

Cours Bases de données 2ème année IUT

JDBC : ou comment lier ORACLE avec Java

Anne Vilnat

Plan

- 1 Introduction
- 2 Connexion
 - Mise en place du pilote
 - Nommer la base de données
 - Etablir la connexion
 - Dialogue avec la base de données
 - Deconnexion
 - Les Exceptions
- 3 Requêtes et Résultats
 - Statement
 - ResultSet
 - PreparedStatement
 - ResultSetMetadata
- 4 Procédures et fonctions stockées
 - CallableStatement
- 5 Conclusion

Introduction : le problème



Usage

JDBC pour exécuter, depuis un programme Java, l'ensemble des ordres SQL reconnus par la base de données cible.

La base de données doit reconnaître le langage ANSI SQL-2.

Introduction : JDBC



Définition

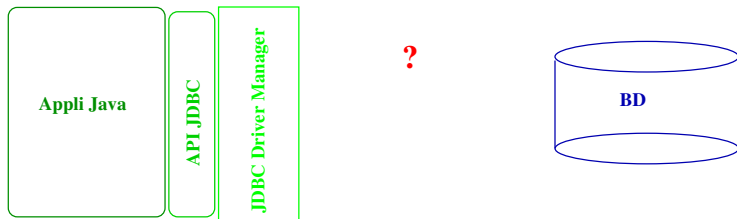
JDBC (*Java DataBase Connectivity*) est une API (*Application Programming Interface*) qui permet d'exécuter des instructions SQL.

JDBC fait partie du JDK (*Java Development Kit*).

Paquetage **java.sql** :

```
import java.sql.*;
```

Introduction : Gestion du pilote



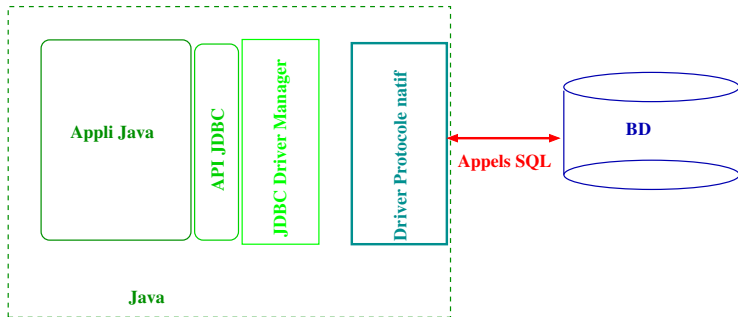
Le pilote...

Il va établir le lien avec la base de données, en sachant “lui parler”.
Dans JDBC : des classes chargées de gérer un pilote...

Pilote “récent” : en Java

Des pilotes existent pour mySQL, postGresSQL, ACCESS,...

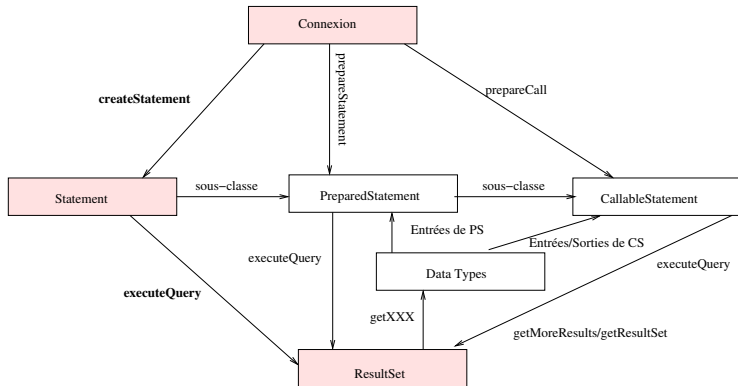
Introduction : Driver Manager



La connexion...

Elle peut s'établir SI on donne l'adresse de la BD à laquelle se connecter...

Les classes et interfaces du package java.sql



Fonctionnement

Etapes d'un programme utilisant JDBC :

- 1 mettre en place le pilote ou *driver*.
- 2 établir une connexion avec une source de données.
- 3 effectuer les requêtes.
- 4 utiliser les données obtenues pour des affichages, des traitements statistiques, etc.
- 5 mettre à jour les informations de la source des données.
- 6 terminer la connexion.
- 7 éventuellement, recommencer en 1.

Les étapes...

exemples

- 1 charger un pilote *driver*
`Class.forName("oracle.jdbc.driver.OracleDriver");`
- 2 créer un objet *Connection*
`Connection.maConnection=DriverManager.getConnection(url);`
url : `String` contenant l'adresse de la base de données
- 3 créer un objet *Statement*
`Statement.maRequeteSQL=maConnection();`
- 4 envoyer la requête et récupérer le résultat dans un *ResultSet*
`ResultSet.monResultat=`
`maRequeteSQL.executeQuery(texteRequeteSQL);`
texteRequeteSQL : `String` contenant le texte de la requête,
par exemple :
"SELECT * FROM Client"

Mise en place du pilote

2 méthodes :

Chargement statique

- enregistrer le ou les drivers(s) à utiliser
- à chaque connexion, passer comme argument l'url correspondante
- utiliser l'interface `java.sql.Driver` : écrire une classe `Driver`, pour créer une instance d'elle-même et l'enregistrer avec la méthode `DriverManager.registerDriver()`

pas la plus simple, ni la plus usitée...

Mise en place du pilote

Chargement dynamique

A la demande, sans noter explicitement le nom des classes :

```
try {  
    Class.forName( "oracle.jdbc.driver.OracleDriver" );  
}  
catch (Exception e){  
    System.out.println(" Impossible de charger le driver");  
    return;  
}
```

Des pilotes existent pour mySQL, postGresSQL, ACCESS,...

Nommage des bases de données

Dérivée des url d'internet.

