Base de données J2EE

Wurtz Jean-Marie

Université Louis Pasteur
Wurtz@igbmc.u-strasbg.fr

Références

- Java in a nutshell : A Desktop reference, David Flanagan, 4th Edition, O'Reilly, ISBN : 0-596-00283-1
- Java Entreprise in a nutshell: Manuel de référence pour Java 2, David Flanagan, Jim Farley, William Crawford & Chris Magnusson, O'Reilly, ISBN : 0-596-00283-1
- The J2EE Tutorial, Stéphanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, Beth Searms, Sun Edition, ISBN :0-201-79168-4

Sites web

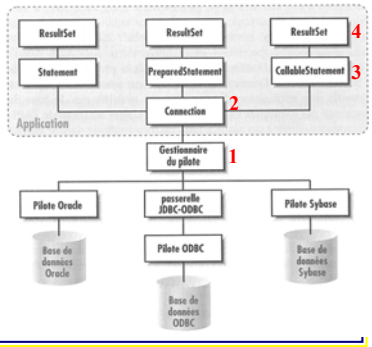
- www.sdv.fr/pages/casa/html/j2ee-os.html :
  Outils J2EE Open Source
- www.javaworld.com/javaworld/jw-09-2001/jw-0928-rup_p.html
  Step into the J2EE architecture and process
- java.sun.com/docs/books/j2eetutorial/index.html :
  Tutorial J2EE Sun
- java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html :
  Documentation J2EE sun

Les APIs J2EE : JBDC

- Pour Java Database Connectivity, une API permettant :
  - travailler avec des bases de données relationnelles
  - d'envoyer des requêtes SQL à une base
  - de récupérer et d'exploiter le résultat
  - d'obtenir des informations sur la base elle même et les tables
- Le code Java l'utilisant :
  - est indépendant de la BD, grâce à l'utilisation de drivers spécifiques fournis par les vendeurs
  - les requêtes JDBC doivent être standards (pas de fonctionnalités spécifiques) afin de rester portable.
- Il est fournie en standard avec le JDK depuis la v 1.1 de Java.
  - Les versions ultérieures fournissent de nouvelles fonctionnalités (comme la manipulation de résultats de requêtes comme des Java Beans, la gestion de pools de connections, les traitements par batchs ou la sérialisation d'objets Java en base).

L'architecture JDBC

- Les BD peu de choses commun
- Sauf le langage d'interrogation
- Sous-ensemble de SQI (SQL-92)
- Ensemble d'interfaces
- Pilote implémente ces interfaces
- JDBC les étapes :



Exemple JDBC

```
import java.sql.*;
public class JDBCSample {
  public static void main(java.lang.String[] args) {
    try {
      // chargement du driver ODBC de microsoft, le nom du paquetage
      //est donnée. Chargement dynamique de JdbcOdbcDriver.class
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch (ClassNotFoundException e) {
          System.out.println("Unable to load Driver Class");
          return;
    }
  ... // suite de la classe
```

Exemple (suite)

```
... // suite de la classe
try {
    // connexion à la base « nom de la base », nom de l'utilisateur, le mot de passe
    Connection con =
          DriverManager.getConnection("jdbc:odbc:companydb",  "", "");
    // création et exécution d'une instruction SQL
    Statement stmt = con.createStatement();
    //instruction SQL, résultat stocké dans un ResultSet
    ResultSet rs =
         stmt.executeQuery("SELECT FIRST_NAME FROM EMPLOYEES");
    // Affichage des résultats
    while(rs.next()) {
         System.out.println(rs.getString("FIRST_NAME"));
    }
    // restitution des ressources de la base
    rs.close();
    stmt.close();
    con.close();
  }
... // suite de la classe
```


Exemple (fin)

```
... // suite de la classe
    // restitution des ressources de la base
         rs.close();
         stmt.close();
         con.close();
    }
    catch (SQLException se) {
         // Informe l'utilisateur de toute erreur SQL
         System.out.println("SQL Exception: " + se.getMessage());
         se.printStackTrace(System.out);
    }
  }
}
```


Les pilotes JDBC

- Pour se servir d'un pilote : l'enregistrer auprès du gestionnaire JDBC DriverManager
- Méthode 1 : Cela se fait par l'appel à Class.forNam() :

```
try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Class.forName("com.oracle.jdbc.OracleDriver");
}
catch (ClassNotFoundException)
      //gestion de l'exception
}
```

- Méthode 2 : ajouter les classes des pilotes à la propriété jdbc.driver
  - Ajout à ~/.hotjava/properties la ligne suivante :

```
jbdc.driver =
    com.oracle.jdbc.OracleDriver:sun.jdbc.odbc.JdbcOdbcDriver:com.al.AlDriver
```

