JDBC MySQL

GUINKO Tonguim Ferdinand

13 octobre 2011

Sommaire

Définitions : Base de données, Table

Base de données

- Une base de données est un regroupement d'informations ;
- Structure de stockage de grandes quantités d'informations, dans un but précis, afin d'en faciliter l'exploitation.

Table

Composant de la base de données qui stocke les informations dans des enregistrements *(lignes)* et dans des colonnes *(champs)*.

En général les informations stockées dans une table le sont par catégorie.

Exemple : table des clients, table des fournisseurs, table des marques de voitures, table des livres etc.

Définitions : Enregistrement, champs, clé

Enregistrement

Ensemble des informations relatives à un élément donné la table.
Exemple : un enregistrement de la table des livres fournira toutes les informations sur un livre donné.

Champ

Chaque enregistrement est composé de plusieurs champs. Chaque champs d'un enregistrement contient une seule information sur l'enregistrement.
Exemple : un champs de la table livre contiendra le titre d'un livre donné ; un autre champs de cette table contiendra l'information sur le nom de l'auteur, etc.

Définitions : Clé, Clés candidates, Clé primaire

Clé

- Groupe minimum de champs ou d'attributs dont la connaissance de la valeur permet de connaître celle des autres attributs
- Identifie un enregistrement parmi les autres

Clés candidates

Plusieurs champs, *(attributs)* ou groupe de champs *(attributs)*, remplissent les critères d'une clé.

Clé primaire

Clé choisie parmi les clés candidates.

Définitions : Clé étrangère

Clé étrangère

Clé primaire `C` d'une table `A` apparaîssant parmis les attributs d'une table `B` : on dit que `C` de `B` référence `A` définit une *contrainte d'intégrité référentielle*.

Modèles de bases de données : modèle hiérarchique

Modèle hiérarchique

Forme de système de gestion de base de données qui lie des enregistrements dans une structure arborescente de façon à ce que chaque enregistrement n'ait qu'un seul possesseur.

Exemple : les fichiers et répertoires dans les systèmes d'exploitation de type Unix forment un exemple de base de données hiérarchique.

Si le principe de relation $1 \rightarrow N$ n'est pas respecté (par exemple, un étudiant peut avoir plusieurs professeurs et un professeur a, à priori, plusieurs étudiants), alors la hiérarchie se transforme en un réseau.

Modèle de bases de données : Modèle réseau

Modèle réseau

Réponds aux limites du modèle hierarchique grâce à la possibilité d'établir des liaisons de type $N \rightarrow N$.

Principal inconvénient : comme dans le cas des bases de données hiérarchiques, l'accès aux données est navigationnel et est totalement lié à la structure physique de la base.

Exemple : les banques utilisent le modèle de base de données en réseau pour stocker les signatures graphiques d'authentification de leurs clients. Aussitôt que la signature est stockée dans la base de données, elle est immédiatement disponible pour les succursales et agences de la banque qui sont branchées sur le même réseau qu'elle.

Modèle de bases de données : modèle objet

Modèle objet

L'apparition de la notion de base de données objets provient du fait qu'un ensemble de fonctionnalités n'était pas simultanément couvert par les langages objets, et les bases de données *classiques*.

Modèle de bases de données : modèle relationnel

Modèle relationnel

1. Basé sur la théorie mathématique de l'algèbre relationnel : dans cette théorie relationnelle, une relation est représentée par l'ensemble des lignes d'une table ;
2. Les relations (donc les tables) étant considérées au sens ensembliste, il n'existe aucune notion d'ordre au sein d'une relation. L'ordre des lignes dans une table est donc quelconque.

Modèle de bases de données : modèle relationnel

Modèle relationnel

1. Les relations sont manipulées en utilisant les différents opérateurs de l'algèbre relationnelle :
   - Sélection
   - Projection
   - Produit
   - Jointure
   - Union
   - Différence
   - Intersection
   - Division
2. La manipulation des données se font à l'aide du langage *Structured Query Language (SQL)*, qui implante l'ensemble des opérateurs de l'algèbre relationnelle.

Le langage SQL

Le langage SQL permet de dialoguer avec la base de données. Il existe différentes versions du langage SQL selon la base de données utilisées. Toutefois le langage dispose aussi d'une syntaxe élémentaire normalisée indépendante de toute base de données.

