Java et les bases de données
L'API JDBC

Java est un excellent candidat pour le développement d'applications de bases de données

- **Robuste et sécurisé**
- **Facile à comprendre**
- **Automatiquement téléchargeable par le réseau**

Avant JDBC, il était difficile d'accéder à des bases de données SQL depuis

Java.

Objectifs :

Permettre aux programmeurs de java d'écrire un code indépendant de la base de données et du moyen de connectivité utilisé

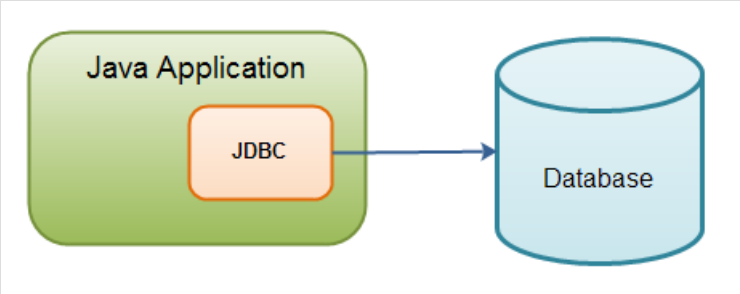
Qu'est ce que JDBC ?

JDBC(Java DataBase Connectivity)

- API Java adaptée à la connexion avec les bases de données relationnelles (SGBDR)
- Fournit un ensemble de classes et d'interfaces permettant l'utilisation

  sur le réseau d'un ou plusieurs SGBDR à partir d'un programme Java

JDBC permet de communiquer avec plusieurs bases de données variées.



- Très grosses bases de données : Oracle, Informix, Sybase,…
- Petites bases de données : FoxPro, MS Access, mSQL.
- Il permet également de manipuler des fichiers textes ou les feuilles de

calculs Excel.

- Liberté totale vis a vis du constructeurs

API JDBC

JDBC fournit un ensemble complet d'interfaces qui permet un accès mobile à une base de données. Java peut être utilisé pour écrire différents types de fichiers exécutables, tels que :

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs)

API JDBC

Est fournit par le package java.sql

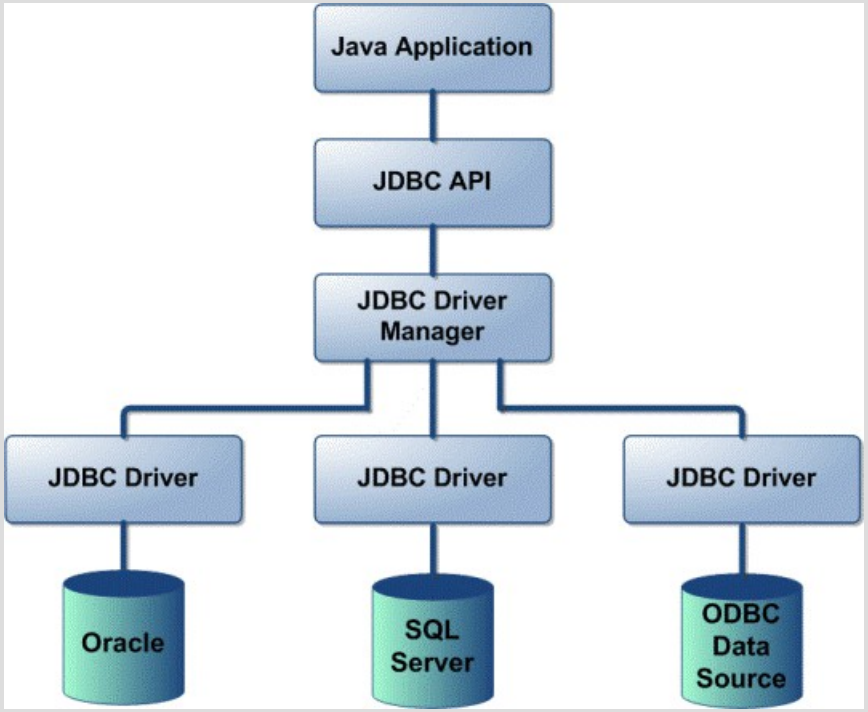
permet de formuler et de gérer les requêtes aux bases de données relationnelles.

- supporte le standard SQL-2 Entry Level
- **8 interfaces :**
  - **Connexion à une base de données éloignée**
  - **Création et exécution de requêtes SQL**
  - **Statement**
  - **CallableStatement, PreparedStatement**
  - **DatabaseMetaData, ResultSetMetaData**
  - **ResultSet**
  - **Connection**
  - **Driver**

Principe de fonctionnement

Chaque base de données utilise un pilote (driver) qui est propre et qui permet de convertir les requêtes JDBC dans un langage natif du SGBDR.