  - Chaque utilisateur doit posséder les classes pilotes dans son fichier propriété

Les types de pilotes

- Il existe des pilotes pour la plupart des bases de données
- Les pilotes sont classés en catégorie :
  1. pilotes passerelles : connecte un client Java à une base de données ODBC (Sun et InterSolv). Nécessite des logiciels non-Java et sont implémentés en code natif
  2. pilotes d'API natives partiellement en Java : fine couche Java et une bibliothèque en code natif. Ex : OCI de Oracle (Oracle Call Interface écrit en C/C++). Performance
  3. Pilotes entièrement écrit en Java incorporant un protocole réseau : utiliser par des applets les classes JDBC entièrement écrites en Java. Besoin d'un logiciel tiers (WebLogic de BEA).
  4. Les pilotes entièrement en Java implémentant un protocole natif : entièrement écrit en Java. Accepte les protocoles réseaux spécifiques de la base de données et accèdent directement la BD. Adaptés pour l'écriture d'applets.

Critères de choix du pilote

- La vitesse, la fiabilité et la portabilité
- Application exécutée exclusivement sur Windows NT : choix du type 2 pour des raisons de performances
- Un pilote de type 3 peut-être utilisé pour permettre de franchir un pare-feu dans une applet
- Déployée une servlet sur plusieurs plateformes nécessite un pilote de type 4
- Liste des JDBC : java.sun.com/products/jdbc/jdbc.drivers.html

Les URLs JDBC

- Identification de la base de données pour se connecter
  DriverManager.getConnection("jdbc:odbc:companydb", "", "");
- Syntaxe : jdbc:pilote:nom_de_la_base
- La norme est très floue, exemple de pilotes :
  - Oracle JBDC Thin : jdbc:oracle:thin@site:port:base_de_données
  - Passerelle JDBC-ODBC : jdbc:odbc:source_des_données;options_odbc);
- La contrainte : que le pilote reconnaisse ses propres URL
- Le pilote passerelle JDBC-ODBC est livré avec le JDK 1.1 et Java 2 SDK pour windows et Solaris.

Pilote passerelle JDBC-ODBC

- Elle installe une interface entre JDBC et les pilotes de bases de données écrits avec l'API ODBC de Microsoft (Open Database Connectivity)
- Elle n'est pas un composant obligatoire de Java SDK et donc n'est pas supporté par les navigateurs
- Permet à la communauté de Java de se former et de disposer rapidement d'un JDBC opérationnel
- Utilise des méthodes natives, ne pas utiliser pour une application

Connexion à la base de données

- Une application peut établir des connections multiples (limite imposée par la BD)
- Ex : BD Oracle standard accepte environ 50 connexions alors qu'une BD d'une grande société peut en accepter plusieurs milliers
- Une connexion est encapsulée dans un objet java.sql.Connection
- Créer une connexion avec :
  Connection con =
  DriverManager.getConnection("url", "utilisateur", "mot de passe")
- Le DriverManager va interroger successivement tous les pilotes enregistrés en présentant l'URL
- Si le pilote reconnaît l'URL, il renvoie un objet Connection
- Des variantes pour la méthode getConnection
  - sans nom d'utilisateur et sans mot de passe.
  - Définition dans un objet java.utilsProperties qui contient les couples nom/valeurs (username=valeur et password=valeur)

Une fois la connexion établie

- Exécution des instructions SQL grâce à des objets Statement
- JDBC propose 3 types d'instructions :*
  - Statement : instruction SQL ordinaire
  - PreparedStatement : instruction SQL compilée, plus efficace
  - CallableStatement : donneaux programme JDBC un accès sans restriction aux procédures stockées dans la BD

```
Statement stmt = con.createStatement();
ResultSet  rs = stmt.executeQuery("SELECT * FROM CLIENTS");
```

  - L'objet Statement possède aussi une méthode executeUpdate() pour les instructions SQL qui ne renvoient pas de résultats (DELETE et UPDATE). Renvoi d'un entier indiquant le nombre de lignes de la BD modifiées

Dans le doute pour les instructions SQL

- Un objet Statement à aussi une méthode execute()
- Execute() renvoie true si un résultat a été généré (un objet ResultSet)
- Récupérer :
  - le résultat par un appel à getResultSet()
  - Le nombre de lignes mises à jour par la méthode getUpDateCount()

```
Statement SQLanonyme = conn.createStatement();
if (SQLanonyme.execute(sqlString)) {
        ResultSet rs = SQLanonyme.getResultSet();
        // affichage
} else {
        System.out.println("Lignes mises à jour : "  +
                SQLanonyme. getUpDateCount());
}
```

