# JDBC avancé

Université de Nice - Sophia Antipolis

Version 1.16 – 19/9/12

Richard Grin

## Plan de cette partie

- Génération des clés
- ResultSet « avancé »
- RowSet
- Regrouper les modifications
- Types de données SQL 3, BLOB et CLOB
- Pool de connexions, source de données
- Transaction distribuée

# Génération automatique de clés et JDBC



# Génération de clés

- La génération automatique de clés n'est pas standardisé
- Oracle utilise des séquences (comme DB2 et PostgreSQL)
- D'autres SGBD utilisent d'autres méthodes :
  - attribut **`AUTO_INCREMENT`** sur une colonne pour MySQL
  - attribut **`IDENTITY`** pour SQL Server
  - …

## Une situation courante

- Une situation peut nécessiter la connaissance d'une clé générée automatiquement
- Exemple : une facture est composée d'une table pour l'en-tête de la facture (numéro de la facture, client, date,…) et d'une table pour les lignes de la commande
- Pour écrire les lignes de la facture on a besoin de connaître la clé de l'en-tête de la facture (chaque ligne contient une clé étrangère vers cette clé de l'en-tête) et cette clé est générée automatiquement par le SGBD



## Récupération des clés générées

- La récupération d'une clé automatique ne devrait pas nécessiter de lancer une requête SQL avec un accès à la base de données (toujours coûteux)
- Il est possible d'indiquer que l'on souhaite recevoir les clés générées par l'exécution d'une requête SQL par **`Statement`** ou **`PreparedStatement`**

## 2 étapes pour récupérer les clés générées par un Statement « insert »

1. Indiquer au driver que l'on souhaite récupérer les clés qui seront générées automatiquement durant l'exécution du insert
2. Après l'exécution du insert, utiliser la méthode **`ResultSet getGeneratedKeys()`** pour récupérer les clés



## `getGeneratedKeys`

- L'interface **`Statement`** contient une méthode **`ResultSet getGeneratedKeys()`**
- Cette méthode s'appelle après l'exécution d'un insert (par **`executeUpdate`** ou **`execute`**)
- Elle renvoie toutes les clés générées par l'ordre SQL, indépendamment de la manière dont elles ont été générées
- Le **`ResultSet`** renvoyé est vide si aucune clé n'a été générée

## Lecture des clés générées

- La méthode **`getGeneratedKeys`** ne marche que si on indique, au moment de l'exécution du insert, que l'on souhaite récupérer les clés
- Pour **`Statement`**, il suffit pour cela de passer un 2ème paramètre à **`execute`** ou **`executeUpdate`**
- Pour **`PreparedStatement`**, on passe ce 2ème paramètre à **`prepareStatement`** (au moment de la création du **`PreparedStatement`**)



## Paramètre pour les clés générées

- Ce 2ème paramètre peut être
  - soit la valeur **`Statement.RETURN_GENERATED_KEYS`** ; en ce cas, le driver essaie de deviner quelles colonnes contiennent les clés
  - soit un tableau **`int[]`** ou **`String[]`** indiquant les colonnes du insert qui contiennent les clés générés

# Exemple

- Ajout d'une nouvelle facture dans la base
- On a besoin de la clé générée pour la facture pour la mettre en clé étrangère dans les lignes de la facture



# Exemple avec Oracle

```
Statement stmt = conn.createStatement();
// Ne pas oublier le 2ème paramètre optionnel
stmt.executeUpdate(
  "INSERT INTO facture (id_facture,...) " +
  "VALUES(sequence.nextval,...)",
  new String[] { "id_facture" });
ResultSet rsCles = stmt.getGeneratedKeys();
if (rsCles.next()) {
  cle = rsCles.getInt(1);
}
// cle utilisé pour insérer les lignes
// de la facture
```

## Clés générées et Oracle

- Pour Oracle, si on utilise une séquence pour générer la clé de la facture, on peut aussi utiliser **`sequence.currval`** dans les ordres insert des lignes de la facture