Ces drivers dits JDBC existent pour tous les principaux constructeurs



Architecture JDBC

Un modèle à 2 couches

- **La couche externe : API JDBC**

C'est la couche visible pour développer des applications java accédant à des

SGBD.

- **Les couches inférieurs :**

- Destinées à faciliter l'implémentation de drivers pour des bases de données.

- Représentent une interface entre les accès de bas niveau au moteur du SGBDR et la partie applicative

Mettre en oeuvre JDBC

- Importer le package java.sql
- Enregistrer le driver JDBC
- Etablir la connexion à la base de données
- Créer une zone de description de requête
- Exécution des commandes SQL
- Inspection des résultats (si disponible)
- Fermer les différents espaces

Enregistrer le driver JDBC -
URL de Connexion à la base de données

Sur SQLDeveloper à la connexion :

- **Nom de connexion : ...**
- **Login : i1....**
- **Passwd : i1...**
- **Nom du serveur : soracle2**
- **Port : 1521**
- **Db1**

en JAVA :

DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

conn = DriverManager.getConnection("**jdbc:oracle:thin:@soracle2:1521:DB1**",user, passwd);

Avec **conn** de type *Connection*

Exemple

```
public class SessionOracle {
Connection conn ;
public SessionOracle(String user, String passwd) t {
Try {
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
conn = DriverManager.getConnection("jdbc:oracle:thin:@soracle2:1521:DB1",user, passwd) ;
conn.setAutoCommit(true);
System.out.println("Connexion OK !") ;
} catch(SQLException e) {
conn = null; System.out.println("Erreur de connexion") ;
}
Public Connection getConnetion() { return conn; }
}
```

Statement

Une fois le connexion établie, pour pouvoir envoyer des requêtes SQL, il faut obtenir une instance de Statement sur laquelle on invoque des méthodes.

3 types de Statement :

Statement : requêtes statiques simples

PreparedStatement : requêtes dynamiques pré compilées

(avec paramètres E/S)

CallableStatement : procédures stockées.

A partir de l'instance de l'objet Connection, on récupère le statement associé :

Statement

Statement req1 = conn.**createStatement**() ;

PreparedStatement req2= conn.**prepareStatement**(str);

CallableStatement req3 =conn.**prepareCall**(str)

Exécution d ’une requête

3 types d’exécution :

executeQuery() : pour les requêtes (SELECT) qui retournent un ResultSet (tuples résultants).

executeUpdate() : pour les requêtes (*INSERT,UPDATE, DELETE, CREATE TABLE, DROP TABLE*) qui retournent un entier (nombre de tuples traités).

execute() : pour les procédures stockées.

Les résultats sont retournés dans une instance de la classe ***ResultSet***. Ce résultat est constitué d'une suite de lignes, le passage à la ligne suivante se fait par la méthode ***next*** de la classe *ResultSet*.

Exemple

```
Statement stmt = conn.createStatement();
ResultSet resultat = stmt.executeQuery("SELECT Nom, Age FROM Etudiant;");
while (resultat.next())
   {
        String nom = resultat.getString("Nom");
       int age = resultat.getInt("Age");
      System.out.println("Nom :" nom + " Age :" + age);
  }
int nb = stmt.executeUPDATE("INSERT INTO ETUDIANT VALUES('toto', 26) ")
```

Exécution d ’une requête

- Le code SQL n’est pas interprété par java. C’est le pilote associé à la connexion (et au final par le moteur de la base de données) qui interprète la requête SQL. Si une requête ne peut s’exécuter ou qu’une erreur de syntaxe SQL à été détectée, l’exception **SQLException** est levée.

- Le driver JDBC effectue d’abord un accès à la base pour découvrir les types des colonnes impliquées dans la requête puis un 2ème pour l’exécuter.

Traitement des types retournés

- On peut parcourir le *ResultSet* séquentellement avec la méthode : ***next***()
- Les colonnes sont référencés par leur numéro (commencent à 1) ou par leur **nom**.
- L'accès aux valeurs des colonnes se fait par les méthodes des la forme **getXXX**().

Lecture du type de données XXX dans chaque colonne du tuple courant

Types de données JDBC

- Le driver JDBC traduit le type JDBC retourné par le SGBD en un type **java** correspondant.
- Le XXX de getXXX() est le nom du type Java correspondant au type JDBC attendu.

Chaque driver a des correspondances entre les types SQL du SGBD.

Le programmeur est responsable du choix de ces méthodes.

SQLException est générée si mauvais choix

Les correspondances entre type SQL et type Java sont données par les spécifications JDBC.

SQL type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
INTEGER	int
BIGINT	long

Cas des valeurs nulles

Pour repérer les valeurs NULL de la base :

Utiliser la méthode wasNull() de ResultSet

Revoie **true** si l'on vient de lire NULL, false sinon.