Renvoi de plusieurs résultats

- Une instruction SQL peut renvoyer plus d'un ResultSet
- Un objet Statement supporte cette fonctionnalité avec la méthode getMoreResults()

```
Statement SQLanonyme = con.createStatement();
SQLanonyme.execute(sqlString);
While (true)  {
   rs = SQLanonyme.getResultSet();
   if (rs != null)
            // afficher les résultats
   else
            // traîter les données de mise à jour
   // passer au jeu de résultt suivant
   if ( SQLanonyme.getMoreResults() == false) &&
            SQLanonyme.getUpdateCount() == -1) ) break;
}
```

Les résultats d'une requête SQL

- Résultat sous forme de table :

NOM	NUM_CLIENT	TELEPHONE
Jane Markham	1	617 555-1212
Louis Smith	2	617 555-1213
Woodroow Lang	3	508 555-7171
Dr John Smith	4	(011) 42 323-123

- JDBC stocke ce résultat dans un objet java.sql.ResultSet
- Un tel objet représente une table contenant le résultat de la requête
- On peut lire successivement chaque ligne

Les résultats d'une requête SQL (2)

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
    "SELECT NOM, NUM_CLIENT, TELEPHONE FROM CLIENT");
While ( rs.next() ) {
    System.out.print(" Client # "+rs.getString("NUM_CLIENT"));
    System.out.print(", " )+rs.getString("NOM"));
    System.out.print(", téléphone " )+rs.getString("TELEPHONE"));
}
rs.close()
stmt.close();
```

Résultat :
Client #1, Jane Markham, téléphone 617 555-1212
Client # 2, Louis Smith, téléphone 617 555-1213
Client # 3, Woodroow Lang, téléphone 508 555-7171
Client #4, Dr John Smith, téléphone (011) 42 323-123

Les méthodes getXXX()

- Spécialisées par type de données :
  - CHAR String getString()
  - NUMERIC java.math.BigDecimal getBigDecimal()
  - BIT Boolean getBoolean
  - FLOAT Double (double) getDouble()
  - BINARY Byte[] GetBytes()
  - TIME java.sql.time getTime()
- 2 variantes :
  - Avec chaîne de caractères getString("CLIENT ") ou getString("Client ")
  - Avec l'index de colonne de la table dans le style SQL (de 1 à n et java de 0 à n-1, n le nb de colonnes)
- Une méthode particulière : getObject()
  - Renvoi de n'importe quel type de données empaqueté dans un objet
  - Pour un entier renvoi d'un objet Integer pour une Date un objet java.sql.Date

getXXX() : Spécialisées par données

CHAR	String	getString()
VARCHAR		
LONGVARCHAR		
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL		
BIT	Boolean	getBoolean
TINYINT	Integer (Byte)	getByte()
SMALLINT	Integer (short)	getShort()
INTEGER	Integer (int)	getInt()
BIGINT	Long (long)	getLong()
FLOAT	Double (double)	getDouble()
DOUBLE		
BINARY	Byte[]	getBytes()
VARBINARY		
DATE	java.sql.date	getDate()
TIME	java.sql.time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimeStamp()

Les cas particuliers

- Valeurs « null » dans un colonne
- Les données de grandes tailles :accès par flot
- Les dates et heures

Une colonne vide de la base de données

- Contient une valeur « null »
- Cas traité différemment selon les APIs JDBC
- Pas de méthode pour anticiper la présence d'une valeur « null »
- getInt() qui ne renvoie pas un objet, renvoie la valeur –1
- JDBC contourne le Pb avec la méthode wasNull() :

```
int nbStock = rs.getInt("STOCK");
if ( rs.wasNull() )
      System.out.println("Le résultat était null");
else
       System.out.println("En stock  était null " + nbStock);
```

Autre possibilité pour la valeur null

- Appel à getInt() :

```
int nbStock = rs.getInt("STOCK");
if ( rs.wasNull() )
     System.out.println("Le résultat était null");
else
      System.out.println("En stock  était null " + nbStock);
```

- Appeler un objet par getObject():

```
Integer nbStock = rs.getInt("STOCK");
if ( nbStock == null )
   System.out.println("Le résultat était null");
else
      System.out.println("En stock  était null " + nbStock.value());
```

Les données de grande taille

- Lire des images ou de grands documents
- Les méthodes de resultSet à utiliser sont :
  - getAsciiStream()
  - getBinaryStream()
  - getUnicodeStream()
- Ces 3 méthodes renvoient un inputStream
- L'API JDBC 2.0 possède des objets « Blob » et « Clob »

Les données de grande taille (2)

- Exemple lecture d'une image et envoie sur un OutputStream :

