Une dose de Java, deux doigts de JDBC avec ou sans accès ODBC...

Par Jean-Marc Quere

Introduction

L'aventure continue...

Rares sont ceux qui débutent la programmation par Java. En effet, ce langage étant particulièrement récent (3 ans) et de nombreux développeurs Java viennent d'autres environnements. Après avoir passé quelques heures à la découverte des spécificités Java et autres "System.out.println("Hello, World.")", l'utilisation de celui-ci via nos navigateurs favoris devient rapidement un sujet de prédilection (applets). Heureusement, Java ne se limite pas à l'animation des pages HTML. Je vous propose donc de passer à l'étape suivante en abordant la connexion aux bases de données et les fonctionnalités du JDBC.

Pour l'ensemble des réalisations vous devez disposer d'un JDK (au moins 1.1.x) et d'un accès à une base de données (ODBC ou non). Les exemples et explications font références à une base de données ODBC intitulée "JDK" composée par le fichier "dBase" suivant :

contacts.dbf

CODE C3	NOM C40	PRENOM C40	VILLE C40	TELEPHONE C20
001	Dupond	Lazar	Paris	01 40 16 17 67
002	Lebert	Ninon	Marseille	04 51 12 35 14

Vous pouvez reconstituer cet ensemble de données avec n'importe quelle base de données pour peu que vous disposiez d'un pilote JDBC compatible avec celle-ci.

J.D.B.C.

Fonctionnalités et limitations...

"**J**ava **D**ata**B**ase **C**onnectivity" permet principalement de réaliser des connexions vers des bases de données et d'effectuer des requêtes SQL (ANSI 92 Entry Level). Son usage permet de se décharger des obligations liées aux spécificités des SGBD utilisés : moyens d'accès, syntaxe SQL, conversions de types, etc... L'utilisation des pilotes JDBC libère le développeur de toutes contraintes liées aux choix du système d'information employé. Lors d'un changement de SGBD, il suffit d'opter pour le pilote correspondant sans avoir à modifier le reste de l'application. Afin de motiver les éditeurs de SGBD à développer les pilotes JDBC nécessaires au déploiement des applications Java, les fonctions offertes ont été volontairement limitées aux plus communes : curseur uni-directionnel etc...

Deux tendances se sont développées : réalisation d'une interface JDBC *(partiellement ou totalement native : non Java)* du SGBD ou développement de classes JDBC *(en pur Java)* cliente d'un serveur *(éventuellement en pur Java également)* assurant l'interface avec le SGBD. La première solution n'autorise l'exploitation des applications que sur les systèmes pour lesquels l'interface a été spécifiquement conçue et est donc *(volontairement ?)* limitative. La seconde permet l'utilisation des applications à partir de n'importe quelle plate-forme Java. Seul le SGBD et éventuellement son serveur s'il est natif doivent être hébergés par un système hôte spécifique. Exemple : Borland DataGateway fournit une connectivité 100% Java JDBC à des données gérées au travers du Borland Database Engine *(qui, lui, est spécifiquement Windows 95/NT)*. L'intérêt de cette solution est évidente : vos contraintes se limitent aux choix des systèmes que vous administrez *(donc, dont vous êtes maître : serveurs de données)* et vous êtes libéré de tout le reste.

Lors de la présentationdu JDK *(point.dbf n° 91)* la facilité de mise en oeuvre du JDK avait été mise en évidence : une quinzaine de lignes de code pour établir la connexion avec la base et parcourir celle-ci afin de lister son contenu à l'écran. En réalité, l'exemple était bien choisi... En effet, l'utilisation de curseurs uni-directionnels implique que le parcours du résultat d'une requête soit immuable : de la première ligne à la dernière. Il est impossible de retourner à la ligne précédente, à la première ligne, de se positionner à une ligne en particulier, etc... Ces actions s'avèrent pourtant indispensables à tout développement d'application de base de données. Deux solutions palliatives sont alors couramment mises en oeuvre :

- soit le résultat de la requête est sauvegardé par l'application lors de son parcours. Les actions ultérieures étant effectuées à partir de l'image locale réalisée sous forme de tableaux, fichiers, etc. Cette solution est particulièrement adaptée aux postes nomades. Ils peuvent continuer à travailler hors connexion. L'inconvénient majeur étant l'obsolescence rapide des informations de l'image générée, donc la nécessité de les synchroniser avec la base de données principale.

- soit une gestion de ces actions est mise en oeuvre en surcouche au JDBC. Ce choix permet une gestion en temps réel, mais nécessite une utilisation en ligne. C'est au travers de son implémentation que je vous propose d'aborder les fonctionnalités du JDBC.