Les méthodes **getXXX**() convertissent une valeur NULL SQL en une valeur acceptable par le type d'objet demandé :

- Les méthodes (**getString**,…) peuvent retourner un null java.
- Les méthode numeriques (**getInt**(), …) retournent 0.
- **getBoolean**() retourne false.

Fermer les différents espaces

Pour terminer proprement un traitement, il faut fermer les différents espaces ouverts sinon le garbage collector s'en

occupera mais moins efficacement.

Chaque objet possède une méthode **close**() ;

```
resultset.close() ;

statement.close() ;

connection.close();
```

Utilité des BEAN

- Chaque **Bean** correspond à une table
- La table employe en java :

public class Employe

{ int le_numempl; String le_nomempl;....

public Employe(int numempl, String nomempl, ...) {

this.numempl = le_numempl;

this.nomempl = le_nomempl;...

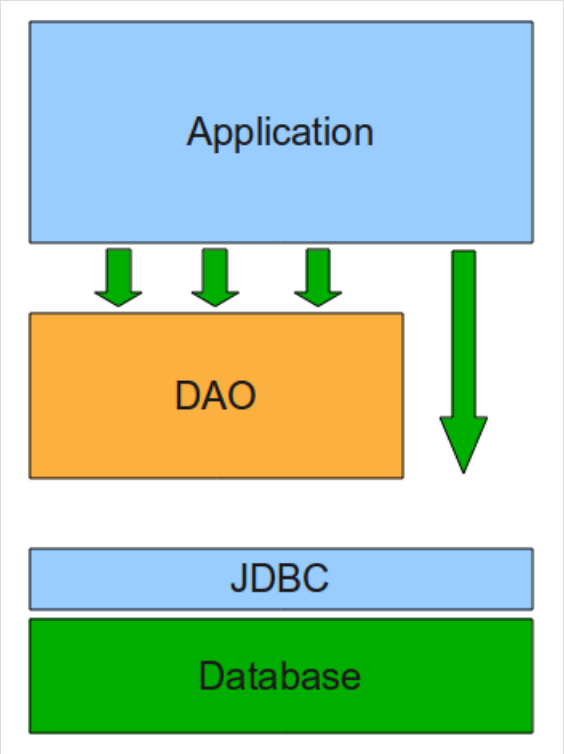
}

Getter + Setter de chaque attribut

}

Utilité des DAOs

- Plus facile de modifier le code qui gère la persistance (changement de SGBD ou même de modèle de données)
- Factorise le code d'accès à la base de données plus facile pour le spécialiste des BD d'optimiser les accès (ils n'ont pas à parcourir toute l'application pour examiner les ordres SQL)



CRUD

Les opérations de base de la persistance :

- **Create**
- **Retrieve**
- **Update**
- **Delete**

```
public class EmployeDAO {
private SessionOracle session;

public DAOEmploye(SessionOracle s) {
                this.session= s;
}
```

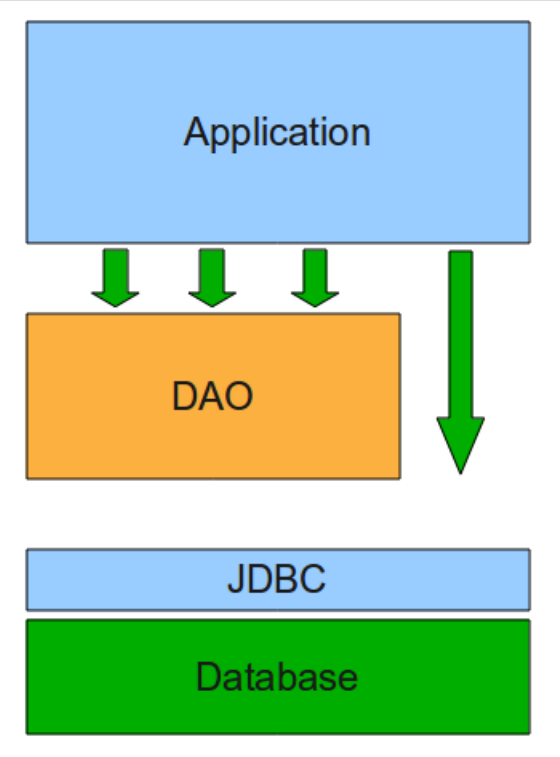
```
public void create(employe em) {

Try {

String insertTableEmploye= "INSERT INTO EMPLOYE VALUES( " + em.getnuempl()+","
+ em.getnomempl()+ ","+.... )";

Statement stmt = session.getConnection().createStatement();

stmt..executeUpdate(insertTableEmploye);

    }

catch (SQLException erreur) {

System.out.printl(erreur.getErrorCode() + "     " +e.getMessage() ;

// getErrorCode : PK, FK, Check, etc..., -0001,


     }

    }
```

```
public void read() {
Try {
String selectTableEmploye= "select * from employe)";
Statement stmt = session.getConnection.createStatement();
ResultSet resultat =stmt..executeQuery(selectTableEmploye);
While (resultat.next()){
  Employe em = new Employe(resultat.getInt(1), getString(2),.....)
 System.out.prinln(em.toStrin()) ;
}
catch (SQLException erreur) {
System.out.printl(erreur.getErrorCode() + "     " +e.getMessage() ;
    }
  }
```

```
public delete (employe em)
{
}
public update(employe em)
{
}
....
}
```