```
ResultSet rs = stmt.executeQuery(
        "SELECT IMAGE FROM PHOTOS where PID = " +
                req.getParameter("PID"));
While ( rs.next() ) {
        BufferedInputStream gifData =
                new BufferedInputStream(rs.getBinaryStream("IMAGE"));
        byte[] buf = new byte[4*1024]; // tampon de 4 K octets
        int len;
        while (len = gifData.read(buf, o, buf.length)) != -1) {
                out.write(buf, o, len);
        }
}
```

Les dates et heures

- 3 classes Java :
  - Java.sql.Date
  - Java.sql.Time
  - Java.sql.TimeStamp
- Elles correspondent aux types SQL :
  - DATE : jour, mois, année
  - TIME : ne donne que l'heure sans information de jour
  - TIMESTAMP : contient les 2 avec la précision de la nanoseconde
- La classe java.util.Date ne convient à aucun des 3 types SQL (précision : millisecondes)
- JDBC a adapter cette classe standard

Les dates et heures (2)

- La méthode de codage varie selon le paquetage SGDB
- JDBC supporte les séquences ISO d'échappement dans les dates
- Les pilotes ont la charge de traduire ces séquences dans la forme imposée par le SGDB sous-jacent
- La syntaxe :
  - DATE : {d 'aaaa-mm-jj}
  - TIME : {t 'hh:mm:ss'}
  - TIMESTAMP : {ts 'aaaa-mm-jj hh:mm:ss.microsecondes.ns'} par défaut ne précise que les secondes
- Exemple :

```
dateSQL.execute("INSERT INTO AMIS(ANNIVERSAIRE) " +
                "VALUE ({d '1978-12-14'}) " ) ;
```

Les signaux

- Pour la gestion des erreurs
- Les avertissements SQL

Les situations d'erreur

- Un objet JDBC doit arrêter l'exécution
  - Mauvaise connexion à une BD
  - Requête SQL mal formée
  - Des droits insuffisants
- Il lance un SQLException (throw Java)
  - Étend la classe java.lang.Exception
  - Ajout de nouvelles méthodes :
    - getNextException() : chaînage d'une suite d'objet exception
    - getSQLState() : code d'états de SQL-92
    - getErrorCode() : pour renseigner une erreur

Les situations d'erreur : exemple

```
try {
    // le code
}
catch ( SQLException e ) {
    while (e  != null) {
        System.out.println("  \nSQL Exception:");
        System.out.println(e.getMessage());
        System.out.println("  ANSI-92 SQL State: " + e.getSQLState());
        System.out.println("  Vendor Error Code: " + e.getErrorCode();
        e = e.getNextException();
    }
}
```

Les avertissements SQL

- Les classes JDBC peuvent générer (et non lancer) des exceptions SQLWarning
- Cas d'une anomalie ou la gravité ne justifie pas l'arrêt du programme
  - Mode non supporté
  - Différence entre warning et erreur dépend de la BD
- SQLWarning encapsule la même information que SQLException
- SQLException utilisé dans un try/catch pas SQLWarning
- Les avertissements sont repris par les méthodes getWarnings() des interfaces Connection, Statement, ResultSet, CallableStatement, et PreparedStatement

Les avertissements SQL : exemple

- SQLWarning implémente les méthodes suivantes :
  - getMessage(), getSQLState(), getErrorCode()
- Au cours de la mise au point d'une application : détecter les moindres anomalies au niveau de la BD
- Exemple :

```
void printWarnings( SQLWarning wan) {
    while ( warn != null ) {
        System.out.println("  \nSQL Warning:");
        System.out.println(warn.getMessage());
        System.out.println("  ANSI-92 SQL State: " + warn.getSQLState());
        System.out.println("  Vendor Error Code: " + warn.getErrorCode();
        warn = warn.getNextWarning();
    }
}
                        ResultSet = rs.statement.executeQuery(
                                "SELECT * FROM CLIENTS");
                        printWarnings(stmt.getWarnings());
                        printWarnings(rs.getWarnings());
```

L'objet PreparedStatement

- Proche de l'objet Statement : exécute des instructions SQL
- Instruction SQL compilée, plus efficace. N'effectue plus l'analyse des chaînes de caractères
- Un objet PreparedStatement se prépare à partir d'un objet Connection comme pour Statement
  - Méthode prepareStatement() au lieu de createStatement()
- Exemple :