Schéma général

jdbc:<sous-protocole>:<compléments>

jdbc = protocole

sous-protocole : pour distinguer le type de pilote jdbc `oracle:thin` à l'IUT

complements : la base de données. Syntaxe :

`login/motDePasse@ordinateur:port:base.`

Exemple : `toto/mdpToto@orasrv1.ens.iut-orsay.fr:1521:etudom`

Connexion

par la méthode getConnection de DriverManager :

Exemple

```
import java.net.*;
import java.sql.*;
String url=
    "jdbc:oracle:thin:toto/mdpToto@r2d2.iut-orsay.u-
    psud.fr:1521:etudom";
try {
    Class.forName( "oracle.jdbc.driver.OracleDriver" );
}
catch (Exception e){
    System.out.println(" Impossible de charger le driver");
    return;
}
Connection maConnexion=DriverManager.getConnection(url); }
```

Connexion

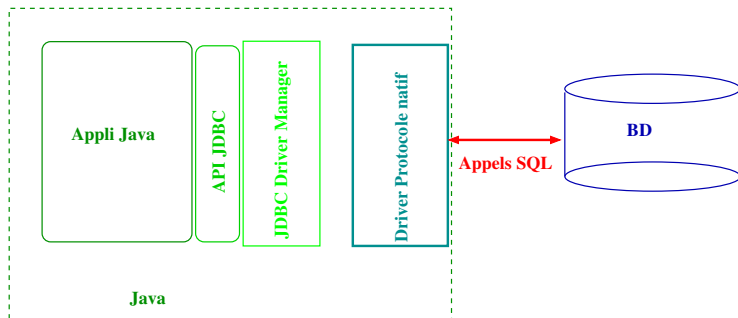
ou en utilisant les objets DataSource

Exemple

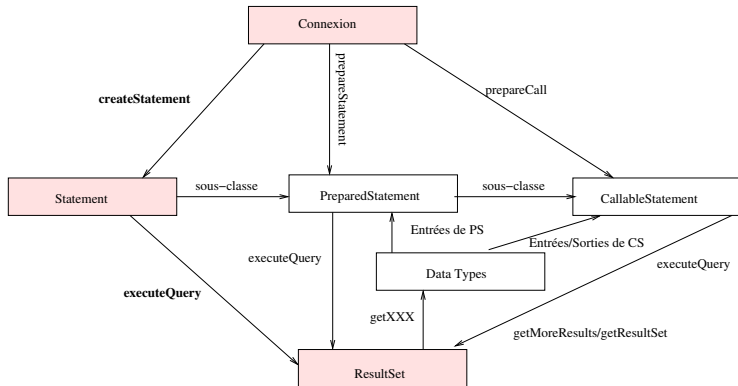
```
import java.sql.*;
import oracle.jdbc.pool.*;
public class TestDataSource {
    public static void main(String args[])
        throws ClassNotFoundException, SQLException {
        OracleDataSource ds = new OracleDataSource();
        ds.setDriverType('thin');
        ds.setServerName("r2d2");
        ds.setPortNumber(1521);
        ds.setDataBaseName("etudom");
        ds.setUser("toto");
        ds.setPassword("mdpToto");

        Connection maConnexion=DriverManager.getConnection(url); }
```

Les tuyaux sont en place...



Rappel : les classes et interfaces du package java.sql



Pour dialoguer : Statement

Exemple

```
Statement monInstruction = maConnexion.createStatement();
```

suivant l'instruction SQL

Instructions SQL	Méthode	Type retourné	Valeur retournée
SELECT	executeQuery	ResultSet	Lignes de résultat
UPDATE, INSERT,DELETE	executeUpdate	int	Nb lignes modifiées
Autres	execute	boolean	Faux si erreur

Consultation et récupération de données

Exemple

```
ResultSet monRésultat = monInstruction.executeQuery(  
    "SELECT login, nomClient FROM toto.Client");
```

ResultSet et ses méthodes

Résultat dans un **ResultSet**

Parcours analogue à celui d'un curseur avec la méthode **next**, et accès aux colonnes avec **getXXX**

Exemple de parcours

```
while (monResultat.next()) {  
    String nom = monResultat.getString("nomClient");  
    int login=monResultat.getInt("login");  
    // traitement des données récupérées  
}
```

Consultation et récupération de données

ResultSet et ses méthodes

Le premier `next` positionne sur la première ligne.

Paramètres de `getXXX` : nom de l'attribut ou rang dans la requête (sous forme d'entier). Obligatoire pour les attributs calculés (`MAX(...)`) ou quand les noms ne sont pas connus (`SELECT *...`)

Exemple de parcours

```
while (monResultat.next()) {  
    String nom = monResultat.getString(2);  
    int login=monResultat.getInt (1);  
    // traitement des données récupérées  
}
```

Les correspondances de types

Type SQL	Type Java
CHAR, VARCHAR2,	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	byte []
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Les valeurs NULL

Problème : reconnaître dans Java le cas d'une valeur **NULL**.

Conventions :

- Pour les méthodes `getString()`, `getObject()`, `getDate()`, ... : **Null** Java (il existe)
- Pour les méthodes `getInt()`, `getByte()`, `getShort()`, ... : la valeur **0** est renvoyée
- Pour la méthode `getBoolean()`, la valeur **Faux** est renvoyée.

MAIS pas correct pour reconnaître des valeurs non renseignées dans la base...

D'où la méthode `wasNull()` de `ResultSet`.

Fonctionnement :

- lire la donnée,
- tester avec `wasNull()` si elle vaut **NULL** au sens SQL

Accès et mise à jour

Pour INSERT, DELETE et UPDATE...

La classe `Statement` a : `executeUpdate()`

Elle retourne un `int` qui contient le nombre de lignes affectées par l'instruction.

Exemple

```
int nbLignes = monInstruction.executeUpdate(  
    "INSERT INTO toto.Client(login, nomClient)  
        VALUES (" + numero + "," + nom + ")");  
System.out.println(nbLignes + " ligne(s) insérée(s)");
```

Modification de la définition des données

Pour modifier la structure de la base

La classe `Statement` a : `execute(ordreSQL)`

L'ordre SQL correspond à la chaîne de caractères contenant l'ordre à exécuter

Elle retourne un `boolean` qui est vrai si il n'y a pas eu d'erreur à l'exécution..