Connexion et Extraction de données
Paramètres indispensables

Pour se connecter à une base de données et en extraire des informations, il est nécessaire d'avoir connaissance des éléments suivants :
- nom de la classe du pilote JDBC correspondant (exemple : ***"sun.jdbc.odbc.JdbcOdbcDriver"***)
- nom et chemin d'accès (url) à la base de données ("jdbc:odbc:JDK")
- identification de l'utilisateur et mot de passe (à noter que des SGBD sous unix nécessitent en plus de la déclaration de l'identification et du mot de passe utilisateur au niveau du SGBD, l'existence d'un compte "login" sur le système pour l'utilisateur concerné) ("", "")
- requête ("select * from contacts")

Dans le meilleur des cas, l'ensemble des informations liées aux schémas des différentes tables est connu, mais il n'en est pas toujours ainsi. L'utilitaire ci-dessous permet de lister le type des colonnes du résultat.

dump.java
```
import java.sql.*;
class dump {
  public static void main(String argv[]) {
    for (int i=0;i<argv.length;i++) {
      System.out.println("- "+i+" : "+argv[i]);
    }
    if (argv.length==5) {
      try {
        Class.forName(argv[0]);
        Connection con=DriverManager.getConnection(argv[1],
          argv[2],
          argv[3]);
        Statement stmt=con.createStatement();
        ResultSet rs=stmt.executeQuery(argv[4]);
        ResultSetMetaData rsmd=rs.getMetaData();
        int nbColumn=rsmd.getColumnCount();
        for (int i=1;i<=nbColumn;i++) {
          System.out.println("- "+i+"\t"+rsmd.getColumnLabel(i)+" ("+
            rsmd.getColumnTypeName(i)+" "+
            rsmd.getPrecision(i)+"."+
            rsmd.getScale(i)+")");
        }
      }
      catch (Exception f) {
        System.out.println(f.getMessage());
        f.printStackTrace();
      }
    }
    else {
      System.out.println("Utilisation : dump <pilote JDBC> <url BD> <utilisateur> <mot passe> <SELECT...>");
    }
  }
}
```

Pour connaître le schéma de la table "contacts", il suffit d'exécuter à la commande (avec les guillemets) :

```
java dump "sun.jdbc.odbc.JdbcOdbcDriver" "jdbc:odbc:JDK" "" "" "SELECT * FROM CONTACTS"
```

Elle affiche :
- 0 : sun.jdbc.odbc.JdbcOdbcDriver *(argv[0] : pilote JDBC)*
- 1 : jdbc:odbc:JDK *(argv[1] : url de la base)*
- 2 : *(argv[2] : utilisateur "")*
- 3 : *(argv[3] : mot de passe "")*
- 4 : SELECT * FROM CONTACTS *(argv[4] : requête)*
- 1 CODE (Char 3.0)
- 2 NOM (Char 40.0)
- 3 PRENOM (Char 40.0)
- 4 VILLE (Char 40.0)
- 5 TELEPHONE (Char 20.0)

Les principes à retenir sont les suivants :

- **Class.forName()** permet de charger le pilote JDBC correspondant au SGBD employé. ATTENTION, ce chargement

est dynamique et effectué lors de l'exécution. Si l'outil de développement JAVA que vous employez dispose d'un expert de déploiement celui-ci ne verra pas la dépendance avec la classe chargée : pensez à l'inclure avec la distribution de votre application sous peine de provoquer l'exception "java.lang.ClassNotFoundException".

- **Statement** est utilisé pour exécuter la requête (select, insert, update, delete).
- **ResultSet** récupère l'ensemble de données issu de la requête.
- **ResultSetMetadata** comporte les indications de structure (type, taille...) de ResultSet

Avant que vous ne posiez la question et ne cherchiez vainement la réponse : ***il n'existe pas de fonction renvoyant le nombre de lignes d'une requête résultat !*** *(question posée tous les jours dans les groupes de "news" dédiés à Java)* Vous devez soit parcourir l'ensemble des données et les comptabiliser (ex1) soit utiliser un "SELECT COUNT..." (ex2).

ex1

```
import java.sql.*;
class ex1 {
  public static void main(String argv[]) {
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con=DriverManager.getConnection("jdbc:odbc:JDK","", "");
      Statement stmt=con.createStatement();
      ResultSet rs=stmt.executeQuery("SELECT * FROM CONTACTS");
      int i=0;
      while (rs.next()) {
        i++;
      }
      System.out.println("Nombre de ligne(s) : "+i);
    }
    catch (Exception f) {
      System.out.println(f.getMessage());
      f.printStackTrace();
    }
  }
}
```