Le langage SQL : création

Le langage SQL : création

Syntaxe :

```
CREATE TABLE "nom de table"
("colonne 1" "type de données pour la colonne 1",
 "colonne 2" "type de données pour la colonne 2",
... )
```

Exemple :

```
CREATE TABLE temps
(ville varchar(80),
 t_basse int,
 t_haute int,
 prcp real,
 date date);
 PRIMARY KEY (ville));
```

Le langage SQL : suppression

Sxyntaxe :

```
DROP TABLE "nom de table"
```

Exemple :

```
DROP TABLE temps;
```

Le langage SQL : ajout et mise à jour

Le langage SQL : ajout d'informations dans une table

Syntaxe :

```
INSERT INTO "nom de table"
("colonne 1", "colonne 2", ...)
valeurS ("valeur 1", "valeur 2", ...)
```

Exemple :

```
INSERT INTO temps (ville, t_basse, t_haute, prcp, date)
VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
                ou
INSERT INTO temps VALUES
('Rimouski', 46, 50, 0.25, '1994-11-27');
```

Le langage SQL : mise à jour d'un enregistrement

Sxyntaxe :

```
UPDATE "nom de table"
SET "colonne 1" = [nouvelle valeur1],
    "colonne 2" = [nouvelle valeur2],
WHERE {condition}
```

Exemple :

```
UPDATE temps
SET t_basse = -20
WHERE ville = "Rimouski"
AND Date = "1994-11-27";
```

Le langage SQL : recherche d'information

L'instruction `select` est utilisée pour extraire un ou plusieurs champs spécifiques d'une ou plusieurs base de données.

Le langage SQL : interrogation d'une table

Syntaxe :

```
SELECT "nom de colonne"
FROM "nom de table"
```

Exemple :

```
SELECT ville, t_basse, t_haute, prcp, date FROM temps;

SELECT * FROM temps;

SELECT ville FROM temps;
```

Le langage SQL : recherche d'information

Une requête peut être *filtrée* en ajoutant une clause `WHERE` qui spécifie quelles lignes sont souhaitées. La clause `WHERE` contient une expression booléenne et seules les lignes pour lesquelles l'expression booléenne est vraie sont renvoyées.

Le langage SQL : la clause WHERE

Sxyntaxe :

```
UPDATE "nom de table"
SET "colonne 1" = [nouvelle valeur1],
    "colonne 2" = [nouvelle valeur2],
WHERE {condition}
```

Exemple :

```
SELECT * FROM temps WHERE ville = 'Québec'
AND prcp > 0.0
```

Le langage SQL : tri des informations

Le langage SQL : la clause ORDER BY

Sxyntaxe :

```
SELECT "nom de colonne"
FROM "nom de table"
[WHERE "condition"]
ORDER BY "nom de colonne" [ASC, DESC]
```

Exemple :

```
SELECT * FROM temps ORDER BY ville;

SELECT * FROM temps
ORDER BY ville, t_basse;

SELECT DISTINCT ville
FROM temps
ORDER BY ville DESC;
```

Pour aller plus loin avec le SQL :
`http://www.w3schools.com/sql/`.

MySQL

MySQL

- Un SGBD de type relationnel ;
- Un SGBD gratuit ;
- L'un des plus utilisés au monde ;
- Accepte le langage SQL ;
- Fonctionne sur les principaux systèmes d'exploitation.

JDBC

JDBC est la bibliothèque `java.sql.*` du langage `java` qui permet de dialoguer avec les serveurs de bases de données et ce :

- à travers une API (Application Programming Interface) et
- grâce au langage SQL (Structured Query Language).

Définition

Un pilote JDBC est un segment de code qui permet la connection avec un type de SGBD précis.

En effet chaque fournisseur de SGBD implémente les classes contenues dans le package `java.sql.*` pour que cela corresonde avec son produit.