Déconnexion

Libérer **ResultSet** et **Statement**, fermer la **Connection**

Exemple

```
monResultat.close() ;  
monInstruction.close() ;  
maConnexion.close() ;
```


La classe SQLException

Méthodes

`java.sql.SQLException` hérite de `java.sql.SQLException`.

Parmi les méthodes définies dans `java.lang.Exception`,:

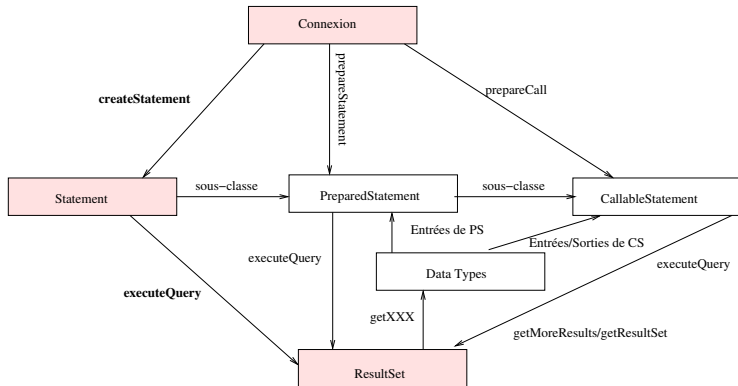
- `getSQLState()` qui renvoie la chaîne de caractères correspondant à SQLSTATE (le code d'erreur de la norme SQL),
- `getErrorCode()` qui renvoie l'entier correspondant au "code d'erreur vendeur" (le code d'erreur propre à l'éditeur de la base de données, donc non normalisé),
- `getNextException()` qui donne l'exception qui suit l'exception courante dans le chaînage des exceptions,
- `setNextException()` qui permet d'ajouter l'exception passée en paramètre au chaînage des exceptions. Cette méthode n'est normalement utilisée que par les développeurs de drivers.

La classe SQLException

Exemple

```
catch (SQLException ex) {  
    System.out.println (" Capture une SQLException :);  
    while (ex != null) {  
        System.out.print ("SQLSTATE: " + ex.getSQLState ( ));  
        System.out.print (" Message: " + ex.getMessage ( ));  
        System.out.println (" Code d'erreur vendeur: "  
                               + ex.getErrorCode ( ));  
        ex.printStackTrace (System.out)  
        ex = ex.getNextException ( );  
        System.out.println (" ");  
    }  
}
```

Les classes et interfaces du package java.sql



Principales interfaces Java de l'API JDBC

Généralités

Interfaces donc méthodes définies comme `public abstract`.
Susceptibles de lever des exceptions `SQLException`.

Les interfaces

Principales interfaces :

- `Connection`
- `Statement`
- `ResultSet`
- `ResultSetMetaData`

Les transactions

- Par défaut, [autocommit on](#).
- Pour gérer les transactions :
void setAutoCommit (boolean autoCommit)
throws SQLException;
void commit () throws SQLException;
void rollback () throws SQLException;
boolean getAutoCommit () throws SQLException;
- Pour clore :
void close () throws SQLException;
boolean isClosed () throws SQLException;
- Pour transcoder les chaînes de caractères :
String nativeSQL (String sql) throws SQLException;

Définition du Statement

Statement

- Définit les objets qui permettent d'exécuter les requêtes statiques SQL et de retourner le résultat produit.
- Donne le type et les propriétés du **ResultSet** qui lui sera associé
- Un seul **ResultSet** est actif à la fois. Si plusieurs, il faut plusieurs **Statement**

Créer un Statement

Les instructions

Pour créer différents `Statement` qui donneront des `ResultSet` ayant des propriétés différentes :

- `Statement createStatement () throws SQLException;`
- `Statement createStatement (int rsType, int rsConcurrency) throws SQLException;`
- `Statement createStatement (int rsType, int rsConcurrency, int rsHoldability) throws SQLException;`

permet de définir

- le type du `ResultSet` (`rsType`), sa “navigabilité”
- le fait qu’il permette ou non des mises à jour (`rsConcurrency`)
- son comportement lors d’un commit (`rsHoldability`)

Principales méthodes sur un Statement

Méthodes

Principales méthodes :

- Pour créer un ResultSet contenant les résultats d'une requête :
`ResultSet executeQuery (String sql) throws SQLException;`
- `void close () throws SQLException;`
- `int executeUpdate (String sql) throws SQLException;`
- `boolean execute (String sql) throws SQLException;`

Les ResultSet : quelles possibilités?

Différents types de ResultSet

- Navigabilité :
 - `TYPE_FORWARD_ONLY` (défaut) :
navigation “en avant” uniquement;
 - `TYPE_SCROLL_INSENSITIVE` :
dans tous les sens, et où on veut, mais pas d'accès aux modifications sur la source de données depuis l'ouverture du `ResultSet`;
 - `TYPE_SCROLL_SENSITIVE` :
navigation + accès aux modifications;

Les ResultSet : quelles possibilités?

Différents types de ResultSet (suite)

- Mise à jour :
 - **CONCUR_READ_ONLY** (défaut) :
pas de modification
 - **CONCUR_UPDATABLE** :
modifications possibles
- Comportement à la validation :
 - **HOLD_CURSORS_OVER_COMMIT** :
reste ouvert lors d'une validation (COMMIT ou ROLLBACK)
 - **CLOSE_CURSORS_AT_COMMIT** (défaut) :
fermé après chaque validation.