L'instruction pour passer d'une ligne à la suivante est "rs.next()". A la lecture du source, on procède à un "rs.next()" avant de commencer à compter. La raison en est simple : le curseur d'un résultat est toujours placé en amont de la première ligne de celui-ci.

ex2

```
import java.sql.*;
class ex2 {
  public static void main(String argv[]) {
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con=DriverManager.getConnection("jdbc:odbc:JDK", "", "");
      Statement stmt=con.createStatement();
     ResultSet rs=stmt.executeQuery("SELECT COUNT(*) FROM CONTACTS");
     if (rs.next())
       System.out.println("Nombre de ligne(s) : "+rs.getInt(1));
   }
   catch (Exception f) {
     System.out.println(f.getMessage());
     f.printStackTrace();
   }
  }
}
```

Les fonctions "getDate", "getInt", "getString", "getObject", ... retournent une valeur correspondant au type indiqué (respectivement : Date, Int, String, Object, ...) à partir du nom ou du numéro de colonne (commence à 1). Les fonctions "<ResultSet>.getString('CODE')" et "<ResultSet>.getString(1)", pour un ResultSet constitué à partir de la requête "SELECT * FROM CONTACTS", retourne la même valeur. IMPORTANT : une colonne qui ne contient pas de valeur renvoie NULL (non typé). Etant sujet à interprétation par les fonctions get<Type> lors de la conversion vers le Type choisi, vérifiez préalablement si la colonne est renseignée avec "wasNull()". *Ayant eu quelques surprises avec des portages "exotiques" du JDK, je vous invite vivement à valider le comportement des fonctions get<Type> en effectuant la lecture d'un enregistrement vide.*

XDataSet

à l'attaque...

Après ce rapide tour d'horizon, entrons dans le vif du sujet en implémentant la première pierre de l'édifice : la classe XDataSet. L'objet de cette classe *(qui deviendra un JavaBean lorsqu'elle sera grande)* est d'offrir au développeur un ensemble de fonctions usuelles : open, close, first, last, prior, next, post, cancel, delete, etc. A noter que les noms des fonctions sont nettement inspirés de celle du Borland Database Engine, ce qui permettra aux utilisateurs de Delphi, Borland C++, etc de rester en terrain connu. A noter que seuls les types suivants (les plus courants) sont interfacés : VARCHAR, CHAR, DOUBLE, INTEGER et DATE. En s'inspirant de ceux-ci, l'implémentation des autres ne devraient poser aucun problème.

Limitation
lorsque la clef unique se multiplie...

Le système employé pour positionner l'enregistrement courant par rapport à l'ensemble de données est basé sur l'utilisation d'une ***clef unique*** pour la table **et** l'ensemble résultat. La table à mettre à jour doit également être précisée, car la requête d'extraction pourra mettre en relation plusieurs tables (exemple : liaison avec une table comportant en clair le libellé d'un code employé, etc). Il faudra donc veiller à ne modifier que des colonnes liées à la table à mettre à jour (ex. : code de la table à mettre à jour et non pas le libellé extrait de la table liée). Les liaisons avec les autres tables devront être **1.1** et non **1.n** (qui implique n duplications de la "clef unique" de la table à mettre à jour). A noter que dans la majorité des cas, les manipulations de l'ensemble de données ne concerneront qu'une seule table pour un même XDataSet et que les éventuelles relations pourront être gérées par le programme (-ur) par l'adjonction d'autres XDataSet. Ce type de limitations apparait également avec des solutions commerciales :

- Borland Database Engine ne permet pas de récupérer des requêtes "vivantes" *(à partir desquelles il est possible de procéder aux mises à jour)* lorsque la requête comporte des "order by"... En Delphi, il est alors nécessaire d'utiliser un autre DataSet *(en règle générale un TTable)* en liaison maître-détail sur le TQuery concerné pour procéder aux mises à jour à partir de ce dernier.

- Avec les autres environnement Java, la résolution des données de plusieurs tables est aussi problématique pour les cas 1.n : "*généralement, il n'est pas intéressant de répliquer les champs maître de chaque enregistrement détail de la requête. A la place, créez un ensemble de données détail séparé, ce qui permet une résolution correcte des modifications.*" peut-on lire pages 19-13 du Guide du Développeur Borland J Builder 2 C/S.

