Introduction à JDBC

Introduction

JDBC, Java Data Base Connectivity est un ensemble de classes (API – Application Programming Interface --JAVA) permettant de se connecter à une base de données relationnelle en utilisant des requêtes SQL ou des procédures stockées.

L'API JDBC a été développée de manière à pouvoir se connecter à n'importe quelle base de données avec la même syntaxe; cette API est dite indépendante du SGBD utilisé.

Les classes JDBC font partie du package **java.sql** et **javax.sql**

JDBC permet entre autre :

1. L'établissement d'une connexion avec le SGBD.
2. L'envoi de requêtes SQL au SGBD, à partir du programme java: création de tables, sélection de données,...
3. Le traitement, au niveau du programme, des données retournées par le SGBD.
4. Le traitement des erreurs retournées par le SGBD lors de l'exécution d'une instruction.

Pilote de bases de données ou driver JDBC

- Un pilote ou driver JDBC est un "logiciel" qui permet de convertir les requêtes JDBC en requêtes spécifiques auprès de la base de données.
- Ce "logiciel" est en fait une implémentation de l'interface Driver, du package java.sql.

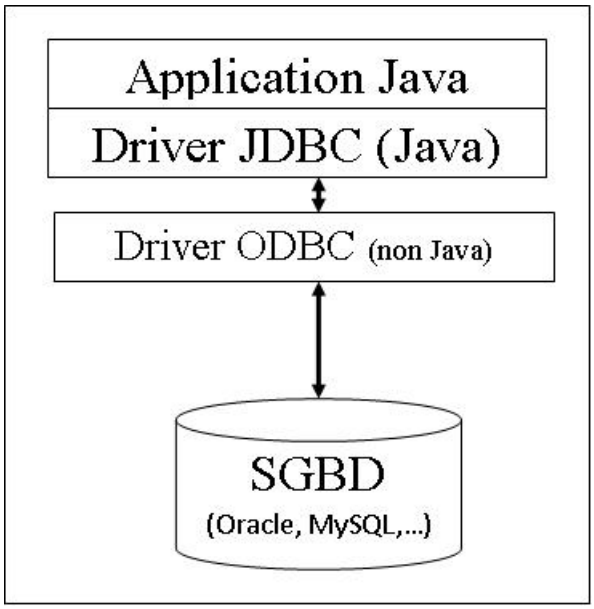
Dans le cas d'oracle, les drivers JDBC sont fournis par Oracle (en principe installés avec la base de données) téléchargeables à l'adresse.

http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html

Types de drivers JDBC

Il existe plusieurs types de pilotes JDBC

Les drivers de Type 1 : ODBC-JDBC bridges, ODBC (Open Data Base Connectivity) est une interface propre à Microsoft et qui permet l'accès à n'importe quelle base de données (Panneau de configuration /Outils d'administration/ Sources de données ODBC



Chaque requête JDBC est convertie par ce pilote en requête ODBC qui est par la suite convertie une seconde fois dans le langage spécifique de la base de donnée.

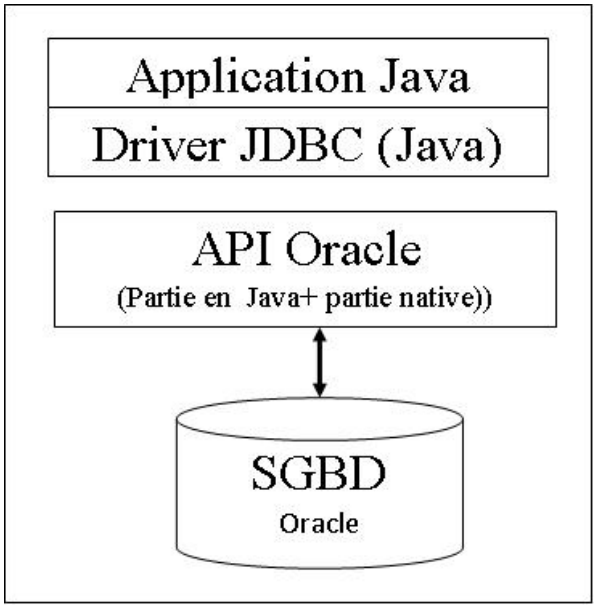
Cette technique est la moins optimale puisque les bases de données sont disponibles uniquement que par technologie ODBC.