```
PreparedStatement pstmt = con.preparedStatement(
                                "INSERT INTO EMPLOYES (
NOM, TELEPHONE) (?, ?)");
```

- La ligne insère un nom et numéro de téléphone dans la table EMPLOYES sans indiquer la valeur
- La valeur est indiquée au moment de l'appel

```
pstmt.clearParameters();            : effacer les spécifications antérieures
pstmt.setString(1, "Robert Dupond"); : positionner les valeurs
pstmt.setString(2, "03 88 67 68 69"); : un grand nombre de setXXX
pstmt.executeUpdate();              : lancer l'execution SQL
```

L'objet PreparedStatement : setObject()

- Cette méthode insère des objets Java si on peut les convertir en types SQL standard
- 3 variantes :
  - setObject(int parameter index, Object o, int targetSqlType, int scale)
  - setObject(int parameter index, Object o, int targetSqlType)
  - setObject(int parameter index, Object o)
- On peut contrôler l'association entre un objet Java et le type SQL
- Si absent une association par défaut
- Les méthodes setXXX() effectuent une association par défaut

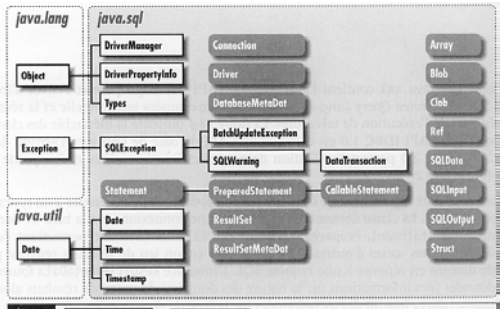
Insertion de valeurs « null » dans BD

- Par setNull() ou par setObject(int index, Object o) avec o un objet Integer et o = null
- Ou si setXXX() accepte un objet comme setTime()

- Exemple :

```
Integer i = new Integer(32);
pstmt.setObject(1, i, Types.INTEGER);
pstmt.setObject(2, null, Types.VARCHAR);
//ou pstmt.setNull(2, Types.VARCHAR);
```

Le paquetage java.sql



Les métadonnées ou la découverte dynamique des BD

- La plupart des JDBC ne peuvent être utilisés qu'avec certaines tables d'une BD
- Le programme connaît précisément le type des données
- Dans certains cas, un programme peut découvrir de manière dynamique la structure du jeu de résultat et la configuration de la BD
- Ces informatisations sont appelées des métadonnées.
- JDBC possède 2 classes permettant de les utiliser
  - DatabaseMetaData
  - ResultSetMetaData

L'interface DatabaseMetaData

- Au travers de cette classe Java.sql.DatabaseMetaData un programme peu affiner son SQL et utiliser SQL au vol pour s'adapter
- Les métadonnées sont associées à une connexion précise et sont obtenues par :
  DatabaseMetaData dmeta = con.getMetaData();
- Les données retournées :
  - Des String : getURL()
  - Des booléens : nullsAreSortedHigh()
  - Des entiers : getMaxConnections()
- D'autres méthodes renvoient des objets ResultSet de longueur variable :
  - getColums()
  - getTableTypes()
  - getPrivileges()
  - getTables() : contient le nom de toutes les tables

Un exemple : extraction des métadonnées

```
import java.sql.*;
import java.util.StringTokenizer;

public class DBViewer {

  final static String jdbcURL = "jdbc:odbc:customerdsn";
  final static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";

  public static void main(java.lang.String[] args) {
    System.out.println("--- Database Viewer ---");
    try {
      Class.forName(jdbcDriver);      // enregistrement du pilote
      Connection con = DriverManager.getConnection(jdbcURL, "", "");
      DatabaseMetaData dbmd = con.getMetaData();

      System.out.println("Nom du pilote: " + dbmd.getDriverName());
      System.out.println("Produit SGBDR: " + dbmd.getDatabaseProductName());
      System.out.println("clés SQL autorisés:");
      StringTokenizer st = new StringTokenizer(dbmd.getSQLKeywords(), ",");
    ... // suite
```

Exemple (2)

```
  StringTokenizer st = new StringTokenizer(dbmd.getSQLKeywords(), ",");
  while(st.hasMoreTokens())
    System.out.println(" " + st.nextToken());

  // Obtenir un  ResultSet avec toutes les tables dans la BD
  // Spécification d'un table_type de "TABLE" pour éviter des tables systèmes
  // les vues et autres définitions de bas niveaux
  String[] tableTypes = { "TABLE" };
  ResultSet allTables = dbmd.getTables(null,null,null,tableTypes);
  while(allTables.next()) {
    String table_name = allTables.getString("TABLE_NAME");
    System.out.println("Nom Table: " + table_name);
    System.out.println("Type Table: " + allTables.getString("TABLE_TYPE"));
    System.out.println("Index: ");