Exemple de ResultSet

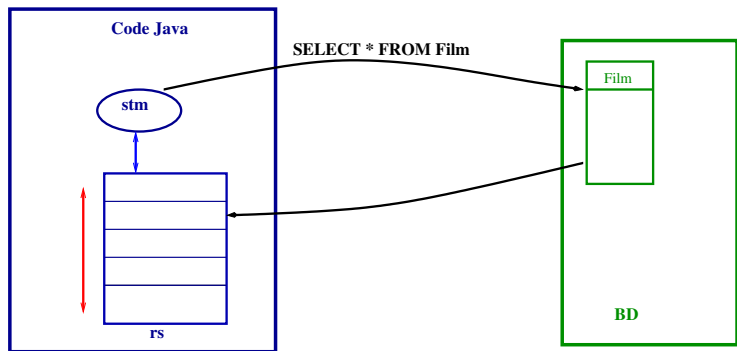
Pour pouvoir :

- parcourir le résultat de la requête dans n'importe quel sens, sans avoir accès à d'éventuelles modifications dans les données sur la base,
- sans pouvoir modifier les éléments au travers du ResultSet,
- et fermer le ResultSet si un commit a lieu :

Exemple

```
Statement stm =  
    co.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
                        ResultSet.CONCUR_READ_ONLY,  
                        ResultSet.CLOSE_CURSOR_AT_COMMIT);  
ResultSet rs=stm.executeQuery( "SELECT * FROM Film");
```

Exemple de ResultSet



Mouvements dans un ResultSet (1)

Les méthodes

Lors de la création : pointe “avant” la première ligne.

Si de type `TYPE_FORWARD_ONLY`, que :

- `next()` : passe à la ligne suivante. retourne `true` si elle existe, `false` sinon (après la dernière ligne).

sinon :

- `previous()` : ligne précédente
- `first()` : sur la première ligne, retourne `true` si elle existe, `false` sinon (resultSet vide)
- `last()` : sur la dernière;
- `beforeFirst()` : avant la première (comme à l'ouverture)
- `afterLast ()` : après la dernière

Mouvements dans un ResultSet (2)

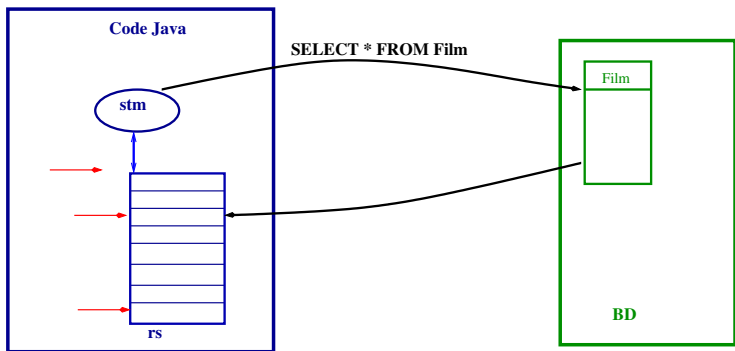
Les méthodes (suite)

- `relative(int rows)` : `rows` lignes après la position courante, revient en arrière si `rows` est négatif;
- `absolute(int row)` : se place à la `row`-ième ligne. Si `row` est égal à 1 : sur la première, si `row` est négatif, sur la dernière. (si 0, sur la première aussi)

Exemple

```
Statement stm =  
    co.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE);  
ResultSet rs=stm.executeQuery( "SELECT * FROM Film");  
rs.absolute(3); // positionne sur la 3ème ligne;  
rs.relative(5); // positionne sur la 8ème ligne
```

Exemple de navigation dans un ResultSet



Modifications d'un ResultSet : mise à jour d'une ligne

Définition

Si ouvert en mode `CONCUR_UPDATABLE`, permet la mise à jour de la base par le biais du `ResultSet` (sinon on le fera par `execute` ou `executeUpdate`)

Mise à jour d'une ligne

En 2 étapes :

- mise à jour de la nouvelle valeur de la colonne : `updateXXX`
- changements affectés à la ligne concernée (alors seulement la base sera mise à jour) : `updateRow()`

Modifications d'un ResultSet : Exemple de mise à jour

Exemple

```
Statement stm =  
    co.createStatement(ResultSet.TYPE_FORWARD_ONLY,  
                        ResultSet.CONCUR_UPDATABLE);  
    "SELECT Titre FROM Film  
    WHERE NumFilm=123");  
rs.next(); // positionne sur la 1ère ligne;  
rs.updateString( "Titre", "Charlie et la chocolaterie");  
//modifie l'attribut Titre  
rs.updateRow(); // effectue la modification de la ligne
```

Mise à jour avec un ResultSet

Exemple

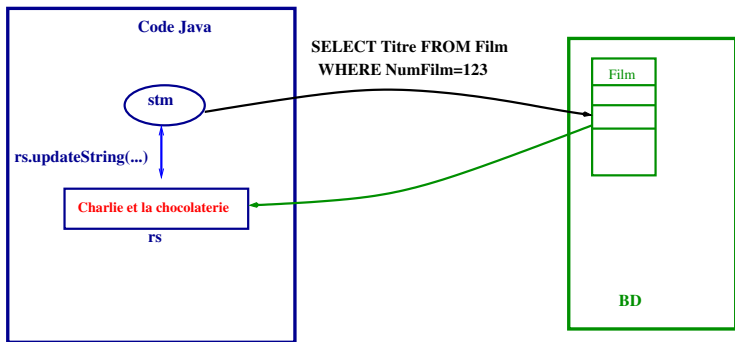
```
Statement stm = co.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=stm.executeQuery("SELECT Titre FROM Film WHERE NumFilm=123");  
rs.next();
```



Mise à jour avec un ResultSet

Exemple

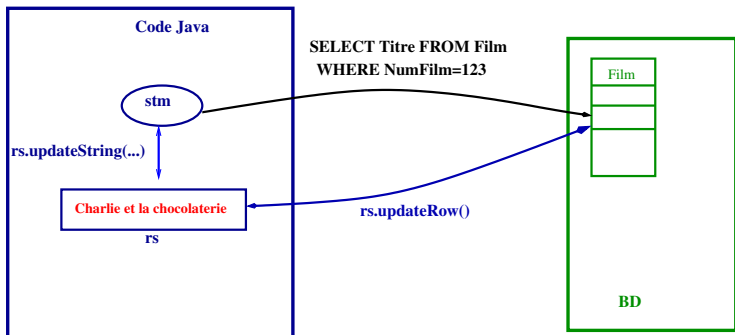
```
Statement stm = co.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=stm.executeQuery("SELECT Titre FROM Film WHERE NumFilm=123");  
rs.next();  
rs.updateString("Titre", "Charlie et la chocolaterie");
```



Mise à jour avec un ResultSet

Exemple

```
Statement stm = co.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=stm.executeQuery( "SELECT Titre FROM Film WHERE NumFilm=123" );  
rs.next();  
rs.updateString("Titre", "Charlie et la chocolaterie");  
rs.updateRow();
```



Modifications d'un ResultSet : suppression d'une ligne

Suppression d'une ligne

Si ouvert en mode `CONCUR_UPDATABLE`, permet la suppression de la ligne dans la base : `deleteRow()`. Se place ensuite avant la première ligne valide suivant celle qui vient d'être supprimée. Son indice devient invalide.