## Nouvelles possibilités de JDBC 2.0 et 3.0

- Parcourir un **`ResultSet`** dans les 2 sens
- Modifier les données de la base correspondant aux données renvoyées par un **`ResultSet`** directement par des méthodes de **`ResultSet`**, sans utiliser explicitement SQL
- Regrouper plusieurs ordres SQL pour les envoyer au SGBD
- Pools de connexions

# ResultSet



# Types de 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

## Types de ResultSet

- Parallèlement à ces types, un ResulSet 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



## Choix d'un type de ResultSet

- Des variantes des méthodes **`createStatement`**, **`prepareStatement`** et **`prepareCall`** de la classe **`Connection`** permettent de faire ce choix (consultez la javadoc des ces méthodes)

# 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");
```



- Les transparents suivants montrent les possibilités offertes par les ResultSet « scrollable » (ceux qui permettent de se déplacer où l'on veut dans les lignes)

## 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);
}
```



## 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);
}
```

# Positionnement absolu et relatif dans un ResultSet

```
srs.absolute(-2); // avant-dernière ligne
srs.absolute(4);
int numLigne = srs.getRow(); // numLigne = 4
srs.relative(-3);
int numLigne = srs.getRow(); // numLigne = 1
srs.relative(2);
int numLigne = srs.getRow(); // numLigne = 3
```

Les numéros de lignes commencent à 1 (pas à 0)



# 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

## ResultSet modifiable

- Le select d'un **`ResultSet`** modifiable doit (cf. conditions sur les vues)
  - 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



## Création d'un ResultSet modifiable

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

# 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



# Insérer des lignes par un ResultSet

```
uprs.moveToInsertRow();
uprs.updateInt("matr", 150);
uprs.updateString("nomE", "Kleber");
uprs.updateDouble("salaire", 10000);
. . .
uprs.insertRow();
```

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

## Supprimer la ligne courante d'un ResultSet

```
uprs.absolute(4);
uprs.deleteRow();
```



## Validation des modifications

- Comme pour les autres commandes de modification des données de la base, les modifications doivent être validées (resp. invalidées) 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)

## RowId

- **`java.sql.RowId`** est un nouvelle interface Java introduite par JDBC 4
- Ce type permet de stocker un identificateur pour une ligne d'un ResultSet afin de modifier par la suite la ligne en la désignant par cet identificateur
- RowId n'est pas supporté par tous les SGBD ; **`getRowIdLifeTime()`** de **`DataBaseMetaData`** indique si RowId est supporté, et la durée de vie d'un RowId



## RowSet

## Présentation

- **`ResulSet`** qui a l'avantage de se conformer au modèle des *Java Beans* (sérialisables, avec propriétés, et observables par des écouteurs)
- Représenté dans l'API par des interfaces, dont l'interface racine **`javax.sql.RowSet`** hérite de **`ResultSet`**
- Le JDK 5 fournit des implémentations des différentes sous-interfaces de **`RowSet`**



## Rowset déconnectable

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

## 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



## 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
- Une implémentation est fournie avec la distribution Java : **`com.sun.rowset.JdbcRowSetImpl`**

# 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



# 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);
  ...
}
```

# JdbcRowSet – constructeur sans paramètre

```
JdbcRowSet rs = new JdbcRowSetImpl();
rs.setUsername(...);
rs.setPassword(...);
rs.setUrl(...);
rs.setCommand("select ... ");
rs.execute();
while (rs.next()) {
  String nom = rs.getString(1);
  ...
}
```

Initialisation

Récupération des données



# Jokers dans la commande

- Outre le **`ResultSet`**, un rowset créé avec le constructeur sans paramètre enveloppe aussi un **`PreparedStatement`**
- La chaîne passée en paramètre de **`setCommand`** peut comporter des joker « ? »
- Les valeurs correspondantes sont passées par des méthodes setXXX comme pour les **`PreparedStatement`**

## Exemple

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



## Erreur à ne pas faire

- Il ne faut pas confondre
  - les méthodes setXXX, par exemple setInt, qui servent à donner des valeurs à des paramètres de la commande
  - les méthodes updateXXX, par exemple updateInt, qui modifient les valeurs des lignes du rowset

