

# JDBC

## Java DataBase Connectivity

---

---

---

---

---

---

---

---

# JDBC

## Introduction

- Offre une API unique d'accès à toute BD conforme au standard SQL-92
- Objectifs :
  - Fournir un accès homogène aux SGBD
    - une application Java est capable d'accéder de façon générique à un SGBD quel que soit son fournisseur
  - Abstraction des SGBD cibles
  - Requêtes SQL
  - Simple à mettre en oeuvre
  - JDBC 1.0 Core API (JDK 1.1) : package `java.sql`
  - JDBC 2.0 (Java 2)

---

---

---

---

---

---

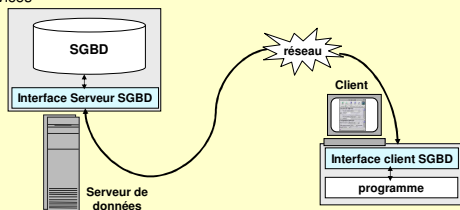
---

---

# JDBC

## Architectures Client/Serveur

- Client/Serveur : un programme **client** s'adresse à un programme sur une machine distante (le **serveur**) pour échanger des informations et des services



- Programme client ne communique pas directement avec SGBD
- Sur le poste client : interface client du SGBD qui gère le protocole de communication spécifique au SGBD
- Sur le poste serveur : interface serveur du SGBD qui gère les connexions avec les différents clients.

---

---

---

---

---

---

---

---

## JDBC Architecture d'une application JDBC

- JDBC permet de développer des programmes Java clients (applications autonomes, applets composants dans applications N-tiers) qui accèdent à des SGBD
- Package java.sql : implémentation de la spécification JDBC fournie en standard avec le JDK
- DriverManager : classe java à laquelle s'adressent les autres objets de l'application cliente
- JDBC interagit avec le SGBD par un driver
- Il existe des drivers pour Oracle, Sybase, Informix, DB2, ...
- JDBC spécifie uniquement l'architecture que ces drivers doivent respecter. Ils sont réalisés par une tierce partie (fournisseur du SGBD, éditeur de logiciel...)
- l'implémentation des drivers est totalement libre

© Philippe GENOUD UJF Février 2004 4

---

---

---

---

---

---

---

---

---

---

## JDBC Services JDBC

- Gestion des drivers
  - chargement, sélection
- Ouverture de connexions à une base de données
- SQL dynamique et SQL statique
  - exécution de requêtes (SQL-92)
- Exploitation des résultats
  - correspondance types SQL-types JAVA
- Accès au méta-modèle
  - informations sur les possibilités du driver
  - description des objets du SGBD

© Philippe GENOUD UJF Février 2004 5

---

---

---

---

---

---

---

---

---

---

## JDBC Classes et interfaces de JDBC

© Philippe GENOUD UJF Février 2004 6

---

---

---

---

---

---

---

---

---

---





## JDBC

### Préparer / exécuter une requête simple

- Création d'un *statement* :

```
Statement stmt = conn.createStatement();
```

- Exécution de la requête :

```
String myQuery = "SELECT prenom, nom, email " +  
                "FROM employe " +  
                "WHERE (nom='Dupont') AND (email IS NOT NULL) " +  
                "ORDER BY nom";  
ResultSet rs = stmt.executeQuery(myQuery);
```

- `executeQuery()` renvoie un objet de type `ResultSet`
  - permet de décrire la table des résultats

---

---

---

---

---

---

---

---

---

---

## JDBC

### Lecture des résultats

- `executeQuery()` renvoie un objet de classe `ResultSet`
  - permet de décrire la table des résultats

```
java.sql.Statement stmt = conn.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT nom, code_client FROM Clients");
```

Nom	Prénom	Code_client	Adresse
DUPONT	Jean	12345	135 rue du Lac
DUBOND	Louise	12345	15 avenue de la Mer
...	...	...	...
ZORG	Albert	45677	8 Blvd De la Montagne



Nom	Code_client
DUPONT	12345
DUBOND	12345
...	...
ZORG	45677

- Les rangées du `ResultSet` se parcourent itérativement ligne (*row*) par ligne
  - `boolean next()` permet d'avancer à la ligne suivante, → `false` si pas de ligne suivante
  - Placé avant la première ligne à la création du `ResultSet`

```
while (rs.next())  
{  
    ... Exploiter les données  
}
```

---

---

---

---

---

---

---

---

---

---

## JDBC

### Lecture des résultats

- Les colonnes sont référencées par leur numéro ou par leur nom
- L'accès aux valeurs des colonnes se fait par des méthodes `get.xxx(String nomCol)` ou `get.xxx(int numCol)` où `xxx` représente le type de l'objet
  - Pour les très gros row, on peut utiliser des streams.

```
java.sql.Statement stmt = conn.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");  
while (rs.next())  
{  
    int i = rs.getInt("a"); // rs.getInt(1);  
    String s = rs.getString("b"); // rs.getString(2);  
    byte b[] = rs.getBytes("c"); // rs.getBytes(3);  
    System.out.println("ROW = " + i + " " + s + " " + b[0]);  
}
```

Attention ! En SQL les numéros de colonnes débutent à 1

---

---

---

---

---

---

---

---

---

---

# JDBC

## Equivalences des types Java-SQL

- Pour chaque méthode `get.xxx` le driver JDBC doit effectuer une conversion entre le type de données de la base de données et le type Java correspondant