JDBC

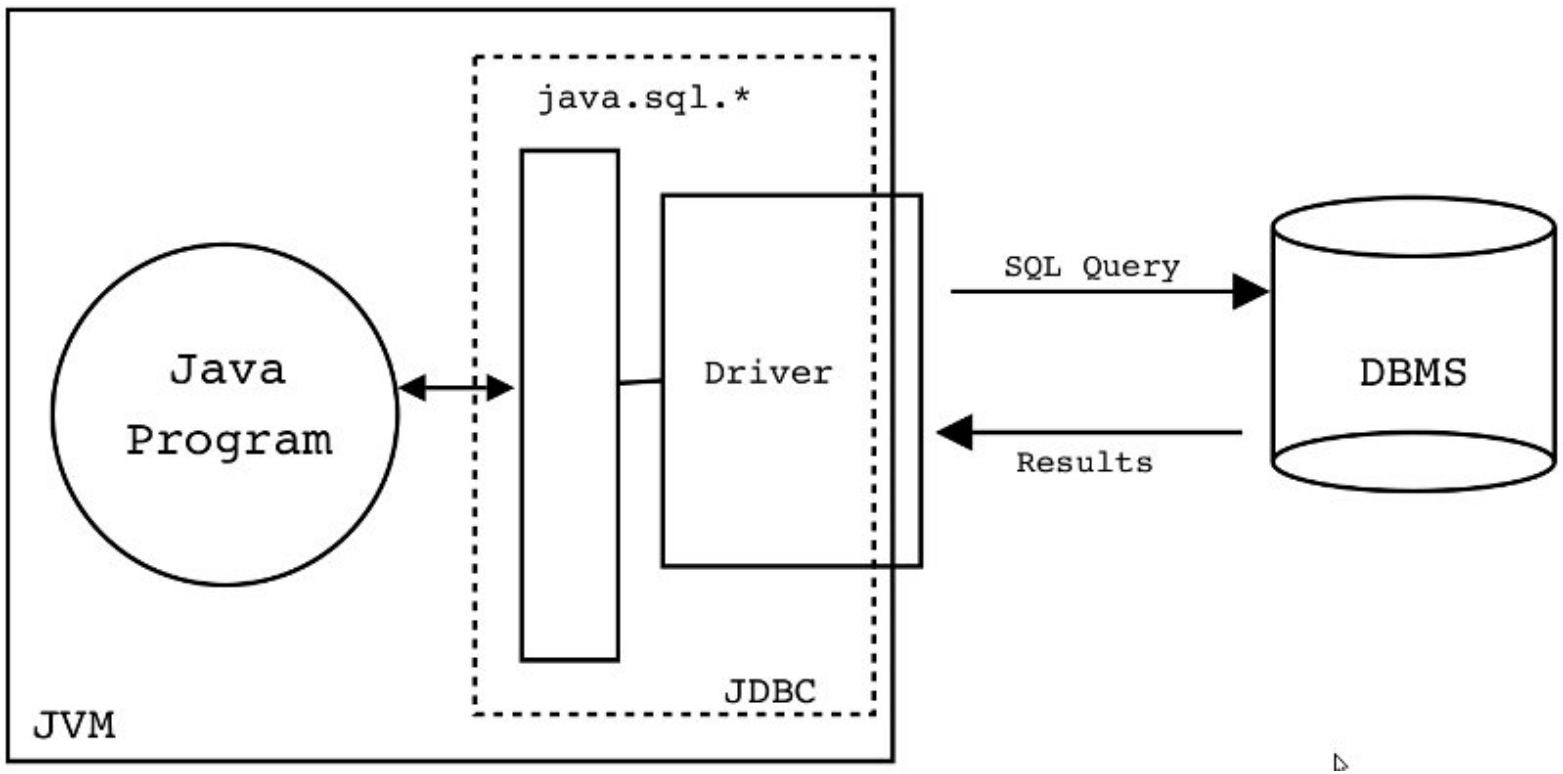

FIGURE: JDBC

JDBC

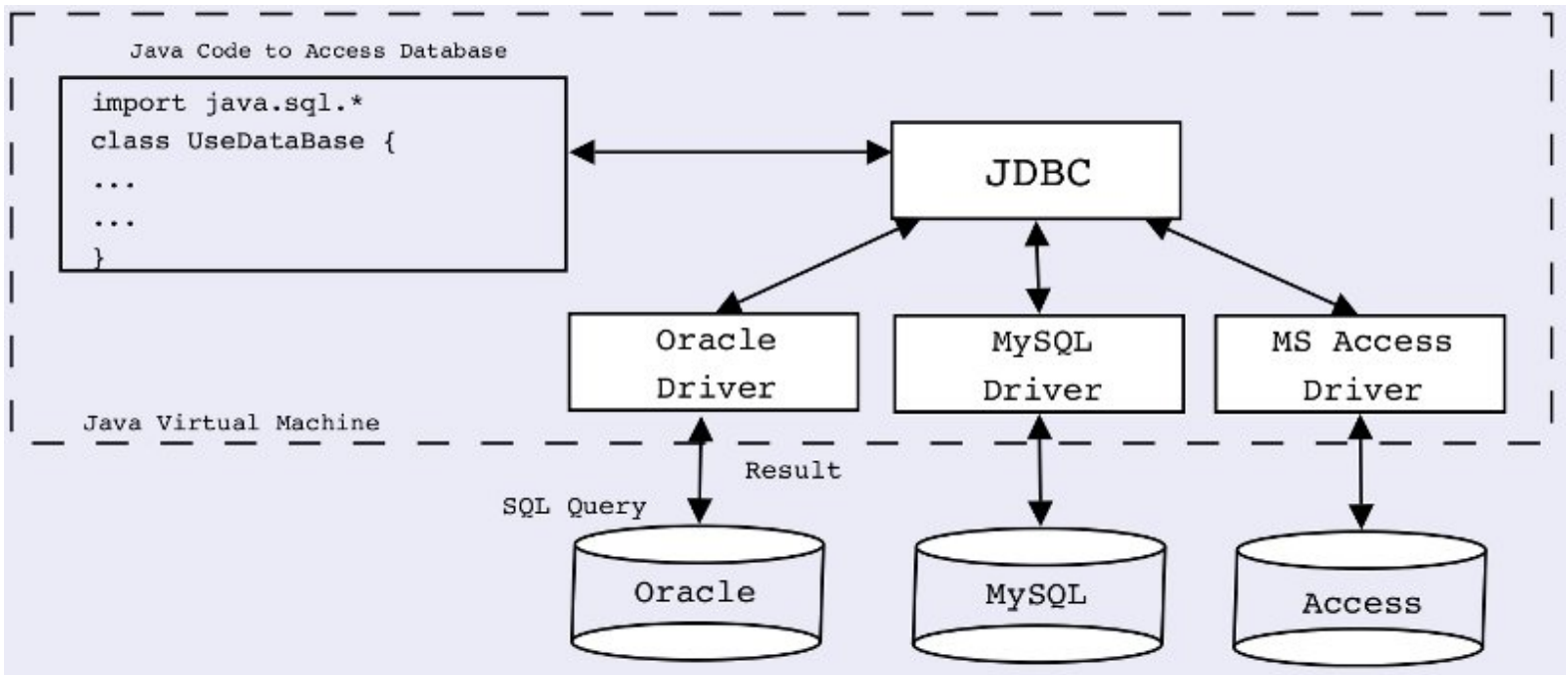

FIGURE: JDBC

JDBC : 4 types de pilotes

Il existe 4 types de pilotes JDBC :

1. pilote `JDBC-ODBC` ; traduit JDBC en ODBC et utilise le pilote ODBC pour communiquer avec le SGBD ; à utiliser lorsqu'un aucun autre pilote n'est disponible ;
2. pilote `natif` ; pilote partiellement écrit en Java et aussi dans un langage natif (C/C++) ; il communique avec l'API du SGBD ; plus efficace que le `JDBC-ODBC` ;
3. pilote `utilisant un serveur intermédiaire` pour communiquer avec le SGBD ; entièrement écrit en Java ;
4. pilote `entièrement écrit en Java` ; communique directement avec le SGBD ; il représente la solution idéale ; la majorité des pilotes sont de ce type.

JDBC : étapes de fonctionnement

Etapes de fonctionnement de JDBC :

1. Chargement du pilote spécific au SGBD utilisé ; classe `Driver` ;
2. Connection à la base de données ; classe `Connection` ;
3. construire et exécuter une requête ; classe `Statement` ;
4. Exécution de la requête ; classe `ResultSet` ;
5. Traiter le résultat de la requête ; classe `ResultSet` ;
6. Fermeture de la connexion.

Chargement du pilote

- C'est la première étape de fonctionnement de JDBC ; il s'agit de charger le pilote adapté au SGBD qui sera utilisé. Puisque nous utilisons le SGBD MySQL, nous utiliserons le pilote fourni par MySQL ;
- Ce pilote doit être chargé grâce à la méthode `forName` de la classe `class` . Cette méthode attend comme paramètre une chaîne de caractères contenant le nom du pilote ;
- Dans notre cas, cette classe porte le nom suivant :

  `com.mysql.jdbc.driver`
- Le chargement du pilote se fait à travers l'instruction suivante : `Class.forName("com.mysql.jdbc.driver")`

Chargement du pilote : forName

- L'instruction `forName` doit être protégée par un bloc `try catch` parce qu'elle est susceptible de déclencher une erreur de type `ClassNotFoundException` .