Classe Application

```
import BEAN.*; import DAO.*
public class Appli{
public static void main(String[] args) {
Try {
SessionOracle s = new SessionOracle("i1a01a", "i1a01a") ;
EmployeDAO  ED = new EMPLOYEDAO(s);
ED.read() ;
e1=new Employe(.....);// créer une instance d'mployé
ED.create(e1) ;
E1.read() ;
}
}
```



Requêtes pré-compilées

L'objet **PreparedStatement** envoi une requête sans paramètres à la base de données pour pré-compilation et spécifiera le moment voulu la valeur des paramètres .

plus rapide qu'un **Statement** classique.

le SGBD n'analyse qu'une seule fois la requête, pour de nombreuses exécutions d'une même requête SQL avec des paramètres variables.

PreparedStatement ps=session.getConnection().prepareStatement("SELECT * from Clients" + "where name = ?");

Les **arguments dynamiques** sont specifies par un "?"

Ils sont ensuite positionnés par les méthodes setInt(), setString(), … de PreparedStatement.

Requêtes pré-compilées

SetNull() positionne le paramètre à NULL de SQL

Ces méthodes nécessitent 2 arguments

- Le premier (int) indique le numéro relatif de l'argument dans la requête.

- Le second indique la valeur à positionner.

Exemple :

```
PreparedStatement ps = session.getConnection().prepareStatement ("UPDATE
empl set sal= ? where name = ?");

For (int i=0; i<10;i++) {
                    ps.setFloat(1, salary[i]);
                    ps.setString(2, name[i]);
count = ps.executeUpdate(); }
```

Exemple

```
public void create(employe em) {
Try {
String insertTableEmploye= "INSERT INTO EMPLOYE VALUES( ?, ?, ?, ?, ?)";
Preparedement ps= session.getConnection().prepareStatement(insertTableEmploye) ;
ps.setInt(1, em.getNuempl()) ;
ps.setString(2, em.getNomempl()) ;….
ps.executeUpdate();
    }
catch (SQLException erreur) {
System.out.printl(erreur.getErrorCode() + "     " +e.getMessage() ;
// getErrorCode : PK, FK, Check, etc..., -0001, -2291,-2292,+ erreurs Triggers
    } }
```

Accès aux procédures stockées

```
public class EmployeDAO {
public void create(employe em)  {
try {
   callstmt = connect.prepareCall("call MAJ.creer_un_employe(?,?,?, ?, ?)");
   callstmt.setInt(1,s.getNumempl());
  callstmt.setString(2, s.getNomempl());+ les autres attributs.
   callstmt.execute();
    } catch (SQLException erreur) {
System.out.printl(erreur.getErrorCode(), erreur.getMessage()) ; //
//getMessage() correspond aux messages d'erreurs de la partie exception de la procédure
//getErrorCode() = 20XXX. Code erreur retourné par la procédure.
    }}
```

Accès aux procédures stockées

Accès aux données :

```
public void read() {
Try { CallableStatement callstmt =
        session.getConnection().prepareCall("call exemple.liste_employe( ?,?)");
callstmt.setInt(1,0) ; / / 1 correspond au 1er point ? Et 0 la valeur associée
callstmt.registerOutParameter(2,OracleTypes.CURSOR);
callstmt.execute();
ResultSet rset = (ResultSet)callstmt.getObject(2);
While (rset.next()){
  Employe em = new Employe(resultat.getInt(1), getString(2),.....)
 System.out.prinln(em.toString()) ;
} catch (SQLException erreur) { System.out.printl(erreur.getErrorCode() + "     "
+e.getMessage() ; }
```

```
public class EmployeDAO {

public void create(employe em)  {

try {

   callstmt = connect.prepareCall("call MAJ.creer_un_employe(?,?,?, ?, ?)");

   callstmt.setInt(1,s.getNumempl());

  callstmt.setString(2, s.getNomempl());+ les autres attributs.

   callstmt.execute();

    } catch (SQLException erreur) {

System.out.printl(erreur.getErrorCode(), erreur.getMessage()) ; //

//getMessage() correspond aux messages d'erreurs de la partie exception de la
procédure

//getErrorCode() = 20XXX. Code erreur retourné par la procédure.

     }}
```