Le SDK de Java fournit un pilote JDBC-ODBC :« sun.jdbc.odbc.JdbcOdbcDriver ».

Les drivers de Type 2

Ce type de driver traduit les appels de JDBC à un SGBD particulier, grâce à un mélange d'API java et d'API natives. (propre au SGBD).

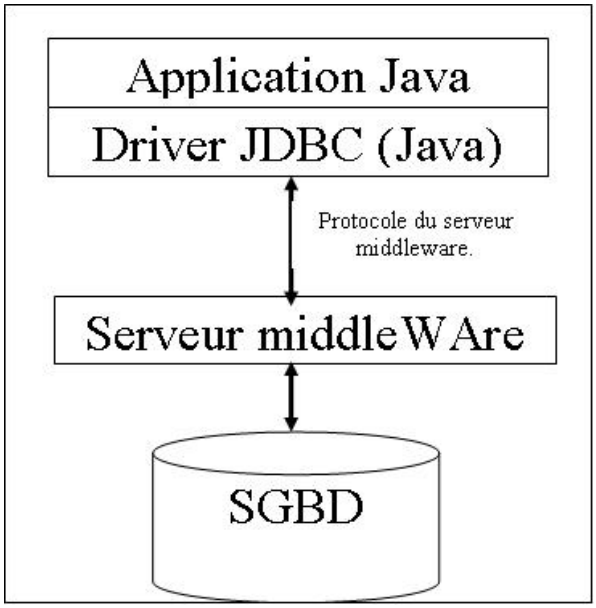
Ce Driver est fourni par l'éditeur de SGBD



Il est de ce fait nécessaire de fournir au client l'API native de la base de données.

Si on change le type de la base de données, on doit changer le pilote.

Drivers de type 3 (complètement écrit en JAVA)



Permet la connexion à une base de données via un serveur intermédiaire régissant l'accès aux multiples bases de données

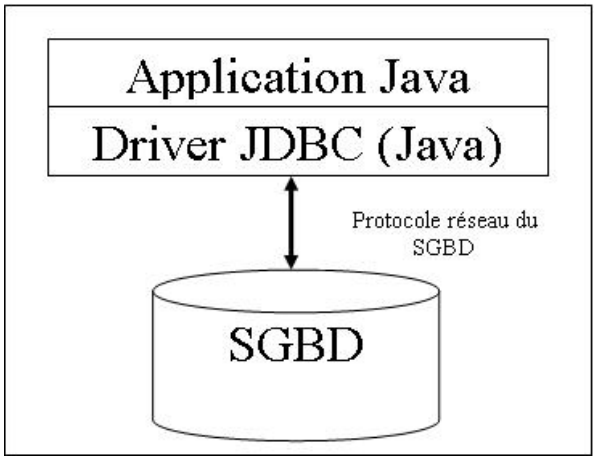
Ce type de driver est portable car écrit entièrement en java. Il est adapté pour le Web.

Cela exige une autre application serveur à installer et à entretenir.

Ce type de driver peut être facilement utilisé par une applet, mais dans ce cas le serveur intermédiaire doit obligatoirement être installé sur la machine contenant le serveur Web.

Drivers de type 4 (complètement écrit en JAVA)

Ce type de driver est connu sous le nom Direct Database Pure Java Driver), permet d'accéder directement à la base de données (sans ODBC ni Middleware). C'est le type le plus optimal.