## Déplacement

- Un rowset peut être toujours être parcouru dans les 2 sens
- La syntaxe est semblable à celle de ResultSet



## Modifications

- Les données contenues dans le rowset peuvent être modifiées avec les méthodes habituelles de **`ResultSet`**
- Les méthodes **`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)

## Exemple de modifications

```
rs.absolute(4);
rs.updateString("nom", "Dupond");
rs.updateRow();
rs.beforeFirst();
rs.next();
rs.updateInt(3, 135);
rs.updateRow();
rs.commit()
```



## RowSet déconnectable

- Interface **`CachedRowSet`**
- 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()`**)

## Remplir un `CachedRowSet`

- 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`**



## Pagination et taille des données

- Un CachedRowSet garde ses données en mémoire ; lorsque les données de la base sont trop volumineuses il est possible de les récupérer par morceaux (par pages)
- On peut fixer une taille pour la page
- Le nombre de lignes contenues dans un rowset est donnée par la méthode **`size()`**

## Exemple de pagination

```
CachedRowSet crs = CachedRowSetImpl();
crs.setPageSize(100);
crs.execute();
while(crs.nextPage()) {
  // traite les lignes de la page courante
  while(crs.next()) {
    . . .
  }
}
```

Il existe aussi **`previousPage()`**



## Connexion d'un rowset

- L'utilisation de la méthode **`populate`** ne renseigne pas le rowset sur la façon de se connecter à la base
- Un 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`** ; voir la section sur les sources de données plus loin dans ce support)

## Connexions à la base

- 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



## Modifier les données

- Les données d'un **`CachedRowSet`** peuvent être modifiées par divers méthodes updateXXX héritées de **`ResultSet`**
- Si la commande SQL le permet (contraintes semblables aux contraintes pour les vues modifiables), 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

## Validation des modifications

- Le plus souvent un commit est effectué à chaque appel de **`acceptChanges`** (vérifiez-le dans la documentation de l'implémentation du rowset)



## Annulations de modifications

- 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`**

## Table modifiée

- Avec certains SGBD (Oracle en particulier) il peut être nécessaire d'indiquer la table sur laquelle les modifications seront faites, par la méthode **`setTableName(String)`**
- Avec d'autres, le rowset peut avoir cette information par les méta données
- Des implémentations (Creator de Sun par exemple) utilisent cette méthode pour restreindre à une seule table ce qui est inséré dans la base lorsque le select du rowset concerne plusieurs tables (permet de modifier des rowset qui ont un select avec jointure)



## Colonnes pour identifier les lignes

- 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 du rowset 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

## Code pour donner les colonnes identifiantes

```
// La 1ère colonne du rowset identifiera
rs.setKeyColumns(new int[] { 1 });
```



## Code schématique pour enregistrer les modifications

```
... // Plusieurs modifications des données
rs.updateInt(2, 134);
rs.updateRow();
...
try {
  rs.acceptChanges();
}
catch(SyncProviderException e) {
  // Traitement des conflits
  ... // la suite au prochain exemple…
}
```

## Conflits à la reconnexion

- Il peut y avoir des conflits au moment de la reconnexion à la base si les données lues par le rowset depuis la dernière synchronisation avec la BD (**`acceptChanges`**) ont été modifiées par un tiers pendant la déconnexion ; en ce cas il y un risque de perte de données
- Le traitement de ces conflits dépend de l'implémentation du rowset
- L'implémentation de **`CachedRowSet`** fournie par le JDK utilise un « blocage » optimiste



## Traitement optimiste des conflits

- 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 ; en ce cas, les modifications sont enregistrées et deviennent les nouvelles valeurs originales

## Détection des conflits

- Sinon, c'est que les données ont été modifiées dans la base par un tiers, et il y a conflit
- **`acceptChanges`** lance alors une **`SyncProviderException`**
- L'application peut alors demander à l'exception (méthode **`getSyncResolver`**) un **`SyncResolver`** pour l'aider à résoudre le conflit