→ test de l'existence d'une ligne par `rowDeleted`

Exemple

```
Statement stm =  
    co.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,;  
                        ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=stm.executeQuery( "SELECT * FROM Film");  
rs.absolute(4); // positionne sur la 4ème ligne;  
rs.deleteRow(); //supprime la ligne  
bool =rs.rowDeleted(); // bool vaut vrai
```

Modifications d'un ResultSet : insertion d'une ligne

Définition

Si ouvert en mode `CONCUR_UPDATABLE`, permet l'insertion de nouvelles lignes

Insertion d'une ligne

En 3 étapes :

- se positionner sur la ligne d'insertion : `moveToInsertRow()`;
- initialiser les valeurs des champs de la ligne à insérer :
`updateXXX`,
- insérer la ligne : `insertRow()`

Modifications d'un ResultSet : Exemple d'insertion

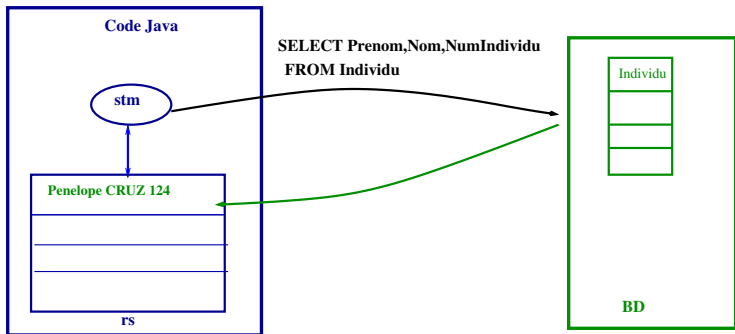
Exemple

```
Statement stm =  
    co.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,;  
                        ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=stm.executeQuery(  
    "SELECT Prenom, Nom, NumIndividu FROM  
Individu");  
rs.moveToInsertRow(); // positionne sur une ligne d'insertion;  
rs.updateString(1, "Pedro"); // crée le prénom  
rs.updateString(2, "Almodovar"); // crée le nom  
rs.updateLong(3, 278866500); // crée le num...  
rs.insertRow(); // insère la ligne  
rs.moveToCurrentRow(); // revient à l'ancienne position
```

Modifications d'un ResultSet : Exemple d'insertion

Exemple

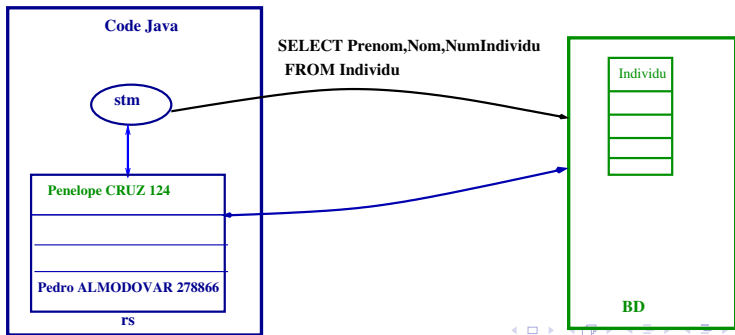
```
Statement stm = co.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=stm.executeQuery("SELECT Prenom, Nom, NumIndividu FROM Individu");  
rs.moveToInsertRow();
```



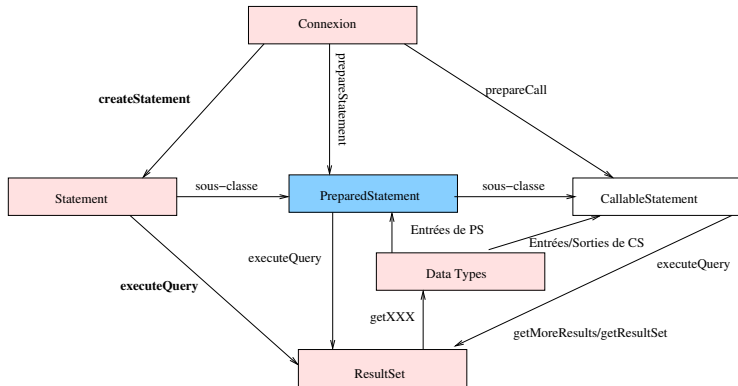
Modifications d'un ResultSet : Exemple d'insertion

Exemple

```
Statement stm = co.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=stm.executeQuery("SELECT Prenom, Nom, NumIndividu FROM Individu");  
rs.moveToInsertRow();  
rs.updateString(1, "Pedro");  
rs.updateString(2, "Almodovar");  
rs.updateLong(3, 278866500);  
rs.insertRow();  
rs.moveToCurrentRow();
```



Les requêtes pré-compilées



Requêtes pré-compilées

Motivation

Si on doit répéter plusieurs fois la même requête en utilisant un objet `Statement`, à chaque fois le serveur devra interpréter la requête SQL et en particulier créer un plan de requête :

- → particulièrement coûteux.
- → utilisation de l'interface `PreparedStatement`.

Le code SQL est transmis à la base une seule fois, dès que la méthode `prepareStatement()`, issue de la classe `Connection` délivre l'objet `PreparedStatement` correspondant.