Une évolution possible est la gestion d'une clef composée par plusieurs colonnes. Le résultat d'une relation 1.n pourra alors disposer d'une clef unique si toutes les tables mises en relations disposent également d'une clef unique *(sous la forme d'une colonne ou d'une combinaison de colonnes)*. A noter qu'il sera nécessaire de distinguer, parmi les colonnes constituant la clef utile au positionnement, celles qui sont effectivement employées lors de la mise à jour de la table *(maître ou détail)*. Ces contraintes sont liées à la manipulation de données en SQL sans curseur.

XdataSet

```
import java.awt.*;
import java.awt.event.*;
import java.sql.*;
public class XDataSet{
private Connection dsCnct =null; // cnx avec le pilote du SGBD
private Statement dsStmt =null; // cnx requête SQL
private ResultSet dsRslt =null; // résultat requête SQL
private String dsSQL =null; // requête SQL initiale
private String dsORDER =null; // ordre de tri
private String dsTable =null; // table à mettre à jour
private String dsMemKey=null; // clef de la requête
private String[] dsMemRcd=null; // données (modifiées ou non)
private String[] dsMemIni=null; // données initiales
private int[] dsMemTyp=null; // type des colonnes
private String[] dsMemNam=null; // nom des colonnes
private String dsMemMrk=null; // position courante
private boolean dsMemBOF=false;// indication début de fichier
private boolean dsMemEOF=false;// indication fin de fichier
private boolean dsMemINS=false;// insertion en cours
private String myDriver="";
private String myConnection="";
private String myID="";
private String myPWD="";
private String mySQL="";
private String myORDER="";
private String myKey="";
private String myTable="";
```

Les variables myXXX stockent les paramètres utilisateurs. Elles seront interfacées aux travers d'un ensemble de fonction getXXX/setXXX dans le cadre de la réalisation du JavaBean. En effet, le "constructeur" d'un bean ne doit pas avoir d'argument : XDataSet(). Adopter cette technique vous évitera de nombreuses modifications de vos classes lors de leur transformation en beans. Evitez également d'utiliser le constructeur à d'autres fins que l'initialisation de la classe.

```
public XDataSet(String sDriver,String sConnection,
  String sID,String
  sPWD,String sSQL,String sORDER,String sKey,
  String sTable) {

  myDriver=sDriver;
  myConnection=sConnection;
  myID=sID;
  myPWD=sPWD;
  mySQL=sSQL;
```

```
  myORDER=sORDER;
  myKey=sKey;
  myTable=sTable;
}
```

Le constructeur XDataSet(...) se limite à la récupération des arguments passés en paramètres et à leur affectation aux variables myXXX.

```
// création de l'objet
// myDriver : accès au pilote du SGBD
// myConnection : connexion requête SQL
// myID : identification de l'utilisateur
// myPWD : mot de passe
// mySQL : requête initiale
// myORDER : ordre de tri de la requête initiale
// myKey : clef de la requête
// myTable : table à mettre à jour
// disposant nécessairement de la clef myKey
//
public void open() {
  try {
    Class.forName(myDriver);
    dsCnct=java.sql.DriverManager.getConnection(myConnection,myID,myPWD);
    dsStmt=dsCnct.createStatement();
    dsSQL=mySQL;
    dsORDER=myORDER;
    dsMemKey=myKey;
    dsTable=myTable;
    try {
      if (dsORDER.length()>0) {
        dsRslt=dsStmt.executeQuery(dsSQL+" ORDER BY "+dsORDER);
      }
      else
        dsRslt=dsStmt.executeQuery(dsSQL);
      if (dsRslt.next())
      dsMemMrk=dsRslt.getString(dsMemKey);
      dsRslt.close();
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
  catch (Exception e) {
    System.out.println(e);
  }
  refresh();
}
```

Pour accéder à l'ensemble des données, la fonction procède au chargement du pilote JDBC (Class.forName), à l'ouverture d'une connexion avec le SGBD (java.sql. DriverManager.getConnection) et à l'exécution d'une requête (createStatement, executeQuery). La requête ressemble à "SELECT < * ou champ1, champ2, ... champ n> FROM table1, ..., table n ORDER BY champ1, champ2, ... champ n". La variable **dsMemMrk** mémorise la position courante.

```
public void close() {
  try {
    dsStmt.close();
    dsCnct.close();
  }
  catch (Exception e) {
    System.out.println(e);
  }
  dsCnct =null; // connexion avec le pilote du SGBD
  dsStmt =null; // connexion requête SQL
  dsRslt =null; // résultat requête SQL
  dsSQL =null; // requête SQL initiale
  dsORDER =null; // ordre de tri
  dsTable =null; // table à mettre à jour
  dsMemKey=null; // clef de la requête
  dsMemRcd=null; // données (modifiées ou non)
  dsMemIni=null; // données initiales
  dsMemTyp=null; // type des colonnes
  dsMemNam=null; // nom des colonnes de la requête
  dsMemMrk=null; // position courante
  dsMemBOF=false;// indication début de fichier
  dsMemEOF=false;// indication fin de fichier
  dsMemINS=false;// indication en cours d'insertion
}
```

