

# JDBC Avancé

# JDBC Avancé

## JDBC Versions

1. JDBC 1.0 : connections
2. JDBC 1.2 : Updatable ResultSets, DatabaseMetaData
3. JDBC 2.0 (Optional Package) : DataSource, JNDI, Pool de connections, transactions distribuées, Rowset
4. JDBC 2.1 (core) : resultset navigable, batch, SQL3, ...
5. JDBC 3.0 : savepoints, améliorations (...)
6. JDBC 4.0 : auto chargement du driver, SQL XML
7. JDBC 4.1 : amélioration (...)

# JDBC Avancé

JDBC (Core) : `java.sql`

- Améliorations Resultset, Scrollable resultset, Updatable resultset (utiliser des méthodes plutôt que des commandes SQL), manipulations BLOB and CLOB(coté serveur), Batch (traitements multiples), Savepoints

JDBC (Optionnel) : `javax.sql`

- JNDI, Pool (connections et statements), transactions distribuées, JavaBeans (objets RowSet), ...

# JDBC Avancé : ResultSet

3 types de ResultSet :

- TYPE\_FORWARD\_ONLY : ne peut pas être parcouru que dans un sens
- TYPE\_SCROLL\_INSENSITIVE : peut être parcouru dans les 2 sens, mais ne reflète pas les modifications faites dans la base après la récupération du ResultSet
- TYPE\_SCROLL\_SENSITIVE : peut être parcouru dans les 2 sens, et reflète les modifications faites dans la base après la récupération du ResultSet

# JDBC Avancé : ResultSet

Parallèlement à ces types, un ResultSet peut être

- `CONCUR_READ_ONLY` : on ne peut pas modifier les données en passant par le ResultSet
- `CONCUR_UPDATABLE` : on peut modifier les données en passant par le ResultSet

On peut donc avoir 6 types (3 x 2) de ResultSet

En fait, tous les drivers ne les permettent pas avec de bonnes performances

# JDBC Avancé : ResultSet

Choix d'un type de ResultSet: Des variantes des méthodes **createStatement**, **prepareStatement** et **prepareCall** de la classe **Connection** permettent de faire ce choix

- Permettre le parcours à double sens dans un ResultSet

```
Statement stmt = conn.createStatement(  
    ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_READ_ONLY);  
ResultSet srs = stmt.executeQuery(  
    "SELECT nomE, salaire FROM emp");
```

# JDBC Avancé : ResultSet

- Parcours en avant dans un ResultSet

**// A ajouter si on ne vient pas de récupérer**

**// le ResultSet : srs.beforeFirst()**

**while (srs.next()) {**

**String nomE = srs.getString("nomE");**

**double salaire = srs.getFloat("salaire");**

**System.out.println(nomE + " ; " + salaire);**

**}**

# JDBC Avancé : ResultSet

- Parcours en arrière dans un ResultSet

```
srs.afterLast();
```

```
while (srs.previous()) {
```

```
String nomE = srs.getString("nomE");
```

```
double salaire = srs.getFloat("salaire");
```

```
System.out.println(nomE + " ; " + salaire);
```

```
}
```



# JDBC Avancé : ResultSet

- Positionnement absolu et relatif dans un ResultSet

**srs.absolute(-2); // avant-dernière ligne**

**srs.absolute(4);**

**int numLigne = srs.getRow(); // rowNum = 4**

**srs.relative(-3);**

**int numLigne = srs.getRow(); // rowNum = 1**

**srs.relative(2);**

**int numLigne = srs.getRow(); // rowNum = 3**

- Les numéros de lignes et de colonnes commencent à 1 (pas à 0)

# JDBC Avancé : ResultSet

## ResultSet modifiable

- Si le select et si la colonne du select le permettent, il est possible de modifier la valeur d'une colonne d'une ligne renvoyée par le select et d'enregistrer cette modification dans la base de données
- Sinon, la méthode **updateXXX** ou la méthode **updateRow** lancera une exception

# JDBC Avancé : ResultSet

## ResultSet modifiable

- Le select d'un ResultSet modifiable doit
  - ne pas contenir de jointure ou de **group by**
  - contenir la clé primaire de la table
- Les expressions des colonnes modifiées doivent être de simples noms de colonnes de tables
- Exemple de colonne non modifiable : une colonne qui contient une expression avec une fonction SQL

# JDBC Avancé : ResultSet

- Création d'un ResultSet modifiable

```
Statement stmt = con.createStatement(  
    ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
ResultSet uprs = stmt.executeQuery(  
    "SELECT nomE, salaire FROM emp");
```