## `SyncResolver` (1)

- Interface et qui représente un objet qui contient des informations sur les conflits
- C'est un rowset (hérite de **`RowSet`**) qui reflète celui qui a eu des conflits (mêmes lignes et colonnes), mais à la place des valeurs du rowset, il y a **`null`** s'il n'y a pas de conflit sur la valeur, ou la valeur correspondante de la base de données en cas de conflit
- Contient aussi des informations comme le type d'opération qui a causé le conflit

## `SyncResolver` (2)

- L'application peut se déplacer dans le syncResolver pour avoir des informations sur chacun des conflits et pour les résoudre
- Le déplacement peut se faire comme dans un rowset, avec en plus les méthodes **`nextConflict`** (resp. **`previousConflict`**) qui se positionne sur la prochaine (resp. précédente) ligne sur laquelle il y a eu un conflit



## Résolution des conflits

- L'application indique quelle valeur mettre dans la base de données par la méthode **`setResolvedValue(`*`numCol, valeur`*`)`** (ou **`setResolvedValue(`*`nomCol, valeur`*`)`** ) de **`SyncResolver`**
- Pour faire ce choix, l'application peut demander la valeur actuellement dans la base de données **`getConflictValue(numCol)`** par **`getConflictValue(`*`nomCol`*`)`** de **`SyncResolver`**

## Exemple de traitement des conflits

```
SyncResolver resolv = e.getSyncResolver();
// Se positionne sur les lignes à conflit
while (resolv.nextConflict()) {
  if (resolv.getStatus() ==
      SyncResolver.UPDATE_ROW_CONFLICT) {
    // Se positionne sur la ligne du
    // rowset liée à ce conflit
    rs.absolute(resolv.getRow());
    // Traitement des conflits d'une ligne
    // dans le transparent suivant
```



## Traitement des conflits d'une ligne

```
int nbCol =
  rs.getMetaData().getColumnCount();
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 qui sera mise dans la base

## Résolution des confllits

- 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



## Autres possibilités (1)

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

## Autres possibilités (2)

- **`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`**)



## Regrouper des modifications

## Performances

- 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



## Les méthodes

- 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 toutes les ordres de la liste

## 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 (voir javadoc pour plus de précisions)
- Si une des commandes n'a pu être exécutée correctement, une **`BatchUpdateException`** est renvoyée ; les ordres SQL suivants sont exécutés ou non suivant le driver (consulter la documentation du driver) ; on peut alors choisir de valider ou non la transaction



## 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();
```

## 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();
```



## Nouveaux types de données de SQL3

## Types de données de SQL3

- JDBC supporte les types suivants de SQL3 :
  - BLOB (*Binary Large Object*)
  - CLOB (*Character Large Object*)
  - ARRAY
  - types structurés définis par l'utilisateur
  - références vers des instances de types structurés
- Les 3 derniers types sont étudiés dans le cours sur JDBC et l'objet-relationnel



## `Blob` et `Clob`

- La plupart des SGBD fournissait déjà les types BLOB et CLOB mais
  - ces types n'étaient pas normalisés
  - l'accès par JDBC n'était pas normalisé
  - les possibilités de manipulation de ces types étaient restreintes (pas de recherche sur le contenu, par exemple)
- Il existe maintenant des types BLOB et CLOB et des interfaces Java Blob et Clob normalisés

## Manipulation des LOB

- Il existe plusieurs façons d'écrire ou d'enregistrer de LOB, en passant par les interfaces BLOB et CLOB ou non (utiliser directement les méthodes de ResultSet qui travaillent sur des flots)
- Les transparents suivants présentent quelques exemples (le traitement des exceptions a été enlevé pour faciliter la lecture du code)



## Interface `Clob`

- On peut lire le contenu d'une colonne de type **`Clob`** comme un flot de caractères ASCII ou Unicode par les méthodes (*valable seulement jusqu'à la fin de la transaction*)

```
InputStream getAsciiStream()
Reader getCharacterStream()
```

- On peut aussi lire un **`Clob`** comme une **`String`** ; on peut rechercher la position d'une sous-chaîne de caractères dans un **`Clob`**, et extraire une partie du **`Clob`** :