Lors de la fermeture, l'ensemble des ressources est libéré (=null). Les drapeaux début, fin de fichiers, insertion en cours et rafraîchissement requis sont réinitialisés.

```
// mettre à jour le contenu mémorisé des colonnes
```

```
//
public void refresh() {
  try {
    String sql;
    if (dsSQL.indexOf("WHERE")>0) {
      sql=dsSQL+" AND "+dsMemKey+"='"+dsMemMrk+"'";
    }
    else {
      sql=dsSQL+" WHERE "+dsMemKey+"='"+dsMemMrk+"'";
    }
    dsRslt=dsStmt.executeQuery(sql);
    ResultSetMetaData meta=dsRslt.getMetaData();
    if (dsMemRcd==null) {
      dsMemRcd=new String[meta.getColumnCount()];
      dsMemIni=new String[meta.getColumnCount()];
      dsMemTyp=new int[meta.getColumnCount()];
      dsMemNam=new String[meta.getColumnCount()];
      for (int i=1;i<=meta.getColumnCount();i++)
        dsMemNam[i-1]=meta.getColumnName(i);
    }
    if (dsRslt.next()) {
      for (int i=0;i<meta.getColumnCount();i++) {
        dsMemTyp[i]=meta.getColumnType(i+1);
        switch (dsMemTyp[i]) {
          case java.sql.Types.VARCHAR :
          case java.sql.Types.CHAR : {
            dsMemRcd[i]=""+dsRslt.getString(dsMemNam[i]);
            if (dsRslt.wasNull())
              dsMemRcd[i]="";
            break;
          }
          case java.sql.Types.DOUBLE : {
            dsMemRcd[i]=""+dsRslt.getDouble(dsMemNam[i]);
            if (dsRslt.wasNull())
              dsMemRcd[i]="";
            break;
          }
          case java.sql.Types.INTEGER : {
            dsMemRcd[i]=""+dsRslt.getInt(dsMemNam[i]);
            if (dsRslt.wasNull())
              dsMemRcd[i]="";
            break;
          }
          case java.sql.Types.DATE : {
            dsMemRcd[i]=""+dsRslt.getDate(dsMemNam[i]);
            if (dsRslt.wasNull())
              dsMemRcd[i]="";
          }
        }
      }
    }
    else {
      for (int i=0;i<meta.getColumnCount();i++)
        dsMemRcd[i]="";
    }
    for (int i=0;i<meta.getColumnCount();i++)
      dsMemIni[i]=dsMemRcd[i];
    dsRslt.close();
  }
  catch (Exception e) {
    System.out.println(e);
  }
}
```