Exemple :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnexionDirecte
{
    public static void main (String[] args)
    {
        try
        {
            Class.forName("com.mysql.jdbc.driver");
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Connexion à la base de données : DriverManager ou DataSource

Un choix à faire !

Deux classes permettent de se connecter à la base de données ; il s'agit des classes

1. `DriverManager` et
2. `DataSource` .

Connexion à la base de données : DriverManager.getConnection : exemple

Exemple :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnexionDirecte
{
    public static void main (String[] args)
    {
        try
        {
            Class.forName("com.mysql.jdbc.driver");
        }
        catch(Exception e)
        {
            System.out.println("Erreur Pendant
                                le chargement du pilote");
            //e.printStackTrace();
        }
        Connection connectionDirecte=null;

        try
        {
            String dbURL =
               "jdbc:mysql://localhost/meteo";
            String utilisateur = "guinko";
            String motDePasse = "ui893djf";
            Conn connDirecte =
               DriverManager.getConnection(dbUrl,
                                           "login",
                                           "password");
        }
        catch(SQLException e)
        {
            System.out.println("Erreur Pendant la
                                connexion à la BD");
        }
    }
}
```

Connexion à la base de données : DriverManager.getConnection

1. La méthode `getConnection` de la classe `DriverManager` est chargée d'établir la connexion avec la base de données ;
2. `getConnection` attend comme paramètre une chaîne de caractères représentant une URL ; cette URL contient les informations nécessaires pour établir la connexion avec la base de données ;
3. Syntaxe de l'URL : `jdbc:nomDuProtocole:URL BD` ; c'est grâce au `nomDuProtocole` que `getConnection` identifie le pilote à utiliser pour se connecter au SGBD ;
4. En cas de succès, `getConnection` retourne une instance de classe implémentant l'interface `Connection` ;
5. En cas d'erreur de connexion, des exceptions sont levées : `SQLException`, `ClassNotFound`

Connexion à la base de données : Datasource

- A cause de la nature du langage Java, basé sur le concept d'objet, l'utilisation d'objets pour établir la connection avec la base de données accroît la portabilité de l'application ;
- Se baser sur le pilote d'une base de données spécifique pour communiquer avec une base de données peut entraver la portabilité d'une application ;
- La portabilité universelle de Java entraîne le fait que les applications doivent être le plus indépendantes possible des pilotes J2EE utilisés pour communiquer avec l'environnement ;
- Une application doit ignorer le type ou la version d'une base de donnée dont elle a besoin, etc.

Connexion à la base de données : Datasource

Solution ?

- Se référer à la base de données en utilisant un nom logique (exactement comme fonctionne le ≪Domain Name Service≫ (DNS)) plutôt que d'utiliser le pilote du concepteur du SGBD qui contient la base de données ;
- Laisser le serveur JEE, au lieu de l'application, se charger de gérer les connexions aux bases de données : le serveur fournira les objets `Datasource` nécessaires à cet effet ;
- Ces objets sont passés au code appelant, soit par requête `JNDI (Java Naming and Directory Interface)` .

Connexion à la base de données : Datasource : JNDI

Qu'est ce que `JNDI (Java Naming and Directory Interface)` ?

1. Solution proposée par J2EE pour exporter, partager, centraliser les paramètres d'une application dans le cadre d'un projet ou d'une application d'entreprise ;
2. Base de données `objet` arborescente dont chaque répertoire peut utiliser une technologie différente ;
3. `JNDI` est aux applications J2EE ce que le DNS est à Internet :
   - Le DNS trouve une information à partir d'un nom de domaine ;
   - Le DNS est hiérarchisé et comporte une convention de nomemclature ;
   - Le DNS établit une correspondance entre une adresse IP et un nom de domaine ; il résoud l'adresse IP en un nom convivial pour l'être humain ;
   - `JNDI` résoud les resources de J2EE (bases de données, objets java, etc.) en des noms conviviaux.

Connexion à la base de données : Datasource : JNDI : convention de noms

Convention des noms de JNDI :

Type de ressource	**Préfixe JNDI**
Variables d'environnement	java :comp/env/var%
URL	java :comp/env/url%
JavaMail Sessions	java :comp/env/mail%
Java Message Service	java :comp/env/jms%
EJB Homes	java :comp/env/ejb%
Datasources JDBC	java :comp/env/jdbc%

Connexion à la base de données : Datasource : JNDI : exemple 1