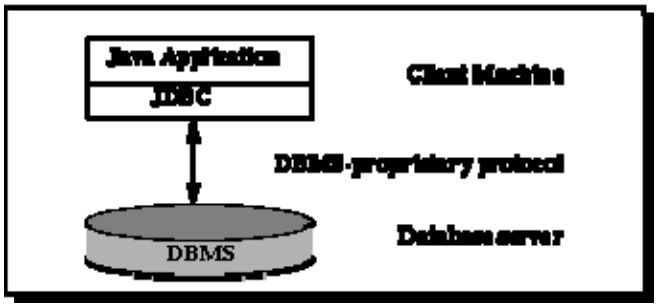
C'est ce type de driver qui sera utilisé pour accéder aux bases de données oracle

Dans ce type de driver on retrouve le driver pour oracle (thin driver ou oracle.jdbc.driver.OracleDriver) dont le format de la chaine de connexion à une base de donnes est sous formes :jdbc:oracle:thin:@chainedeconnexion

Architecture

JDBC fonctionne selon les deux modèles suivants :

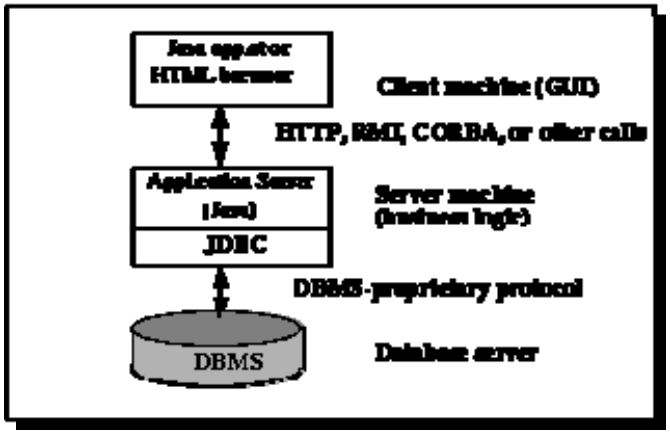
Modèle à deux couches (two-tier)



Dans le modèle two-tier, une application JAVA (ou une applet) dialogue avec le SGBD par l'intermédiaire du pilote JDBC. L'application JAVA et le pilote JDBC s'exécutent sur l'ordinateur client tandis que le SGBD est placé sur un serveur.

C'est ce type d'architecture qui nous concerne actuellement dans notre cours.

Modèles 3 couches (three-tier)



Dans le modèle three-tier, l'applet (ou l'application JAVA) ne dialogue plus directement avec un SGBD : un **middle-tier** fait le lien entre ces deux composants

Le SGBD exécute les requêtes SQL et envoie les résultats au middle tier. Ces résultats sont ensuite communiqués à l'applet sous forme d'appels http.