Requêtes pré-compilées : PreparedStatement

Exemple

```
PreparedStatement psm = myconnexion.prepareStatement (
    "SELECT nom_emp FROM employe
     WHERE prime > 999
     AND service = 'Comptabilite'
     AND salaire > 2499 ");
for (int i = 0; i < 10; i++) {
    ResultSet myresultat = psm.executeQuery ( );
    while (myresultat.next ( )) {
        // traitement des donnees recupérées
    }
}
```

Pas forcément très intéressant!...

Requêtes pré-compilées : PreparedStatement (2)

Avec des paramètres symbolisés par ? et `setXXX`

Exemple

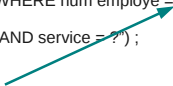
```
PreparedStatement psm = myconnexion.prepareStatement (
    "UPDATE employe SET prime = prime + ?
    WHERE num_employe = ?
    AND service = ?");

while (suite ( )) {
    psm.setInt (2, saisie_numero_employe ( ));
    psm.setInt (1, saisie_prime ( ));
    psm.setString (3, saisie_service ( ));
    int myresultat = psm.executeUpdate ( );
}
```

Avec : `saisie_numero_employe`, `saisie_prime`, `saisie_service` sont des méthodes, et `suite` est un booléen.

Les requêtes pré-compilées

```
PreparedStatement psm = myconnexion.prepareStatement (
    "UPDATE employe SET prime = prime + ?" (1)
    WHERE num employe = ? (2)
    AND service = ?" ); (3)
while (suite ( )) {
    psm.setInt (2, saisie numero employe ( )) ;
    ...
}
```



Les requêtes pré-compilées

```
PreparedStatement psm = myconnexion.prepareStatement (
    "UPDATE employe SET prime = prime + ?" (1)
    WHERE num employe = ? (2)
    AND service = ?" ); (3)

while (suite ( )) {
    psm.setInt (2, saisie_numero employe ( ));
    psm.setInt (1, saisie_prime ( ));
    ...
}
```

Les requêtes pré-compilées

```
PreparedStatement psm = myconnexion.prepareStatement (
    "UPDATE employe SET prime = prime + ?" (1)
    WHERE num employe = ? (2)
    AND service = ?" ); (3)

while (suite ( )) {
    psm.setInt (2, saisie numero employe ( ));
    psm.setInt (1, saisie prime ( ));
    psm.setString (3, saisie service ( ));
    int myresultat = psm.executeUpdate ( );
}
```


Meta données sur le ResultSet : ResultSetMetadata

Interface pour obtenir des informations supplémentaires sur la structure d'un [ResultSet](#). Pour répondre à :

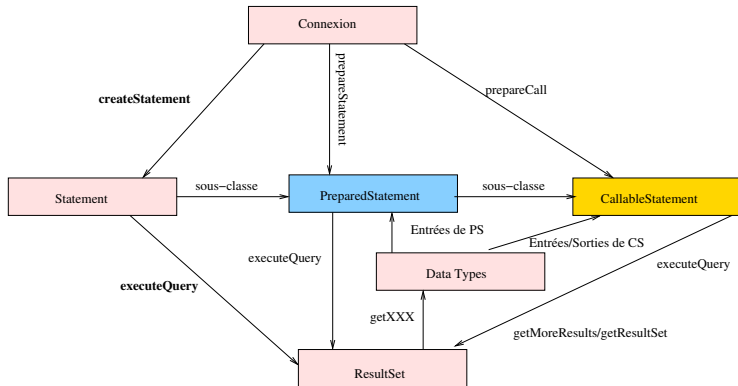
- Combien de colonnes le [ResultSet](#) contient-il ?
- Les noms de ces colonnes sont-ils sensibles à la casse ?
- Est-il possible de rechercher des données dans la colonne de son choix ?
- Est-il possible d'affecter NULL à une colonne donnée ?
- Quel est le nombre maximal de caractères affichables pour une colonne donnée ?
- Quel libellé faut-il attribuer à la colonne choisie lors de l'affichage ?
- Quel est le nom de la colonne choisie ?
- De quelle table la colonne choisie provient-elle ?
- Quel type de données la colonne choisie renferme-t-elle ?

ResultSetMetadata : Exemple

Exemple

```
Connection myconnexion = DriverManager.getConnection(url);
Statement stmt = myconnexion.createStatement ( );
ResultSet rs = stmt.executeQuery (
    "SELECT a,b,c FROM Table1");
ResultSetMetaData rsmd = rs.getMetaData ( );
int numberOfColumns = rsmd.getColumnCount ( );
for (int i = 1; i <= numberOfColumns; i++) {
    int jdbcType = rsmd.getColumnType (i);
    String name = rsmd.getColumnTypeName (i);
    System.out.print ("L'attribut " + i +
        "est du type JDBC " + jdbcType);
    System.out.println ("dont le nom de type JDBC est " + name);
}
```

Procédures et fonctions stockées



Appel aux procédures et aux fonctions stockées

Motivation

- L'interface `CallableStatement` permet d'appeler des procédures ou des fonctions stockées.
- On indique le nom de la procédure ou de la fonction requise lors de l'initialisation de l'objet `CallableStatement` grâce à la méthode `prepareCall()` de l'interface `Connection`.
- Deux formes possibles, selon que la procédure ou la fonction stockée comporte des paramètres lors de l'appel ou n'en comporte pas

Appel aux procédures et aux fonctions stockées

Comment?