    // Obtenir la liste des index de la table courante
    ...// suite
```

Exemple (3)

```
...// suite
      // Obtenir la liste des index de la table courante
      ResultSet indexList =
                dbmd.getIndexInfo(null,null,table_name,false,false);
      while( indexList.next() ) {
            System.out.println(" Nom index: " +
                  indexList.getString("INDEX_NAME"));
            System.out.println(" Nom colonne:" +
            indexList.getString("COLUMN_NAME"));
      }     // indexList.next()
      indexList.close();
    }       // allTablesNext()
    allTables.close();
    con.close();
  }         // try
...// suite
```

Exemple (4)

```
..// suite
            indexList.close();
      }     // indexList.next()
      allTables.close();
      con.close();
    }       // allTablesNext()
    catch (ClassNotFoundException e) {
            System.out.println("mpossible de changer le pilote de la base");
    }       // try
    catch (SQLException e) {
      System.out.println("Exception SQL : " + e.getMessage());
    }
  }         // main()
}           //class DBViewer
```

Résultat de l'exécution

```
--- Database Viewer ---
Nom du pilote : JDBC-ODBC Bridge (odbcjt32.dll)
Produit SGBDR : ACCESS
Mots clés autorisés :
    ALPHANUMERIC
    AUTOINCREMENT
    BINARY
    BYTE
    FLOAT8
    .....
Nom Table : Clients
Type Table : TABLE
Index:
    Nom index : PrimaryKey
    Nom colonne : NumClient
    Nom index : AdresseIndex
    Nom colonne : Adresse
```

L'interface ResultSetMetaData

- Informe sur la structure d'un jeu de résultat
- Les données fournies :
  - Le nombre de colonnes présentes
  - Leur nom
  - Leur type
- Exemple : affichage du contenu d'une table et le type de données des colonnes
- Certaines versions du JDK (incluant celui la version 1.15 de NT) ne supportent pas les ResultSet.getMetaData().

Un exemple

```
import java.sql.*;
import java.util.StringTokenizer;

public class TableViewer {
  final static String jdbcURL = "jdbc:oracle:oci8:@net8name";
  final static String jdbcDriver = "oracle.jdbc.driver.OracleDriver";
  final static String table = "CLIENTS";

  public static void main(java.lang.String[] args) {
    System.out.println("--- Table Viewer ---");
    try {
      Class.forName(jdbcDriver);
      Connection con =
            DriverManager.getConnection(jdbcURL, "ssdata", "ssite");
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM "+ table);
      ResultSetMetaData rsmd = rs.getMetaData();
.../ suite
```

Exemple (2)

```
...//suite

  int columnCount = rsmd.getColumnCount();
  for (int col = 1; col <= columnCount; col++) {
        System.out.print(rsmd.getColumnLabel(col));
        System.out.print(" (" + rsmd.getColumnTypeName(col)+")");
        if(col < columnCount)  System.out.print(", ");
  }     // for
  System.out.println();

  while(rs.next()) {
        for(int col = 1; col <= columnCount; col++) {
              System.out.print(rs.getString(col));
              if(col < columnCount) System.out.print(", ");
        }     // for
    System.out.println();
  }     // while

  ...//suite
```

Exemple (3)

```
      while(rs.next()) {
            for(int col = 1; col <= columnCount; col++) {
                  System.out.print(rs.getString(col));
                  if(col < columnCount) System.out.print(", ");
            }     // for
        System.out.println();
      }     // while
      rs.close();
      stmt.close();
      con.close();
    }       // try
    catch (ClassNotFoundException e) {
            System.out.println("Unable to load database driver class");
    }
    catch (SQLException e) {
            System.out.println("SQL Exception: " + e.getMessage());
    }
  }         // main()
}           // class TableViewer
```

Le résultat

- Les méthodes utilisées :
  - getColumnCount(), getColumnLabel(), getColumnTypeName()
- Exemple :

--- Table Viewer ---
NumClient (SHORT), NomClient (VARCHAR), AdresseClient(VARCHAR)
1, Julie Dupont, 12 rue des Primevères
2, Louis Renaud, 213 rue des roses
3, Robert Winston, 17 rue du cimetière

Les transactions (1)

- Un groupe d'opérations qui doivent se comporter de façon atomique
- Elles forment une opération unique et indivisible
- La solution : les services transactionnels de JDBC
- Les étapes :
  - Lancer la transaction
  - Exécuter les opérations qui la composent
  - Valider les opération en cas de succès ou rétablir l'état antérieur en cas d'échec des opérations
- Les modifications ne sont visibles que si la transaction a été validée
- L'objet Connection prend en charge la gestion transactionnelle

Les transactions (2)

- Avec JDBC toute opération est une transaction
- Une nouvelle connexion se lance en mode auto transactionnel autovalidant (auto-commit)
- Chaque opération est considérée comme une transaction et est validée
- Pour envoyer plusieurs opérations pour une transaction :
  - Appeler la méthode : setAutoCommit(false)
  - Les instruction SQL sont placées à la suite
  - getAutoCommit() : état du mode transactionnel
  - commit() : valider la transaction
  - rollback() : rétablir la situation initiale

Les transactions : exemple