```
long position(String sousCh, long debut)
String getSubString(long pos, int longueur)
```

## Exemple de récupération de `Clob`

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(
  "select * from livre where isbn = 1421");
if (rset.next()) {
  Clob clob = rset.getClob("resume");
  long l = clob.length();
  // Affiche par paquets de 80 caractères
  for (long i = 0; i < l; i += 80) {
    String pRes = clob.getSubString(i, 80);
    System.out.println(pRes);
  }
} // Il reste à afficher le dernier paquet…
```



## Exemple de récupération de Blob

```
requete = "select pdf from livre where id = 1421";
ResultSet rs = stmt.executeQuery();
if (rs.next()) {
   // Flot en lecture dans la base
   is = rs.getBinaryStream(1);
   // Flot en écriture pour enregistrer le BLOB
   os = new FileOutputStream(nomFichier);
   byte[] buffer = new byte[1024];
   int length = 0;
   while ((length = is.read(buffer)) != -1) {
      os.write(buffer, 0, length);
   }
}
```

## Enregistrer un `Clob` ou un `Blob`

- La méthode **`setClob(int, Clob)`** de **`PreparedStatement`** met les objets de type **`Clob`** dans la base de données
- Pour les **`Blob`**, utiliser **`setBlob()`**
- Chaque driver peut fournir un constructeur de **`Blob`** et de **`Clob`**, par exemple à partir de tableau d'octets ou de caractères
- On peut aussi enregistrer un **`Blob`** ou un **`Clob`** à partir de tableaux, avec la méthode **`setObject()`** ou directement avec des flots



## Avertissement

- Les dernières parties de ce support ne sont que des survols rapides de possibilités avancées offertes par JDBC et ses extensions
- Le but est de faire connaître l'existence de ces possibilités pour que le lecteur intéressé puisse approfondir leur étude par d'autres sources

# Pools de connexions et sources de données



# Politique de connexion

- Une connexion à une base de données est une ressource rare et coûteuse qui ne peut être partagée par des *threads*
- Il faut donc réfléchir à la politique des connexions quand on écrit une application qui utilise une base de données :
  - Quand faut-il ouvrir et fermer une connexion ?
  - Faut-il garder une connexion ouverte entre 2 utilisations de la base ?

## Pool de connexions

- Les choix pour une politique de connexion sont facilités si on travaille avec un pool de connexions déjà ouvertes et disponibles pour les clients
- Lorsque le client n'a plus besoin de la connexion, il appelle la méthode **`close()`** qui la restitue au pool (la connexion n'est pas fermée)



## Comment obtenir un pool de connexions

- On peut utiliser un des nombreux projets « *open source* » qui fournit le code pour gérer un pool de connexion
- Le plus souvent un tel pool est fourni par une source de données de type **`javax.sql.DataSource`**

## Source de données

- Depuis JDBC 3 (mais déjà en extension de JDBC 2), on peut obtenir une connexion d'une instance de **`DataSource`** (une interface) au lieu de l'obtenir de la classe **`DriverManager`**
- Une **`DataSource`** représente une base de données (mais elle peut aussi représenter un simple fichier texte)
- Tout driver JDBC 2 doit fournir une implémentation de **`DataSource`** ; **`oracle.jdbc.pool.OracleDataSource`** pour Oracle



## Méthode `getConnection`

- La méthode principale de **`DataSource`** est
  **`getConnection(String utilisateur, String motDePasse)`**
  qui renvoie une **`Connection`** (et peut lancer une **`SQLException`**)
- Si le nom et le mot de passe de l'utilisateur ont été entrés dans la configuration de la source (par des *setters* par exemple), on peut aussi utiliser la méthode **`getConnection()`**

## Connexion d'une DataSource

- La connexion renvoyée par **`getConnection`** peut être une connexion ordinaire mais le plus souvent c'est une connexion renvoyée par un pool de connexions
- L'utilisation d'une telle connexion se fait exactement comme une connexion ordinaire ; tout est transparent pour l'utilisateur
- Une différence essentielle est que, lorsque le client appelle la méthode **`close`** de la connexion, celle-ci n'est pas fermée mais remise dans le pool des connexions disponibles