La fonction "refresh" recharge (ou charge dans le cas de la fonction "open") le contenu des colonnes de la ligne concernée. Le positionnement est effectué par une requête sur style : "SELECT < * ou champ1, champ2, ... champ n> FROM table1, ..., table n WHERE <colonne clef unique>=<position mémorisée>". Le nom de la colonne "clef unique" est stocké dans la variable **dsMemKey**. La position est stockée dans **dsMemMrk**. Lors de la constitution de la requête, le nom "WHERE" est recherché afin de déterminer si la nouvelle condition "<colonne clef unique>=<position mémorisée>" complète ou non un ensemble de conditions existant. Les notions d''ordre (ORDER BY) ne sont pas prises en compte puisque qu'il s'agit de se placer sur une ligne identifiée et qu'il ne sera pas nécessaire de parcourir l'ensemble résultat. Le contenu des colonnes est stocké dans deux tableaux : **dsMemRcd** et **dsMemIni**. Le premier comporte les valeurs en cours de saisie, le second les données initialement lues.

```
// ligne suivante
//
public void next() {
  String sql=null;
  String sqlPos=dsMemMrk;
  try {
    sql="SELECT "+dsMemKey+dsSQL.substring(dsSQL.indexOf("FROM")-1,dsSQL.length());
    if (dsORDER.length()>0)
      sql=sql+" ORDER BY "+dsORDER;
```

```
    dsRslt=dsStmt.executeQuery(sql);
    while (dsRslt.next()) {
      sql=dsRslt.getString(dsMemKey);
      if (sql.equals(sqlPos))
        break;
    }
    if (dsRslt.next()) {
      dsMemMrk=dsRslt.getString(dsMemKey);
    }
    else
      dsMemMrk=sqlPos;
    dsRslt.close();
  }
  catch (Exception e) {
    System.out.println(e);
  }
  dsMemBOF=false;
  if (dsMemMrk.equals(sqlPos)) {
    if (dsMemINS)
      refresh();
    dsMemEOF=true;
  }
  else
    refresh();
  dsMemINS=false;
}

// ligne précédente
//
public void prior() {
  String sql=null;
  String sqlPos=dsMemMrk;
  try {
    sql="SELECT "+dsMemKey+dsSQL.substring(dsSQL.indexOf("FROM")-1,dsSQL.length());
    if (dsORDER.length()>0)
      sql=sql+" ORDER BY "+dsORDER;
    dsRslt=dsStmt.executeQuery(sql);
    sql=null;
    while (dsRslt.next() && ((sql==null) || (! sql.equals(sqlPos)))) {
      dsMemMrk=sql;
      sql=dsRslt.getString(dsMemKey);
    }
    dsRslt.close();
    if (dsMemMrk==null)
      dsMemMrk=sqlPos;
  }
  catch (Exception e) {
    System.out.println(e);
  }
  dsMemEOF=false;
  if (dsMemMrk.equals(sqlPos)) {
    if (dsMemINS)
      refresh();
    dsMemBOF=true;
  }
  else
    refresh();
  dsMemINS=false;
}
```

Le principe retenu pour la navigation (précédent, suivant, etc.) dans l'ensemble résultat est le suivant : une première requête permet de récupérer l'ensemble des clefs uniques dans l'ordre choisi. La position courante (contenu de dsMemMrk) est comparée aux différentes valeurs de l'ensemble résultat, jusqu'à ce qu'elle soit retrouvée. Dans ce cas la valeur suivante comporte la clef utile au positionnement sur la prochaine ligne et la valeur précédemment lue celle de la ligne précédente. Les indicateurs dsMemEOF (fin de fichier atteint) et dsMemTOP (sommet du fichier atteint) sont actualisés en conséquence.

```
// première ligne
//
public void first() {
  String sql=null;
  String sqlPos=dsMemMrk;
  try {
    if (dsORDER.length()>0) {
      sql=dsSQL+" ORDER BY "+dsORDER;
    }
    else
      sql=dsSQL;
    dsRslt=dsStmt.executeQuery(sql);
    sql="";
    if (dsRslt.next())
      dsMemMrk=dsRslt.getString(dsMemKey);
    dsRslt.close();
  }
  catch (Exception e) {
```

```
    System.out.println(e);
  }
  dsMemEOF=false;
  if (dsMemMrk.equals(sqlPos)) {
    if (dsMemINS)
      refresh();
    dsMemBOF=true;
  }
  else
    refresh();
  dsMemINS=false;
}

// dernière ligne
//
public void last() {
  String sql=null;
  String sqlPos=dsMemMrk;
  try {
    sql="SELECT "+dsMemKey+dsSQL.substring(dsSQL.indexOf("FROM")-1,dsSQL.length());
    if (dsORDER.length()>0)
      sql=dsSQL+" ORDER BY "+dsORDER;
    dsRslt=dsStmt.executeQuery(sql);
    while (dsRslt.next())
      dsMemMrk=dsRslt.getString(dsMemKey);
    dsRslt.close();
  }
  catch (Exception e) {
    System.out.println(e);
  }
  dsMemBOF=false;
  if (dsMemMrk.equals(sqlPos)) {
    if (dsMemINS)
      refresh();
    dsMemEOF=true;
  }
  else
    refresh();
  dsMemINS=false;
}
```