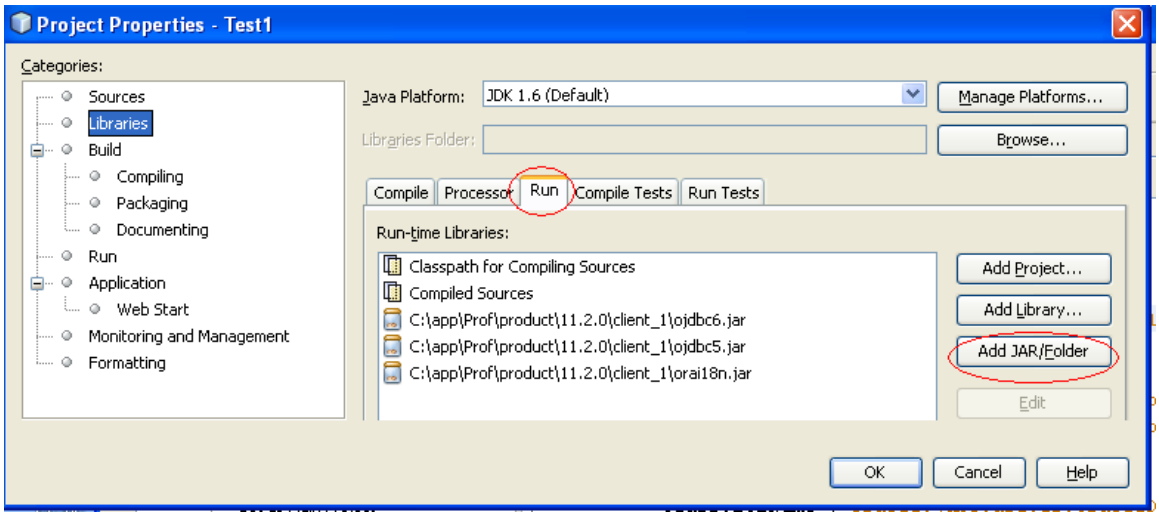
Fonctionnement

Tout programme JDBC fonctionne selon les étapes suivantes :

1. **Connexion à la base de données**
   i. Chargement du pilote de la BDD
   ii. Demande de connexion: s'identifiant auprès du SGBD et en précisant la base utilisée

2. **Traitement des commandes SQL**
3. **Traitement des résultats**
4. **Fermeture de la connexion**.

Pour utiliser NetBeans avec JDBC et oracle, vous devez inclure les librairies (.Jar) à votre projet : ces librairies sont, selon la version de votre JDK, ojdbc6.jar (pour JDK 1.6) et orai18n.jar



Établissement d'une connexion

1. **Chargement du pilote (driver)**

Pour établir une connexion, il faut d'abord charger le driver en utilisant la méthode forName de la classe Class comme suit : **Class.forName(string driver)**. Pour oracle, l'instruction est la suivante :

```
Class.forName ("oracle.jdbc.driver.OracleDriver");
```

Quand une classe **Driver** est chargée, elle doit créer une instance d'elle même et s'enregistrer auprès du **DriverManager.**

Code complet pour le chargement du pilote

```
try
        {
         Class.forName ("oracle.jdbc.driver.OracleDriver");
        System.out.println("Pilote chargé");
        }
    catch(ClassNotFoundException cnfe)
        {
        System.out.println("ERREUR : Driver manquant.");
        }
```

La méthode **Class.forName(string driver)** fait partie du pckage **java.lang et** peut lancer une exception de type «ClassNotFoundException ».

2. **Demander une connexion**
   Une fois le pilote chargé, alors on peut demander une e connexion à la base de données. Cette connexion est obtenue grâce à la méthode **getConnection** de la classe DriverManager

   Cette méthode retourne la connexion qui est en fait, un o**bjet** implémentant l'interface «Connection».

   Connection connexion = DriverManager.getConnection(url); url désigne la chaine de connexion , dans le cas d'oracle la chaine de connexion est de forme :

   ```
   "jdbc:oracle:thin:@IP:port:orcl", "nomUsager","Motdepasse"
   ```

```
  public static void main(String[] args) {
  String url = "jdbc:oracle:thin:@172.17.200.251:1521:orcl";
  String usager ="usager1";
  String motdepasse="oracle1";
          try
        {
         Class.forName ("oracle.jdbc.driver.OracleDriver");
        System.out.println("Pilote chargé");
        }
    catch(ClassNotFoundException cnfe)
        {
        System.out.println("ERREUR : Driver manquant.");
        }
  try
        {
Connection connexion = DriverManager.getConnection(url,usager, motdepasse);
        System.out.println("connécté");
        }
        catch (SQLException se)
        {
                System.out.println("ERREUR : bd  manquante ou connexion
invalide.");
        }
}
```

Exécution de requêtas SQL : créer un « statement » d'une requête particulière.

Cette étape consiste à obtenir une **déclaration (zone de description de requête ou «*statement*») *au*** travers de laquelle les requêtes SQL seront exécutées.