# JDBC Avancé : ResultSet

- Modifier des lignes par un ResultSet

**uprs.last();**

**uprs.updateDouble("salaire", 10000);**

**uprs.cancelRowUpdates(); // annule**

**uprs.updateDouble(2, 12000);**

**uprs.updateRow(); // enregistre modifs dans BD**

- Ne pas oublier **updateRow()** !
- **updateNull** permet de donner la valeur **NULL**

# JDBC Avancé : ResultSet

- Insérer des lignes par un ResultSet

**uprs.moveToInsertRow();**

//Va dans le buffer dans lequel seront rangées les  
//valeurs de la nouvelle ligne

**uprs.updateInt("matr", 150);**

**uprs.updateString("nomE", "Kleber");**

**uprs.updateDouble("salaire", 10000);**

**...**

**uprs.insertRow();**

# JDBC Avancé : ResultSet

- Supprimer la ligne courante d'un ResultSet

**uprs.absolute(4);**

**uprs.deleteRow();**

# JDBC Avancé : ResultSet

## Validation des modifications

- Comme pour les autres commandes de modification des données de la base, les modifications doivent être validées (resp. annulées) par un appel de la méthode **commit()** (resp. **rollback()**) de la connexion en cours (sauf si la connexion est en mode autocommit, ce qui n'est pas recommandé)



# JDBC Avancé : Batch

## Regrouper des modifications

- Dans les applications distribuées il est important de réduire le nombre d'accès distants aux bases de données pour améliorer les performances
- Les procédures stockées le permettent mais elles provoquent des problèmes de portabilité
- On peut aussi regrouper plusieurs ordres SQL de type DML (insert, update, delete) pour les envoyer en une fois au SGBD
- Un driver JDBC peut ne pas implémenter cette fonctionnalité

# JDBC Avancé : Batch

Regrouper des modifications

- 3 méthodes de l'interface **Statement** (et donc aussi de ses sous-interfaces) permettent de manipuler les regroupements d'ordres SQL
- Elles peuvent lancer une **SQLException**
- **void addBatch(String sql)** : ajoute un ordre SQL à la liste des ordres à exécuter
- **int[] executeBatch()** : exécute les ordres SQL
- **void clearBatch()** : enlève tous les ordres de la liste

# JDBC Avancé : Batch

## Méthode **executeBatch**

- Elle retourne un tableau d'entiers qui indique le nombre de lignes modifiées par chacun des ordres regroupés ; la valeur peut être négative s'il y a eu des problèmes
- Si une des commandes n'a pu être exécutée correctement, une **BatchUpdateException** est renvoyée ; les ordres SQL suivants sont exécutés ou non selon le driver; on peut alors choisir de valider ou non la transaction

# JDBC Avancé : Batch

Exemple de regroupement

```
Statement stmt = conn.createStatement();  
stmt.addBatch(  
    "INSERT INTO DEPT " +  
    "VALUES(70, 'Finances', 'Nancy')");  
stmt.addBatch(  
    "INSERT INTO DEPT " +  
    "VALUES(80, 'Comptabilité', 'Nice')");  
int[] nbLignes = stmt.executeBatch();  
conn.commit();
```

# JDBC Avancé : Batch

Exemple avec des requêtes paramétrées

```
PreparedStatement pstmt = conn.prepareStatement(  
    "INSERT INTO DEPT VALUES(?, ?, ?)";  
pstmt.setInt(1, 70);  
pstmt.setString(2, 'Finances');  
pstmt.setString(3, 'Nancy');  
pstmt.addBatch();  
// Ajout d'autres départements avec pstmt  
...  
int[] nbLignes = stmt.executeBatch();  
conn.commit();
```

# JDBC Avancé : RowSet

## Présentation

- Des ResultSet qui ont l'avantage de se conformer au modèle des *Java Beans* (*sérialisables*, avec propriétés, et observables par des écouteurs)
- Représentés dans l'API par des interfaces, dont l'interface racine **javax.sql.RowSet** hérite de ResultSet

# JDBC Avancé : RowSet

## Rowset déconnectable

- Certains rowsets peuvent être déconnectés de la base après avoir récupéré des données dans la base
- On peut alors modifier leurs données en mode déconnecté
- Le rowset peut ensuite se reconnecter et enregistrer les modifications dans la base

# JDBC Avancé : RowSet

## Sous-interfaces de RowSet

- Interfaces du paquetage **javax.sql.rowset** :
  - **JDBCRowSet** : rowset qui reste connecté
  - **CachedRowSet** : rowset déconnectable
  - **WebRowSet** : fille de **CachedRowSet** qui peut se sauvegarder au format XML
  - **JoinRowSet** et **FilterRowSet** : filles de **WebRowSet** qui représentent des rowsets sur lesquels on peut effectuer des jointures et des sélections quand ils sont déconnectés