- sans paramètre :

```
CallableStatement cst = myconnexion prepareCall (
    "{call nom_procedure}" );
```

- avec des paramètres :

```
CallableStatement cst = myconnexion prepareCall (
    "{? = call nom_fonction ( ? , ? , ... )}");
```

ou :

```
CallableStatement cst = myconnexion prepareCall (
    "{call nom_procedure ( ? , ? , ... )}");
```

CallableStatement : Exemple

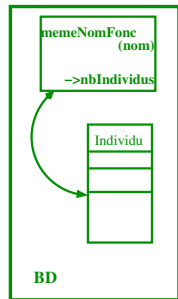
Exemple d'une fonction stockée

On cherche le nombre d'Individus portant le même nom

```
CREATE FUNCTION memeNomFonc (nom
Individu.nomIndividu%type) RETURN NUMBER IS
    nbIndividus NUMBER;
BEGIN
    SELECT COUNT(*) INTO nbIndividus
    FROM individu
    WHERE nomIndividu = nom
    GROUP BY nomIndividu
    RETURN Individus;
END;
```

Exemple d'une fonction appelée

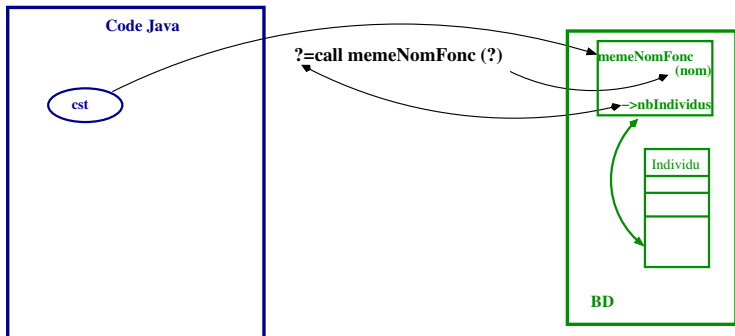
Code Java



Exemple appel à une fonction stockée

Comment?

```
CallableStatement cst = myconnexion prepareCall (  
    (" {? = call memeNomFonc ( ? ) }");
```



Appel aux procédures et aux fonctions stockées : paramètres

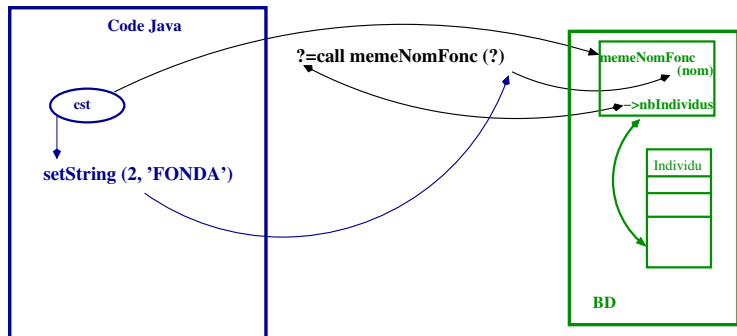
Paramètres en entrée

- Pour positionner les paramètres effectifs d'entrée (IN ou IN OUT):
`setXXX()`, avant le : `execute()`
où `XXX` est un nom de type Java.
- Paramètres de gauche à droite dans l'ordre d'apparition dans l'instruction SQL.
- `setXXX()` a deux paramètres : le rang du paramètre à positionner et la valeur transmise au paramètre de la fonction ou de la procédure.

CallableStatement : Exemple

Exemple d'une fonction stockée : les paramètres en entrée

- le nom en entrée : setString
- `cst.setString (2, 'FONDA');`



Appel aux procédures et aux fonctions stockées : paramètres (suite)

Paramètres en sortie

- Paramètres de sortie (OUT ou IN OUT): récupérés après le `execute()`, par `getXXX()`, où `XXX` est un nom de type Java.
- `getXXX()` a un paramètre : le rang du paramètre à récupérer.

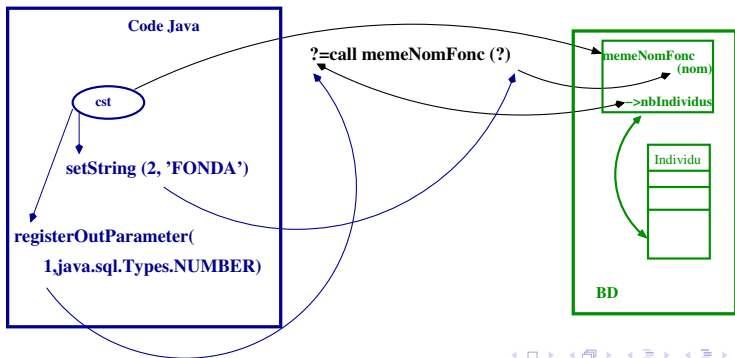
Typage des paramètres

Lorsqu'une fonction ou une procédure stockée renvoie une valeur (valeur de retour, paramètre OUT ou paramètre IN OUT), JDBC exige d'en spécifier le type:
`registerOutParameter()` avant exécution de la procédure.

CallableStatement : Exemple

Exemple d'une fonction stockée : les paramètres en sortie

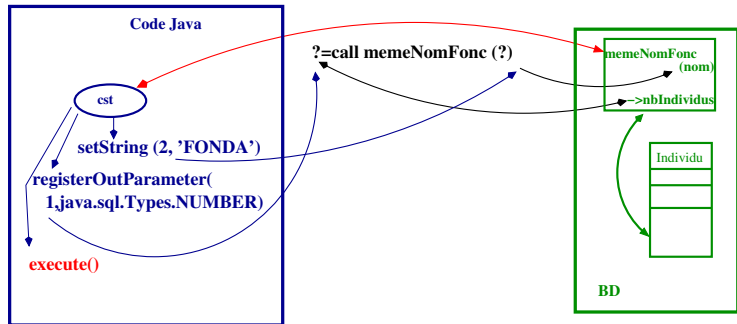
- typage du résultat : un NUMBER
- `cst.registerOutParameter (1,java.sql.Types.NUMBER);`



CallableStatement : Exemple

Exemple d'une fonction stockée : Exécution de la fonction!