Il existe 3 types de déclarations:

1. Statement: instruction simple : permet d'exécuter directement et une fois l'action sur la base de données :

   Statement declaration1= connexion.createStatement();

2. PreparedStatement: instruction paramétrée. (cas des requêtes avec paramètres)

   - L'instruction est générique, des champs sont non remplis
   - Permet une précompilation de l'instruction optimisant les performances
   - Pour chaque exécution, on précise les champs manquants

   PreparedStatement declaration2= connexion PreparedStatement (requetesql);

3. CallableStatement:

Une déclaration de type « CallableStatement» permet l'accès complet aux fonctions contenues dans la base de données.(cas des procédures stockées)

Exécution de requêtas SQL : executeUpdate; executeQuery, execute

La méthode **ExecuteUpdate** est utilisée pour les requêtes DML (INSERT, DELETE, UPDATE)

Syntaxe

objetStatement.**executeUpdate**(String Requête_SQL]);

ou

objetPreparedStatement.**executeUpdate**(String Requête_SQL]);

Exemple :

----------D'abord le chargement du driver---- ensuite

```
String requete1 = "INSERT INTO employes (numemp, nom) VALUES (1,'Patoche')";
Connection connexion = DriverManager.getConnection(url1,usager,motdepasse);
        System.out.println("connexion établie");
        Statement stm = connexion.createStatement();
        stm .executeUpdate(requete1);
        System.out.println("insertion complétée");
```

Exercice 1 :

Écrire un programme JAVA JDBC qui permet

De se connecter à notre base de données Oracle

De supprimer les employés dont le codedep ='inf'

D'afficher le nombre d'enregistrements supprimer

La méthode **executeQuery,** permet d'exécuter une instruction SQL de type SELECT

Elle retourne un objet de type **ResultSet** contenant tous les résultats de la requête (les tuples sélectionnés).

Syntaxe

objetResultSet=objetStatement.**executeQuery(**String ordreSQL);

ou

objetResultSet=objetPreparedStatement.**executeQuery(**String ordreSQL);

Exploitation des résultats des requêtes SQL

L'interface ResultSet représente une table de lignes et de colonnes.

- Une ligne représente un enregistrement
- Une colonne représente un champ particulier de la table
- Un objet de type RsultSet possède un pointeur sur l'enregistrement courant. À la réception de cet objet, le pointeur se trouve devant le premier enregistrement.
- On y accède ligne par ligne, puis colonne par colonne dans la ligne.

Pour pouvoir récupérer les données contenues dans l'instance de **ResultSet**, celui-ci met à disposition des méthodes permettant de :

- Positionner le curseur sur l'enregistrement suivant :
- public boolean **next();** Renvoi un booléen indiquant la présence d'un élément suivant.
- Accéder à la valeur d'un champ (par indice ou par nom) de l'enregistrement actuellement pointé par le curseur avec les méthodes getString(), getInt(), getDate() .....
  - public String getString(int indiceCol);
  - public String getString(String nomCol);
  - etc.

À la création du ResultSet, le curseur de parcours est positionné avant la première occurrence à traiter.

Le premier indice étant 1

Exemple

String requete2= "select nom, prenom from employes where codedep = 'inf'";

----------D'abord le chargement du driver---- ensuite

```
Connection connexion = DriverManager.getConnection(url1,user,passwd);
        System.out.println("connexion établie");
        Statement stm = connexion.createStatement();
        ResultSet rest = stm.executeQuery(requete2);
        while (rest.next())
        {
           String NOMR = rest.getString("nom");
           String PRENOMR = rest.getString("prenom");
           System.out.print(NOMR + " "  + PRENOMR);
                  System.out.println();
        }
```

Exercice 2

Compléter le programme précédent pour qu'il affiche la liste des employés dont le nom commence par R;

Méthode **execute** (String ordre) : Retourne « **true** » si un résultat est disponible, « **false** » si non.

valeurbooléenne=objetStatement.**execute** (String ordre);

valeurbooléenne=objetPreparedStatement.**execute** (String ordre);

Fermeture d'une connexion :

La connexion est fermée avec la méthode close de l'objet **connexion.close();**

Utilisation du PreparedStatement

Ce type d'interface est utilisé pour des requêtes paramétrées. PreparedStatement est utilisé dans le cas 0ù la requête va être exécutée plusieurs fois. De plus Les requêtes sont précompilées.