# JDBC Avancé : RowSet

## **JdbcRowSet**

- C'est essentiellement une enveloppe autour d'un ResultSet, qui a les propriétés d'un Java bean
- Le rowset est modifiable (si le select le permet) et peut être parcouru dans les 2 sens, même s'il enveloppe un resultSet qui ne le permettait pas

# JDBC Avancé : RowSet

## **JdbcRowSet**

- 2 constructeurs :
  - avec un ResultSet en paramètre, pour « envelopper » un ResultSet existant
  - sans paramètre ; il faudra ensuite donner les informations pour la connexion à la base et pour indiquer les données à récupérer

# JDBC Avancé : RowSet

- **JdbcRowSet** – constructeur avec un paramètre `ResultSet`

```
Statement stmt = conn.createStatement();  
ResultSet r = stmt.executeQuery("select ... ");  
JdbcRowSet rs = new JdbcRowSetImpl(r);  
while (rs.next()) {  
String nom = rs.getString(1);  
...  
}
```

# JDBC Avancé : RowSet

**JdbcRowSet** – constructeur sans paramètre

```
JdbcRowSet rs = new JdbcRowSetImpl();
```

```
//Initialisation
```

```
rs.setUsername(...);
```

```
rs.setPassword(...);
```

```
rs.setUrl(...);
```

```
rs.setCommand("select ... ");
```

```
rs.execute(); //Récupération des données
```

```
while (rs.next()) {
```

```
String nom = rs.getString(1);
```

```
...
```

```
}
```

# JDBC Avancé : RowSet

## JdbcRowSet

- Outre le ResultSet, un rowset créé avec le constructeur sans paramètre enveloppe aussi un **PreparedStatement**
- La chaîne qu'on passe en paramètre de **setCommand** peut comporter des **joker** « ? »
- On peut passer les valeurs correspondantes par des méthodes setXXX comme pour les **PreparedStatement**

# JDBC Avancé : RowSet

## JdbcRowSet

- Exemple

```
rs.setCommand("select nome, salaire "  
+ " from employe where dept = ?");  
rs.setInt(1, 20);  
rs.execute();
```

# JDBC Avancé : RowSet

## **JdbcRowSet** : Modifications

- Les données contenues dans le rowset peuvent être modifiées avec les méthodes habituelles de **ResultSet**
- Les méthode commit et rollback de **JDBCRowSet** valident ou invalident les modification de la transaction courante
- Elles ne doivent être utilisées que si la transaction n'est pas en autoCommit ; voir méthodes **{get|set}AutoCommit** de **JDBCRowSet** (par défaut la connexion est en autoCommit)

# JDBC Avancé : RowSet

## **JdbcRowSet** : Modifications

- Exemple

```
rs.absolute(4);
```

```
rs.updateString("nom", "Dupond");
```

```
rs.updateRow();
```

```
rs.beforeFirst();
```

```
rs.next();
```

```
rs.updateInt(3, 135);
```

```
rs.updateRow();
```

```
rs.commit();
```



# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Il se connecte à la base juste le temps de récupérer des données
- Il peut être déconnecté de la base ; il est alors possible de lire, modifier, supprimer des données du rowset (même syntaxe que ResultSet) « en local »
- Il peut ensuite se reconnecter pour répercuter dans la base les modifications faites pendant la déconnexion (méthode **acceptChanges()**)

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Un **CachedRowSet** peut être rempli avec les données d'un **ResultSet** par la méthode **populate(ResultSet)**
- Cependant le plus simple est souvent d'initialiser le **CachedRowSet** pour qu'il puisse se connecter à la base (méthodes **setUsername**, **setPassword**, **setUrl** ou **setDataSourceName**), et indiquer la commande pour récupérer les données (**setCommand**)
- On peut ensuite lancer cette commande par la méthode **execute**

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- L'utilisation de la méthode **populate** ne renseigne pas le rowset sur la façon de se connecter à la base
- Une connexion nécessite donc de renseigner le rowset avec les méthodes **setUsername**, **setPassword** ; la base est indiquée par son URL (**setUrl**) ou par son nom de source de données (**setDataSourceName**)

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Les méthodes **execute** et **acceptChanges** peuvent recevoir en paramètre une connexion à la base
- En ce cas, cette connexion est utilisée pour lire ou écrire les données dans la base de données
- Sinon, le rowset ouvre une connexion en interne en utilisant les propriétés de connexion du rowset

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Les données d'un **CachedRowSet** peuvent être modifiées par divers méthodes updateXXX héritées de **ResultSet**
- Si la commande SQL le permet, les modifications peuvent ensuite être enregistrées dans la base de données grâce à la méthode **acceptChanges**
- Le rowset se reconnecte à la base, enregistre les modifications, puis se déconnecte