Les valeurs des positions "première ligne" et "dernière ligne" s'obtiennent respectivement avec le contenu de la colonne "clef unique" retournée à la première et à la dernière position de l'ensemble résultat *(logique, non ?)*.

```
// mettre à jour la table
// renvoie false si erreur, true sinon
//
public boolean post() {
  String sqlSet=null;
  String sqlKey=null;
  if (dsMemINS) {
    // nouvelle ligne
    //
    sqlSet="INSERT INTO "+dsTable+" (";
    sqlKey=") VALUES (";
    for (int i=0;i<dsMemRcd.length;i++) {
      if (! dsMemRcd[i].equals(""))
        switch (dsMemTyp[i]) {
          case java.sql.Types.VARCHAR :
          case java.sql.Types.CHAR : {
            sqlSet=sqlSet+dsMemNam[i]+",";
            sqlKey=sqlKey+"'"+dsMemRcd[i]+"',";
            break;
          }
          case java.sql.Types.DOUBLE :
          case java.sql.Types.INTEGER : {
            sqlSet=sqlSet+dsMemNam[i]+",";
            sqlKey=sqlKey+dsMemRcd[i]+",";
            break;
          }
          case java.sql.Types.DATE : {
            sqlSet=sqlSet+dsMemNam[i]+",";
            sqlKey=sqlKey+"'"+dsMemRcd[i]+"',";
          }
        }
        if (dsMemNam[i].equals(dsMemKey))
          dsMemMrk=dsMemRcd[i];
    }
    sqlSet=sqlSet.substring(0,sqlSet.length()-1)+
    sqlKey.substring(0,sqlKey.length()-1)+")";
    if (! sqlSet.equals("INSERT INTO "+dsTable+" (")) {
      try {
        dsStmt.executeUpdate(sqlSet);
      }
      catch (Exception e) {
        System.out.println(e);
```

```
        return false;
      }
    }
  }
  else {
    // mise à jour
    //
    sqlSet="UPDATE "+dsTable+" SET ";
    sqlKey=" WHERE "+dsMemKey+"='";
    for (int i=0;i<dsMemRcd.length;i++) {
      if (! dsMemIni[i].equals(dsMemRcd[i])) {
        switch (dsMemTyp[i]) {
          case java.sql.Types.VARCHAR :
          case java.sql.Types.CHAR :
            sqlSet=sqlSet+dsMemNam[i]+"='"+dsMemRcd[i]+"',";
            break;
          case java.sql.Types.DOUBLE :
          case java.sql.Types.INTEGER : {
            if (dsMemRcd[i].equals("")) {
              sqlSet=sqlSet+dsMemNam[i]+"=NULL,";
            }
            else
              sqlSet=sqlSet+dsMemNam[i]+"="+dsMemRcd[i]+",";
            break;
          }
          case java.sql.Types.DATE : {
            if (dsMemRcd[i].equals("")) {
              sqlSet=sqlSet+dsMemNam[i]+"=NULL,";
            }
            else
              sqlSet=sqlSet+dsMemNam[i]+"='"+dsMemRcd[i]+"',";
          }
        }
      }
      if (dsMemNam[i].equals(dsMemKey)) {
        sqlKey=sqlKey+dsMemIni[i]+"'";
        dsMemMrk=dsMemRcd[i];
      }
    }
    sqlSet=sqlSet.substring(0,sqlSet.length()-1)+sqlKey;
    if (! sqlSet.equals("UPDATE "+dsTable+" SET ")) {
      try {
        dsStmt.executeUpdate(sqlSet);
      }
      catch (Exception e) {
        System.out.println(e);
        return false;
      }
    }
  }
  dsMemBOF=false;
  dsMemEOF=false;
  dsMemINS=false;
  return true;
}
```

La fonction de mise à jour (post) procède à la comparaison des valeurs mémorisées à celles nouvellement saisies pour composer une requête de mise à jour : "UPDATE <table à mettre à jour> SET <colonne1=valeur1>, ... <colonne n=valeur n> WHERE <colonne clef unique>=<position mémorisée>".

```
// renvoie le contenu d'une colonne (état en cours)
//
public String getFieldByName(String myField) {
  for (int i=0;i<dsMemNam.length;i++) {
    if (myField.equals(dsMemNam[i]))
    return dsMemRcd[i];
  }
  return "";
}

public void cancel() {
  for (int i=0;i<dsMemRcd.length;i++)
    dsMemRcd[i]=dsMemIni[i];
}

// actualise le contenu d'une colonne (état en cours)
//
public void setFieldByName(String myField, String myValue) {
  for (int i=0;i<dsMemNam.length;i++) {
    if (myField.equals(dsMemNam[i]))
      dsMemRcd[i]=myValue;
  }
}

// renvoie l'indication fin de fichier atteinte
```

```
public boolean eof() { return dsMemEOF; }

// renvoie l'indication début de fichier atteint
public boolean bof() { return dsMemBOF; }

// insertion / ajout d'une ligne
public void insert() {
  for (int i=0;i<dsMemNam.length;i++)
    dsMemRcd[i]="";
  dsMemINS=true;
}

// supprimer la ligne
// renvoie false si erreur, true sinon
public boolean delete() {
  String sqlDel="DELETE FROM "+dsTable+" WHERE "+dsMemKey+"='"+dsMemMrk+"'";
  try {
    prior();
    dsStmt.executeUpdate(sqlDel);
  }
  catch (Exception e) {
    System.out.println(e);
    return false;
  }
  dsMemBOF=false;
  dsMemEOF=false;
  dsMemINS=false;
  return true;
}
}
```