Remarquez

1. Dans le requête le paramètre est représenté par ?
2. Les paramètres sont passés dans l'ordre de leur présentation de la requête
3. Les paramètres et les valeurs sont passés comme suit :

- [Objet PreparedStatement].setString([index],[objet String]);
- [Objet PreparedStatement].setBoolean([index],[valeur]);
- [Objet PreparedStatement].setInt([index],[valeur]);
- [Objet PreparedStatement].setFloat([index],[valeur]);
- Etc...

4. La requête est exécuté par executeUpdate() ou executeQuery

Exemple :

```
String requete3 = "update employes set prenom = ?  where nom = ? ";

// se connecter

PreparedStatement stm2 = connexion.prepareStatement(requete3);

 stm2.setString(1, "Blabla");

 stm2.setString(2, "Alpha");

stm2.executeUpdate();
```

Exercice 3 :

Écrire la partie du programme JAVA qui permet de mettre à jour le prix unitaire (paramètre) et la quantité (paramètre) selon le numéro de commande (paramètre) de la table commander

Type de parcours du ResultSet

il est possible de parcourir le ResultSet de trois façons différentes selon le type de ce dernier. Par défaut, le parcourt est farward only.

ResultSet.TYPE_FORWARD _ONLY : accès séquentiel

ResultSet.TYPE_SCROLL_INSENSITIVE, accès direct sans modification (les occurrences ne reflètent pas les mises à jour qui peuvent intervenir durant le parcours) Permet de parcourir les résultats dans les deux sens.

1. Permet de parcourir les résultats dans les deux sens grâce aux méthodes:
    - Public boolean next();
    - public boolean previous();
    - Public boolean first();
    - Public boolean last();
2. Permet de connaitre la position courante du curseur à l'intérieur du ResultSet
    - Public boolean isBeforFirst();
    - public boolean isAfterLast();
    - Public boolean isFirst();
    - Public boolean isLast();

ResultSet.TYPE_SCROLL_SENSITIVE accès direct avec modification

Même principe que le type précédent sauf que, les occurrences reflètent les mises à jour qui peuvent intervenir durant le parcours

Modification des données du ResultSet:

Par défaut, un ResultSet contient des données en lecture seulement, mais il est possible d'obtenir un ResultSet modifiable.

ResultSet.CONCUR_READ_ONLY : lecture seule

ResultSet.CONCUR_UPDATABLE : mise à jour

Le type de ResultSet et le mode d'utilisation (read only ou ubdatable) doit se faire lors de la création du Statement

Exemple

```
Statement instruction =
connexionBd.getConnexion().createStatement(ResultSet.
TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_ READ_ONLY);
```

Dans le cas d'une modification (updatable), voici les opérations pour une mise à jour ou une insertion

Modifier la valeur du type et de la colonne donnée (par indice ou par nom) de l'enregistrement actuellement **pointé** :

- public void updateString(int indiceCol, String value);
- public void updateString(String nomCol, String value);
- public void updateInt(int indiceCol, Int value);
- public void updateInt(String nomCol, Int value);
- etc.

Appliquer dans la base de données les changements effectués sur l'enregistrement actuellement pointé : public void **updateRow**();

Exemple

```
ResultSet resultat = state.executeQuery("SELECT nom, age from etudiants");

resultat.next()

resultat.updateString("nom", "martin");

resultat .updateInt("age", 28);

resultat.updateRow();
```

Dans le cas d'une insertion :

Aller sur un emplacement vide permettant d'insérer un nouvel enregistrement : public void **moveToInsertRow**();

Insérer dans la base de données l'enregistrement actuellement pointé : public void **insertRow**();

```
resultat.moveToInsertRow();
 resultat.updateString("nom", "Yanick");
resultat.updateInt("age", 19);
resultat.insertRow();
```

Méthode de déplacement dans le ResultSet

Pour se déplacer à l'intérieur du ResultSet, on utilisera la méthode **next()** pour le parcours avant et la méthode **previous()** pour le parcours inverse. Les autres méthodes sont données dans le tableau suivant.