```
try {
    con.setAutoCommit(false);
    // des commandes SQL
    stmt.executeUpdate(
        "UPDATE INVENTORY SET ONHAND = 10 WHERE ID = 5");
    stmt.executeUpdate("INSERT INTO SHIPPING (QTY) VALUES (5) ");
    con.commit();
}
catch ( SQLException e) {
    con.rollback(); // annule les effets de la transaction
}
```

En mode autovalidant il faut absolument appeler commit() ou rollback()

Les transactions (3)

JBDC définit plusieurs modes pour isoler des transactions :

1. TRANSACTION_NONE : les transactions ne sont pas supportées ou sont interdites
2. TRANSACTION_READ_UNCOMMITED : support minimal; permet des lectures sans garanties
3. TRANSACTION_READ_COMMITED : les transactions ne peuvent lire des lignes modifiées mais non encore validées
4. TRANSACTION_REPEATABLE_READ : se protéger à la fois contre des lectures répétées et contre des lectures sans garanties
5. TRANSACTION_SERIALIZABLE : ajoute une protection contre les insertions de lignes à TRANSACTION_REPEATABLE_READ

le mode est défini par :
con.setTransactionIsolation(TRANSACTION_READ_COMMITED)

Les transactions (4)

La classe DatabaseMetaData permet de connaître le support transactionnel de la BD par les méthodes suivantes :

- getDefaultTransactionIsolation()
- supportsTransactions()
- supportsTransactionIsolationLevel()
- supportsDataDefinitionAnDataManipulationTransactions()

Les procédures stockées

- Langage de programmation propre à chaque SGBD (PL/SQL d'Oracle)
- Développeur intègre du code applicatif dans la BD
- Exemple : la procédure sp_interest prend 2 paramètres en entrée, un numéro de compte et un solde et renvoie les solde mis à jour

```
CREATE OR REPLACE PROCEDURE sp_interest
        (id IN INTEGER bal IN OUT FLOAT) IS
BEGIN
  SELECT balance INTO bal FROM accounts WHERE account_id = id;
  bal := bal+bal*0.03;
  UPDATE accounts SET balance = bal WHERE account_id = id;
END;
```

Les procédures stockées (2)

- L'interface CallableStatement est l'objet JDBC qui supportent les procédures stockées
- La classe Connection possède une méthode prepareCall() (similaire à PreparedStatement pour les instructions compilées)
- Avantage :
  - code accessible par l'application,
  - celui-ci peut-être séparé des données
  - si la structure change on change les procédures
- JDBC normalise l'appel à ces procédures par l'emploi d'un caractère d'échappement :
  le ?, un signet qui remplace un paramètre
  - sans résultat : {call procedure [(?,[,?...])]}
  - avec résultat : {? = call procedure [(?,[,?...])]}

Les procédures stockées : exemple

```
// instanciation d'un objet CallableStatement
CallableStatement cstmt = con.pepareCall("{call sp_interest(?,?)} ");
cstmt.registerOut(2, Types.FLOAT);          // un paramètre de sortie
cstmt.setInt(1, accountId);                 // les paramètres à substituer
cstmt.setInt(2, 2343.23);
cstmt.execute();
out.println("Nouveau solde:" + cstmt.getFloat(2)); // nouveau solde
```

Les séquences d'échappement

- Préserver l'indépendance du JDBC des SGB et la portabilité
- Deux exemples de mots clés :
  - call : appel de procédure stockée
  - d, t, ts : pour DATE, TIME? TIMESTAMP
- Autres cas :
  - escape : modifie la signification du caractère '_' (joker)