## Exemple d'utilisation

```
DataSource ds = new OracleDataSource();
((OracleDataSource)ds).setURL("jdbc:oracl
  e:thin:@euterpe.unice.fr:1521:INFO");
Connection conn =
  ds.getConnection("toto", "mdp");
// Le reste est identique au code qui
// n'utilise pas de source de données
. . .
```

## Obtenir une DataSource

- Le plus souvent, la source de données est enregistrée auprès d'un registre JNDI
- On l'obtient en donnant la clé du registre qui lui correspond :

```
DataSource source =
  (DataSource) new InitialContext()
    .lookup("MaSource");
```

- Ce code ne peut fonctionner que dans un environnement dans lequel fonctionne un registre JNDI (serveur d'applications le plus souvent)



## Configurer une DataSource

- Le plus souvent elle se configure par un fichier XML de configuration ; voir le manuel des logiciels que vous utilisez
- Exemple : fichier **`server.xml`** de Tomcat
- Elle peut aussi se configurer par programmation ; voir la javadoc de la classe qui implémente **`DataSource`**

# Exemple avec Tomcat : server.xml

```
<Context path="/BDTest"
         ...>
  <Resource
     name="jdbc/TestDB"
     auth="Container"
     type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/TestBD">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSource
Factory
      </value>
    </parameter>
```



# Exemple avec Tomcat : server.xml

```
  <!- Le plus grand nombre de connexions dans le
  pool. Mettre à 0 si pas de limite. -->
  <parameter>
    <name>maxActive</name>
    <value>100</value>
  </parameter>
etc…
```

- Voir la documentation Tomcat pour le reste du fichier server.xml

## Exemple avec Tomcat : web.xml

```
<?xml version="1.0"
      encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC ...>
<web-app>
  <description>TestDB</description>
  <resource-ref>
    <description>DB Connection</description>
    <res-ref-name>jdbc/TestBD</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```



## Exemple avec Tomcat – code servlet

```
import javax.naming.*;
import javax.sql.*;
...
Context context = new InitialContext();
DataSource ds = (DataSource)ctx.lookup(
  "jdbc/TestBD" );
Connection conn = ds.getConnection(
  "toto", "mdp" );
...
finally {
  if( conn != null ) { conn.close(); }
}
```

# Transactions JTA
# (*Java Transaction API*)



# Limite des transactions JDBC

- Elles ne permettent pas à une transaction de couvrir plusieurs connexions à des bases de données différentes
- Pourtant, l'existence de nombreuses bases de données héritées du passé conduisent parfois les applications d'entreprise à travailler avec des transactions qui couvrent plusieurs bases de données à la fois

## JTA

- Cette API fournit les interfaces pour travailler avec des transactions distribuées sur plusieurs bases de données, ou même des ressources qui ne sont pas des bases de données, d'une façon indépendante de l'implémentation du gestionnaire de transactions
- Elle est utilisée par les serveurs d'applications pour les transactions gérées par le container



## `Transaction` (1)

- Interface du paquetage **`javax.transaction`** qui décrit une transaction JTA
- Contient les méthodes **`commit()`**, **`rollback()`**
- La méthode **`setRollbackOnly()`** indique que la transaction ne pourra se terminer que par un rollback (utile quand le code ne gère pas directement le début et la fin de la transaction, par exemple quand on utilise un serveur d'applications)

## `Transaction` (2)

- L'interface contient aussi des méthodes pour associer (ou désassocier) des ressources à la transaction (le plus souvent des ressources liées à des SGBDs) : **`enlistResource`** et **`delistResource`**
- Ces ressources pourront participer à un commit à 2 phases (commit pour les transactions distribuées qui comprend 2 phases : la préparation et l'accord des ressources, puis le commit effectif)



## Références sur les JTA

- JTA n'est pas étudiée en détails dans ce cours
- 2 références :
  http://java.sun.com/products/jta/
  http://www.theserverside.com/articles/article.tss?l=Nuts-and-Bolts-of-Transaction-Processing