Méthode	**Rôle**
boolean isBeforeFirst()	booléen qui indique si la position courante du curseur se trouve avant la première ligne
boolean isAfterLast()	booléen qui indique si la position courante du curseur se trouve après la dernière ligne
boolean isFirst()	booléen qui indique si le curseur est positionné sur la première ligne
boolean isLast()	booléen qui indique si le curseur est positionné sur la dernière ligne
boolean first()	déplacer le curseur sur la première ligne
boolean last()	déplacer le curseur sur la dernière ligne

boolean absolute()	déplace le curseur sur la ligne dont le numéro est fourni en paramètre à partir du début si il est positif et à partir de la fin si il est négatif. 1 déplace sur la première ligne, -1 sur la dernière, -2 sur l'avant dernière ...
boolean relative(int)	déplacer le curseur du nombre de lignes fourni en paramètre par rapport à la position courante du curseur. Le paramètre doit être négatif pour se déplacer vers le début et positif pur se déplacer vers la fin. Avant l'appel de cette méthode, il faut obligatoirement que le curseur soit positionné sur une ligne.
boolean previous()	déplacer le curseur sur la ligne précédente. Le booléen indique si la première occurrence est dépassée.
Boolean next()	déplacer le curseur sur la ligne suivante. Le booléen indique si la dernière occurrence est dépassée.
void afterLast()	déplacer le curseur après la dernière ligne
void beforeFirst()	déplacer le curseur avant la première ligne
int getRow()	renvoie le numéro de la ligne courante

Méthodes pour obtenir les données et la structure

Méthode	**Rôle**
getInt(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme d'entier.
getInt(String)	retourne le contenu de la colonne dont le nom est passé en paramètre sous forme d'entier.

getFloat(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de nombre flottant.
getFloat(String)	
getDate(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de date.
getDate(String)	
next()	se déplace sur le prochain enregistrement : retourne false si la fin est atteinte
Close()	ferme le ResultSet
getMetaData()	retourne un objet ResultSetMetaData associé au ResultSet.

Type de données JDBC (correspondance SQL et JAVA)

Type SQL	**Méthode ResultSet**	**Type Java**
ARRAY	getArray	java.sql.Array
BIGINT	getLong	long
BINARY	getBytes	byte[]
BIT	getBoolean	boolean
BLOB	getBlob	java.sql.Blob
CHAR	getString	java.lang.String
CLOB	getClob	java.sql.Clob
DATE	getDate	java.sql.Date
DECIMAL	getBigDecimal	java.math.BigDecimal
DINSTINCT	getTypeDeBase	typeDeBase

DOUBLE	getDouble	double
FLOAT	getDouble	double
INTEGER	getInt	int
JAVA_OBJECT	(type)getObject	type
LONGVARBINARY	getBytes	byte[]
LONGVARCHAR	getString	java.lang.String
NUMERIC	getBigDecimal	java.math.BigDecimal
OTHER	getObject	java.lang.Object
REAL	getFloat	float
REF	getRef	java.sql.Ref
SMALLINT	getShort	short
STRUCT	(type)getObject	type
TIME	getTime	java.sql.Time
TIMESTAMP	getTimestamp	java.sql.Timestamp
TINYINT	getByte	byte
VARBINARY	getBytes	byte[]
VARCHAR	getString	java.lang.String

Sources :

http://en.wikipedia.org/wiki/JDBC_driver#Type_4_Driver_-_Native-Protocol_Driver

http://download.oracle.com/javase/tutorial/jdbc/overview/architecture.html

Introduction à JDBC, de Denis Brunet

http://java.developpez.com/