurs des paramètres inconnus avec les instructions :

```
cs.setTypeDeLaDonnée(noParamètre, valeurParamètre);
```

  - On exécute la requête avec l'instruction :

```
cs.executeQuery();
```



Fonctions stockées

- Pour exécuter une fonction stockée, on opère aussi en deux ou trois temps :
  - Utilisation d'un objet de la classe CallableStatement, dérivée de PreparedStatement :

```
CallableStatement cs = req.prepareCall("{ ? = call maFonction(1, ?)}");
```

  - On donne le type du paramètre de retour de la fonction (1er paramètre).

```
cs.registerOutParameter(1, java.sql.Types.MonTypeDeRetour);
```

  - Si un paramètre de la procédure est inconnu, on le mentionne avec le symbole "?". Avant d'exécuter le call, on donne les valeurs des paramètres inconnus avec les instructions suivantes.
    ATTENTION : les numéros des paramètres commencent à partir de 2 car le 1er est réservé au retour de la fonction :

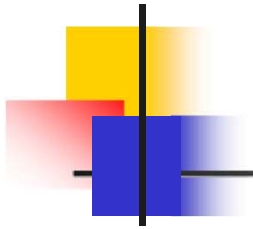
```
cs.setTypeDeLaDonnée(noParamètre, valeurParamètre); // Paramètres IN

// Paramètres OUT et IN  OUT pour les fonctions et procédures
cs.registerOutParameter(noParamètre, java.sql.Types.MonTypeDeRetour);
```

  - On exécute la requête avec l'instruction :

```
cs.executeQuery();
```

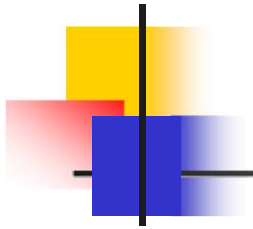
Batch et Transactions



Batch

- Quand on veut grouper une série de requêtes de mises à jour de la base dans un seul lot, il est préférable d'utiliser un batch pour un traitement par lots.
- Le principe :
  - Création d'un objet Statement: Statement batch = connexion.createStatement();
  - Ajout des requêtes: batch.addBatch("la requête");
  - Exécution : batch.executeBatch();
  - Validation : batch.commit();
  - Exemple :

```
connection.setAutoCommit(false);
Statement batch = connection.createStatement();
for(int i=0; i<10 ; i++) {
   batch.addBatch("INSERT INTO personne VALUES('nom"+ i +"','prenom«  + i + "')");
}
batch.executeBatch();
batch.commit();
```



Transaction

- Pour gérer une transaction, on utilise l'objet de connexion. On a alors les fonctions de gestion de transaction suivantes :
  - A la création d'une connexion, une transaction est initiée
  - Validation de transaction :

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

  - Annulation de transaction :

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

  - Création d'un point de sauvegarde :

```
SavePoint sp = connexion.setSavePoint();
```

  - Création d'un point de sauvegarde nommé :

```
SavePoint sp = connexion.setSavePoint("monPoint");
```

  - Annulation de transaction sur point nommé :

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

A vous de jouer !

☺