```
DataSource ds = null;
Connection conn = null;
try
{
    ds = ServiceLocator.getDataSource("java:comp/env/jdbc/DefaultDS");
    conn = ds.getConnection();
}

catch (SQLException ex)
{
    ...
}
catch (ServiceLocatorException e)
{
    ...
}
```

Dans cet exemple, JNDI a été utilisé à travers l'instruction suivante :

```
ds = ServiceLocator.getDataSource("java:comp/env/jdbc/DefaultDS");
```

Connexion à la base de données : Datasource : JNDI : exemple 2

```
import javax.naming.*;
import javax.sql.*;
public class ServiceLocator
{
    public static DataSource getDataSource(String dataSrcJndiName)
    throws ServiceLocatorException
    {
        DataSource ds = null;
        try
        {
            Context ctx = new InitialContext():
            DataSource ds = (DataSource) ctx.lookup(meteo);
            ds.setPort(1527);
            ds.setHost("localhost");
            ds.setUser("guinko")
            ds.setPassword("ui893djf");
            Connection conn = ds.getConnection();
        }
        catch // catch blocks snipped ...
        return ds;
    }
}
```

Les noms et les valeurs de JNDI sont stockés dans InitialContext()

Accéder à la base de données en utilisant JNDI : exemple 2

Dans le fichier web.xml :

```
<web-app>
    <servlet> ...
    <servlet-mapping> ...
    <resource-ref>
        <res-ref-name>jdbc/DefaultDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</web-app>
```

Connexion à la base de données : Datasource : exemple

Exemple avec DriverManager

```
public class ArchivesMeteo extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response) ...
    {
        Class.forName("com.mysql.jdbc.driver");
        Connection connection = DriverManager.getConnection("dbUrl,"login", "password");
        Statement statement = connection.createStatement();
        String query = "SELECT ville, t_basse, t_haute FROM temps";
        ResultSet resTemps = statement.executeQuery(query);
```

Exemple avec DataSource

```
public class ArchivesMeteoJNDI extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response) ...
    {
        DBUtils services = DBUtils.getInstance();
        Connection con = services.getConnection();
        Statement statement = con.createStatement();
        String query = "SELECT ville, t_basse, t_haute FROM temps";
        ResultSet resTemps = statement.executeQuery(query);
```

Construire et exécuter une requête SQL

java.sql.Statement

- modélise une requête SQL ;
- `executeQuery(String query)` : exécute la requête et retourne un `ResultSet (SELECT)` ;
- `executeUpdate(String query)` : renvoie le nombre de lignes modifiées `(INSERT, DELETE, UPDATE)` ;
- `execute(String query)` à utiliser dans les deux cas.

construire et exécuter une requête SQL

java.sql.PreparedStatement

- consiste à mettre en place un plan d'accès aux tables ;
- la classe `PreparedStatement` optimise les traitements en permettant d'établir une seule fois le plan d'accès ;
- pour l'exécution de plusieurs requêtes.

Traiter et exécuter une requête SQL

java.sql.ResultSet

- encapsule le résultat des résultats ;
- représente un ensemble de résultats (lignes) extraites de la base de données ;
- permet d'accéder aux lignes et au colonnes de la table ;
- méthodes
  - `next` : pour l'itération
  - `getType(position|name)` : pour accéder aux données.

Exemple 1 :

```
String ville;
float t_basse;
Statement st = connection.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM Temps");
while (rs.next())
{
    ville = rs.getString("ville");
    t_basse = rs.getString("t_basse");
}
```

Exemple 2

```
<HTML><HEAD><TITLE>Températures enregistrées</TITLE></HEAD>
<%@ page import="java.io.*, java.sql.*"%>
<BODY><CENTER>
<H3>List of Cars</H3>
<TABLE BORDER="1">
<TR><TH>Ville</TH><TH>Temp basse</TH><TH>Temp haute</TH><TH> </TH></TR>
<%
try
{
    Class.forName("com.mysql.jdbc.driver");
    Connection connection = DriverManager.getConnection(dbUrl, "login", "password");
    Statement statement = connection.createStatement();
    String query_car = "SELECT ville, t_basse, t_haute FROM Temps";
    ResultSet resTemps = statement.executeQuery(query_temps);
    while(resTemps.next())
    {
        String ville = resTemps.getInt("ville");
        float t_basse = resTemps.getString("t_basse");
        float t_haute = resTemps.getInt("t_haute");
        %>
        <TR><TD ALIGN="center"><%= ville %></TD>
        <TD><%= t_basse %></TD>
        <TD ALIGN="right"><%= t_haute %></TD>
        <TD ALIGN="center"><A HREF='temperatures.jsp?temps=<%= ville %>'>DEL</A></TD>
        </TR>
        <%
    }// fin du while
```

Exemple 2