Améliorations
Après les vacances, quelques devoirs...

La clef unique doit actuellement impérativement être STRING. La gestion d'autres types ainsi que de clefs à plusieurs colonnes peut être envisagée. Afin de diminuer les temps de réponse l'utilisation de "preparedStatement" peut être envisagée. **ATTENTION** : vérifiez préalablement si le pilote JDBC employé supporte ces fonctionnalités *(surprise ...)*.

XDataSet (bis)
Un peu de pratique...

xdex1.java
parcourir un résultat de la dernière ligne vers la première...

```
import XDataSet;
public class xdex1 {
  public static void main(String argv[]) {
    try {
      XDataSet mySet=new XDataSet("sun.jdbc.odbc.JdbcOdbcDriver",
        "jdbc:odbc:JDK","","",
        "SELECT * FROM CONTACTS",
        "NOM,PRENOM",
        "CODE",
        "CONTACTS");
      mySet.open();
      mySet.last();
      while (! mySet.bof() ) {
        System.out.println("->"+mySet.getFieldByName("NOM")+" "+
        mySet.getFieldByName("PRENOM"));
        mySet.prior();
      }
      mySet.close();
    }
    catch (Exception f) {
      System.out.println(f.getMessage());
      f.printStackTrace();
    }
  }
}
```

xdex2.java
annuler et poster une modification...

```
import XDataSet;
public class xdex2 {
  public static void main(String argv[]) {
    try {
      XDataSet mySet=new XDataSet("sun.jdbc.odbc.JdbcOdbcDriver",
        "jdbc:odbc:JDK","","",
```

```
        "SELECT * FROM CONTACTS WHERE NOM='Lebert'",
        "NOM,PRENOM",
        "CODE",
        "CONTACTS");
      mySet.open();
      if (! mySet.eof() ) {
        tring prenom=mySet.getFieldByName("PRENOM").equals("Ninon") ? "Charles" : "Ninon";
        System.out.println("Origine-> "+mySet.getFieldByName("NOM")+" "+mySet.getFieldByName("PRENOM"));
        mySet.setFieldByName("PRENOM",prenom);
        System.out.println("Modifie-> "+mySet.getFieldByName("NOM")+" "+mySet.getFieldByName("PRENOM"));
        mySet.cancel();
        System.out.println("Cancel -> "+mySet.getFieldByName("NOM")+" "+mySet.getFieldByName("PRENOM"));
        mySet.setFieldByName("PRENOM",prenom);
        System.out.println("Post -> "+mySet.getFieldByName("NOM")+" "+mySet.getFieldByName("PRENOM"));
        mySet.post();
      }
      mySet.close();
    }
    catch (Exception f) {
      System.out.println(f.getMessage());
      f.printStackTrace();
   }
  }
}
```

Vérifiez **TOUJOURS** quel est le délimiteur employé pour les chaînes dans les requêtes SQL *(ici, il s'agit de la cote " ' ")*. Si un des termes de la requête fait l'objet d'une saisie *(codes ou libellés recherchés, etc..)*, il est **IMPERATIF** de contrôler la valeur saisie afin de doubler le cas échéant les caractères délimiteurs employés pour qu'ils soient interprétés comme des caractères et non comme délimiteurs lors du traitement SQL.

Bibliographie
Un peu de lecture...

SQL 2 Initiation Programmation de Christian Marée, Guy Ledant (édition Armand Colin) - Ouvrage plutôt scolaire qui offre une compilation de l'essentiel avec de nombreux exemples (et exercices) indispensable aux néophytes, il permet aux experts de dissiper un doute ou de retrouver une syntaxe oubliée.

Java Client-Serveur de Cédric Nicolas, Christophe Avare, Frédéric Najman (édition Eyrolles) - Indispensable à tout développeur Java pour les mêmes raisons qu'un programmeur sur PC doit disposer de "La bible PC, programmation système" (Micro-Application).