```
stmt.executeQuery(
"SELECT * FROM ApiDocs WHERE Field_Name LIKE 'TRANS\_%' {escape '\'}");
```

  - fn : fonction scalaire
  - suppression de l'interprétation des séquences d'échappement par setEscapeProcessing()

JDBC 2.0

- Première API JDBC (v 1.0) : paquetage additionnel du JDK 1.0
- Puis partie intégrante du noyau à partir du JDK Java 1.1
- En 1998, Sun diffuse la spécification de l'API JDBC 2.0
  - gestion améliorée des jeux de résultats
  - les bases de données orientées Java
  - les champs BLOB et CLOB
- Portabilité de JDBC 2.0 avec les version antérieures
- Début 1999, il existait très peu de pilote
- Encore aujourd'hui toutes les fonctionnalités des spécifications JDBC 2.0 n'ont pas été implémentées au niveau des pilotes

Analyse des résultats

- ResultSet : itération avec rs.next() dans JDBC 1.0, il n'existe pas de possibilité de retour en arrière
- Avec JDBC 2 : vers l'avant, vers l'arrière et positionnement à un endroit particulier :
  - first(), last(), next(), previous()
  - beforeFirst(), afterLast()
  - absoluteInt(), relativeInt()
  - isLast(), isFirst(), isAfterLast(), isAFterFirst()

Lire et modifier résultats

- Lire et modifier en même temps selon le mode :
  - TYPE_FORWARD _ONLY : standard, en lecture
  - TYPE_SCROLL_INSENSITIVE : en avant, en arrière, insensible aux changements
  - TYPE_SCROLL_SENSITIVE : en avant, en arrière, sensible aus changements
  - CONCUR_READ_ONLY : l'ensemble des résultats ne peut pas être utilisé pour mettre à jour la base
  - CONCUR_UPDATABLE : : l'ensemble des résultats peut être utilisé pour mettre à jour la base

```
Statement stmt =
con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_READ_ONLY);
                     // parcourir sans modifier les résultats
```

Un ResultSet modifiable

- Les données sont modifiées par les méthodes updateXXX() (autant que des setXXX())

```
Statement stmt =
con.createStatement( ResultSet.TYPE_SCROLL_SENSITIVE,
                     ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery(
    "SELECT NAME, CUSTOMER_ID FROM CUSTOMERS");
rs.first();
rs.update(2, 35243); // modifie la 2eme entité CUSTOMER_ID
rs.updateRow();
```

- moveToInsertRow(), insertRow() : ligne vide puis insertion
- last(), deleteRow() : dernière ligne, suppression

Mise à jour par lots

- Lorsqu'il faut charger de nombreuses données, prepareStatement() avec 10000 insertions aura des Pb de performance
- La méthode addBatch() avec ou sans argument avec des instructions qui retournent un compteur de mise à jour comme CREATE, DROP, INSERT, UPDATE et DELETE

```
con.setAutoCommit(false); // en cas d'échec avec une opération on annule
Statement stmt = con.createStatement();
stmt.addBatch(
"INSERT INTO CUSTOMERS VALUES (1, DUPONT, 0388632740)");
...
int[] upCounts = stmt.executeBatch();
con.commit();
```

Mise à jour par lots

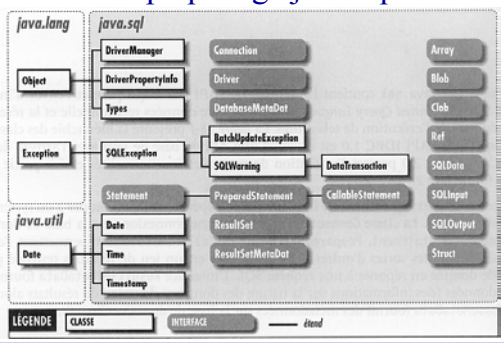
- Difference avec l'objet PreparedStatement() :

```
con.setAutoCommit(false); // en cas d'échec avec une opération on annule
PreparedStatement pstmt = con.preparedStatement(
        "INSERT INTO CUTOMERS VALUE (?, ?, ?)");
pstmt.setInt(1,1);
pstmt.setString(2, "DUPONT");
pstmt.setString(3, " 0388632740");
pstmt.addBatch();
    .....
int[] upCounts = stmt.executeBatch();
con.commit();
```

BLOB et CLOB

- Données en très grande quantité :
  - BLOB : Binary Large Object
  - CLOB : Character Large Object
- Divers noms aux BD
  - Oracle 7 : LONG et LONG RAW
  - Access : des champs objets OLE
  - Oracle 8 : introduit CLOB et BLOB
- JDBC 2, ResultSet intègre getBlob() et getClob() qui renvoient des objets Clob et Blob

Le paquetage java.sql



Le paquetage javax.sql