```
        <%
    }// fin du while
} catch (ClassNotFoundException cnfe)
{// ...}
%>
</TABLE></CENTER></BODY></HTML>
```

Exemple 3

```
<%@ page import="java.io.*, java.sql.*"%>
<HTML><HEAD><TITLE>Archives des températures</TITLE></HEAD>
<BODY><CENTER><H3>Mise à jour de la base de données des températures</H3>
<%
final String DB_DRIVER_CLASS = "com.mysql.jdbc.driver
";
final String DB_URL = "jdbc:mysql://localhost/meteo";
try
{
    Class.forName(DB_DRIVER_CLASS);
    Connection con = DriverManager.getConnection(DB_URL);
    Statement statement = con.createStatement();
    String updateCMD = "UPDATE ville "+
    "SET t_haute = 1.1*t_haute "+
    "WHERE Make = 'Québec'";
    int rowsDone = statement.executeUpdate(updateCMD);
    %>
    Nombre d'enregistrment mis à jour: <%= rowsDone %>
    <%
    con.close();
}
    catch (ClassNotFoundException cnfe)
{
    // catch blocks snipped ...}
%>
<P><A HREF='temperatures.jsp>Retour à la liste des températures</A></CENTER></BODY></HTML>
```

Fermer la connexion

```
public class ServiceLocator
{
    public static DataSource getDataSource(String dataSrcJndiName)
    throws ServiceLocatorException
    {
        DataSource ds = null;
        try
        {
            Context ctx = new InitialContext():
            DataSource ds = (DataSource) ctx.lookup(meteo);
            ds.setPort(1527);
            ds.setHost("localhost");
            ds.setUser("guinko")
            ds.setPassword("ui893djf");
            Connection conn = ds.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM Temps");

        }
        catch // catch blocks snipped ...
        return ds;

        ...
        rs.close();
        stmt.close();
        conn.close();
    }
```

Les transactions

Une transaction est un groupe d'opérations liées l'une à l'autre et devant être exécutées en une seule opération de telle sorte que la non exécution d'une seule opération entraîne l'échec des autres.

Exemple :
Considérons les 2 opérations bancaires suivantes :

1. Retrait d'argent du compte A ;
2. Dépôt d'argent dans le compte B.

Ces 2 opérations doivent être exécutées successivement, au risque de perdre l'argent retiré si le retrait s'est bien déroulé, tandis que le dépôt à échoué.

Les transactions : propriétés dite ACID

Les transactions doivent répondre à 4 propriétés essentielles :

1. `Atomicité` : Chacune des opérations qui constituent la transaction doit s'exécuter entièrement, ou échouer ;
2. `Consistence` : l'architecture de la transaction doit permettre une mise à jour effisciente de la base de données ;
3. `Isolation` : les données ne doivent pas être corrompues par des accès concurrents à des sources de données différentes ; les transactions doivent donc être isolées ou séparées jusqu'à ce qu'elles s'exécutent complètement ;
4. `Durabilité` : s'assurer que la base de données a été bien mise à jour, après l'exécution de la transaction.

Les transactions

Exemple :

```
try
{
    Connection myConnection = dataSource.getConnection();
    // autoCommit est initialisé à faux
    myConnection.setAutoCommit(false);
    retraitdArgentCompteA(...); //opération 1
    depotArgentCompteB(...); //opération 2
    myConnection .commit();
}
```

```
catch(Exception sqle)
{
    try
    {
        myConnection .rollback();
    } catch( Exception e){}
}

finally
{
    try
    {
        if( conn != null)
        {
            conn.close();
        }
    } catch( Exception e) {}
}
```

Dans cet exemple, l'on s'assure de ce que les opérations 1 et 2 réussissent ou échouent, et s'exécutent comme des opérations atomiques.

- valider : `connection.commit()` ou
- invalider : `connection.rollback()`

Outils

jspMyAdmin