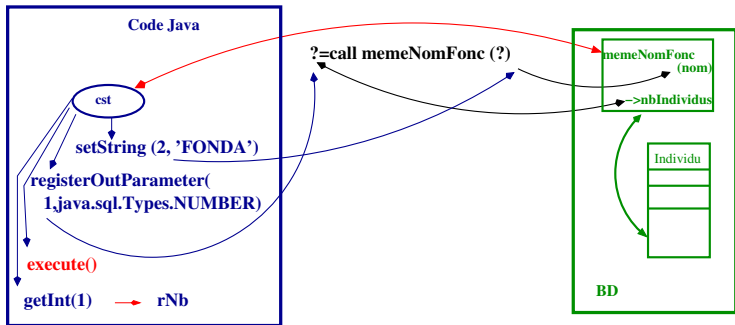
- execute!
- boolean succes = cst.execute ();



CallableStatement : Exemple

Exemple d'une fonction stockée : et on récupère le résultat

■ `int rNb = cst.getInt(1);`



CallableStatement : Exemple

Exemple d'une fonction stockée : et on récapitule!

```
CallableStatement cst = myconnexion prepareCall  
    (" {? = call memeNomFonc ( ? ) }");  
cst.setString (2,'FONDA');  
cst.registerOutParameter (1,java.sql.Types.NUMBER);  
  
boolean succes = cst.execute ( );  
  
int rNb = cst.getInt (1);  
cst.close ( );
```

CallableStatement : Exemple

Exemple d'une procédure stockée plus complexe

Le nom, le prenom du réalisateur et le numfilm d'un film...

côté PLSQL

On a une procédure DetailsFilm qui prend en paramètre :

- en entrée : le titre du film,
- en sortie : le numfilm, les nom et prénom du réalisateur

```
CREATE PROCEDURE DetailsFilm (  
    titre IN Film.titre%type,  
    nomReal OUT Individu.NomIndividu%type,  
    prenomReal OUT Individu.PrenomIndividu%type,  
    num OUT Film.NumFilm%type)
```


CallableStatement : Exemple

Code jdbc

```
CallableStatement cst = myconnexion prepareCall  
                                (" {call DetailsFilm ( ? , ? , ?, ? ) }");
```

on établit le lien avec la procédure PL/SQL, et avec ses 4 paramètres

....

CallableStatement : Exemple

Code jdbc

```
CallableStatement cst = myconnexion prepareCall  
    (" {call DetailsFilm ( ? , ? , ?, ? ) }");  
cst.setString (1,'CHARLIE ET LA CHOCOLATERIE');  
on donne le type et la valeur du paramètre en entrée (le titre)
```

. . .

CallableStatement : Exemple

Code jdbc

```
CallableStatement cst = myconnexion prepareCall  
    (" {call DetailsFilm ( ? , ? , ?, ? ) }");  
cst.setString (1,'CHARLIE ET LA CHOCOLATERIE');  
cst.registerOutParameter (2,java.sql.Types.VARCHAR2);  
cst.registerOutParameter (3,java.sql.Types.VARCHAR2);  
cst.registerOutParameter (4,java.sql.Types.NUMBER);  
on donne le type des trois paramètres en sortie : nom, prénom et  
numFilm
```

. . .

CallableStatement : Exemple

Code jdbc

```
CallableStatement cst = myconnexion prepareCall
    (" {call DetailsFilm ( ? , ? , ? , ? ) }");
cst.setString (1,'CHARLIE ET LA CHOCOLATERIE');
cst.registerOutParameter (2,java.sql.Types.VARCHAR2);
cst.registerOutParameter (3,java.sql.Types.VARCHAR2);
cst.registerOutParameter (4,java.sql.Types.NUMBER);

boolean succes = cst.execute ( );
on exécute la procédure, en vérifiant au besoin qu'elle s'est bien
exécutée

. . .
```

CallableStatement : Exemple

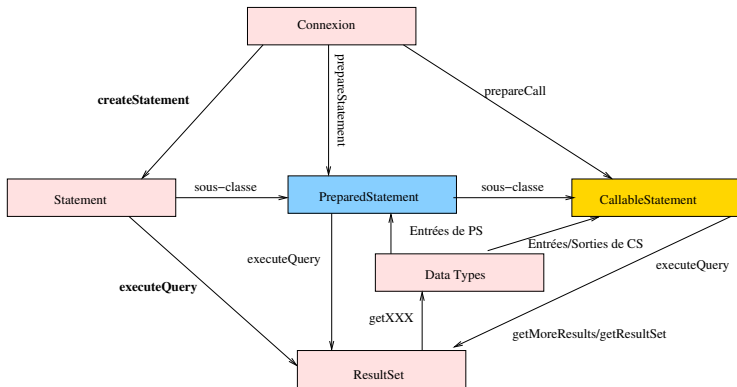
Code jdbc

```
CallableStatement cst = myconnexion prepareCall
    (" {call DetailsFilm (? , ? , ?, ?)}");
cst.setString (1,'CHARLIE ET LA CHOCOLATERIE');
cst.registerOutParameter (2,java.sql.Types.VARCHAR2);
cst.registerOutParameter (3,java.sql.Types.VARCHAR2);
cst.registerOutParameter (4,java.sql.Types.NUMBER);

boolean succes = cst.execute ( );

String rNom = cst.getString (2);
String rPrenom = cst.getString (3);
int rNumFilm = cst.getInt (4);
on récupère les valeurs calculées dans les variables...
. . .
cst.close ( );
```

Les classes et interfaces du package java.sql



Bilan PL/SQL

Quels ajouts

- des fonctions (stockées)
- des procédures (stockées)
- des paquetages pour regrouper
- des triggers pour de nouvelles formes de contrôle

Quels avantages

- plus rapide
- plus efficace
- mieux structuré

Quelles possibilités?

- établir une connexion par le biais d'un driver
- attacher des requêtes à des instructions (**Statement**)
 - requêtes simples
 - appel à des fonctions ou des procédures
- transmettre ces requêtes pour les faire exécuter sur le serveur
- récupérer les résultats dans des (**ResultSet**), ou dans des variables (**appel à des fonctions ou à des procédures stockées**)
- fermer la connexion

Comment choisir?

Qui doit faire quoi?

Faire les traitements sur le serveur (donc en PL/SQL) ou sur le client (donc en Java)?

- Règle générale : préférer les traitements en PL/SQL, surtout quand ils nécessitent des consultations multiples de la base
 - pour limiter les transferts entre la base (serveur) et le client éventuellement distant
 - pour profiter mieux des optimisations d'Oracle
 - pour profiter de l'efficacité des fonctions/procédures stockées déjà compilées/optimisées en PL/SQL
- Mais profiter des plus grandes souplesses de Java pour gérer des structures de données complexes, des interfaces,...