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Un commit est effectué à chaque appel de **acceptChanges**
- Les méthode **undoInsert()**, **undoDelete()** et **undoUpdate()** annulent la dernière modification de type insert, delete ou update effectuée sur le rowset
- Il est ainsi possible d'annuler plusieurs modifications qui ont été effectuée depuis le dernier **acceptChanges**

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Il peut évidemment y avoir des conflits au moment de la reconnexion à la base si les données initialement lues par le rowset ont été modifiées pendant sa déconnexion
- Le traitement de ces conflits dépend de l'implémentation du rowset
- L'implémentation de **CachedRowSet** fournie par défaut utilise un « blocage » optimiste

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Quand le rowset récupère les données dans la base, il enregistre ces données comme « valeurs originales »
- Au moment de l'enregistrement des modifications (**acceptChanges**) ces valeurs « originales » sont comparées aux valeurs actuelles de la base
- Il n'y a pas de conflit s'il y a égalité
- Sinon, c'est que les données ont été modifiées dans la base par un tiers, et il y a conflit



# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Si le **CachedRowSet** doit être modifié et si on veut répercuter ces modifications dans la base, il est indispensable d'indiquer une (ou plusieurs) colonne qui servira d'identificateur de ligne dans le rowset par la méthode **setKeyColumns**
- Le **CachedRowSet** pourra ainsi comparer les lignes du rowset avec les lignes de la base pour vérifier s'il n'y a pas de conflit
- Exp : La 1<sup>ère</sup> colonne servira à identifier  
**rs.setKeyColumns(new int[] {1});**

# JDBC Avancé : RowSet

**CachedRowSet** : RowSet déconnectable

- Schéma pour enregistrer les modifications

... **// Plusieurs modifications des données**

**rs.updateInt(2, 134);**

**rs.updateRow();**

...

**try {**

**rs.acceptChanges();**

**}**

**catch(SyncProviderException e) {**

**// Traitement des conflits**

...

**}**

# JDBC Avancé : RowSet

**CachedRowSet** : RowSet déconnectable

- Traitement des conflits

```
SyncResolver resolv = e.getSyncResolver();  
while (resolv.nextConflict()) {  
    if (resolv.getStatus() ==  
        SyncResolver.UPDATE_ROW_CONFLICT) {  
        int row = resolv.getRow();  
        rs.absolute(row);  
        int nbCol = rs.getMetaData().getColumnCount();  
        // Traitement des conflits d'une ligne
```

# JDBC Avancé : RowSet

## CachedRowSet : RowSet déconnectable


- Traitement des conflits d'une ligne
  - Pour les lignes qui ont provoqué un conflit, le « *resolver* » (***SyncResolver***) contient la valeur actuelle dans la base pour les colonnes de la ligne qui ont changé de valeur
  - Pour les autres colonnes, le *resolver* a la valeur **null**

# JDBC Avancé : RowSet

**CachedRowSet** : RowSet déconnectable

- Traitement des conflits d'une ligne

```
for (int j = 1; j <= nbCol; j++) {  
    if (resolv.getConflictValue(j) != null) {  
        Object valRs = rs.getObject(j);  
        Object valResolv = resolv.getConflictValue(j);  
        // Décide ce qu'il faut faire  
        ...  
        resolv.setResolvedValue(j, ...);  
    }  
}
```



La valeur qu'on a décidé  
de mettre dans la base

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Résolution des conflits
  - La méthode **setResolvedValue** met la valeur « originale » du rowset à la valeur actuelle de la base de données
  - Ainsi, au prochain appel de la méthode **acceptChanges**, il n'y aura plus de conflit (si la valeur dans la base n'a pas à nouveau été modifiée entre-temps)
  - Attention, après avoir résolu tous les conflits il ne faut pas oublier d'appeler **acceptChanges** pour enregistrer dans la base les valeurs choisies

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Autres possibilités
  - **release** permet de vider un CachedRowSet : il ne contient plus aucune données (mais les informations sur la connexion ne sont pas touchées)

# JDBC Avancé : RowSet

## **CachedRowSet** : RowSet déconnectable

- Autres possibilités
  - **CachedRowSet** permet aussi d'ajouter des observateurs et de les avertir si on change de ligne, si une ligne est modifiée ou si le rowset est rempli avec d'autres données (**add/removeRowSetListener, cursorMoved, rowChanged, rowSetChanged**)
  - Il est aussi possible de récupérer les données dans la base page par page lorsqu'il y a une grande quantité de données à récupérer (**setPageSize, nextPage, previousPage**)