Type SQL	Méthode	Type Java
CHAR	<code>getString</code>	String
VARCHAR	<code>getString</code>	String
NUMERIC	<code>getBigDecimal</code>	<code>java.Math.BigDecimal</code>
DECIMAL	<code>getBigDecimal</code>	<code>java.Math.BigDecimal</code>
BIT	<code>getBoolean</code>	boolean Boolean
TINYINT	<code>getByte</code>	byte Integer
SMALLINT	<code>getShort</code>	short Integer
INTEGER	<code>getInt</code>	int Integer
BIGINT	<code>getLong</code>	long Long
REAL	<code>getFloat</code>	float Float
FLOAT	<code>getDouble</code>	double Double
DOUBLE	<code>getDouble</code>	double Double
DATE	<code>getDate</code>	<code>java.sql.Date</code>
TIME	<code>getTime</code>	<code>java.sql.Time</code>
TIME STAMP	<code>getTimeStamp</code>	<code>java.sql.Timestamp</code>

Peut être appelée sur n'importe quel type de valeur

`getObject` peut retourner n'importe quel type de donnée « package » dans un objet java (object wrapper)

Si une conversion de données invalide est effectuée (par ex DATE -> int), une `SQLException` est lancée

---

---

---

---

---

---

---

---

---

---

# JDBC

## Préparer/exécuter une requête simple

- Un objet `Statement` représente une simple (seule) requête SQL.
  - Un appel à `executeQuery()`, `executeUpdate()` ou `execute()` ferme implicitement tout `ResultSet` actif associé avec l'objet `Statement`.
  - Avant d'exécuter une autre requête avec un objet `Statement` il faut être sûr d'avoir exploité les résultats de la requête précédente.

```
Statement stmt = conn.createStatement();
ResultSet rs1 = stmt.executeQuery(myQuery1);
ResultSet rs2 = stmt.executeQuery(myQuery2);
//exploitation des résultats de myQuery1
while (rs1.next() {
    ...
}
//exploitation des résultats de myQuery2
while (rs2.next() {
    ...
}

Statement stmt = conn.createStatement();
ResultSet rs1 = stmt.executeQuery(myQuery1);
//exploitation des résultats de myQuery1
while (rs1.next() {
    ...
}
ResultSet rs2 = stmt.executeQuery(myQuery2);
//exploitation des résultats de myQuery2
while (rs2.next() {
    ...
}
```

- Si application nécessite d'effectuer plus d'une requête simultanément, nécessaire de créer et utiliser autant d'objets `Statement`.

---

---

---

---

---

---

---

---

---

---

# JDBC

## Préparer/exécuter une requête précompilée

- Création d'un `preparedStatement` (requête SQL dynamique):
  - paramètres formels spécifiés à l'aide de ?

```
PreparedStatement ps = conn.prepareStatement (
    "SELECT * FROM ? WHERE NAME = ? "
);
```

Dès que l'objet est instancié, la procédure SQL est transmise au SGBD qui la compile

- Passage des paramètres effectifs
  - à l'aide de méthodes au format `setXXX(indice,valeur)` où `XXX` représente le type du paramètre

```
ps.setString(1, "Person" );
```

- Invocation et exploitation des résultats
  - phase identique à celle utilisée pour SQL statique

```
for (int i=0; i < names.length; i++) {
    ps.setString(2, names[i] );
    ResultSet rs = ps.executeQuery();
    // ... Exploitation des résultats
}
```

---

---

---

---

---

---

---

---

---

---

## JDBC

### Procédures stockées

- La plupart des SGBD incluent un langage de programmation interne (ex: PL/SQL d'Oracle) permettant aux développeurs d'inclure du code procédural dans la BD, code pouvant être ensuite invoqué depuis d'autres applications.
  - le code est écrit une seule fois et peut être utilisé par différentes applications.
  - permet de séparer le code des applications de la structure interne des tables. (cas idéal : en cas de modification de la structure des tables seul les procédures stockées ont besoin d'être modifiées)
- Utilisation des procédures stockées depuis JDBC indépendante de la manière dont celles-ci sont gérées par le SGBD
- Utilisation possible de la valeur de retour
- Gestion des paramètres IN, OUT, INOUT

---

---

---

---

---

---

---

---

## JDBC

### Procédures stockées

- Préparation de l'appel
  - Appel avec valeur de retour et paramètres

```
CallableStatement proc = conn.callableStatement(" {? = call maProcédure(?,?) }");
```
  - Appel sans valeur de retour et avec paramètres

```
CallableStatement proc = conn.callableStatement(" call maProcédure(?,?) ");
```
- Préparation des paramètres

```
proc.registerOUTParameter(2, Types.DECIMAL, 3);
```

2ème paramètre de type OUT      Nombre de chiffres après décimale
- Passage des paramètres IN

```
proc.setByte(1, 25);
```

1er paramètre (type IN)      valeur
- Appel

```
ResultSet rs = proc.executeQuery();
```
- Exploitation du ResultSet (idem que pour Statement et PreparedStatement)
- Récupération des paramètres OUT

```
java.Math.BigDecimal bigd = proc.getBigDecimal(2,3);
```

---

---

---

---

---

---

---

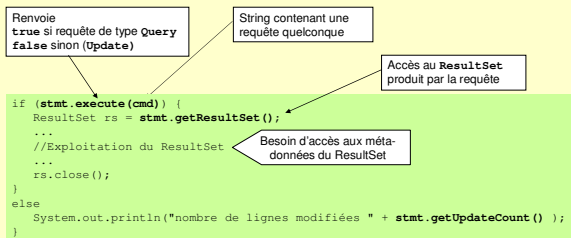
---

## JDBC

### Accès aux méta-données

- Permet de découvrir dynamiquement (au moment de l'exécution) des propriétés concernant la base de données ou les résultats de requêtes
- Exemple : lors de l'exécution d'une requête non connue à l'avance.

```
if (stmt.execute(cmd)) {
    ResultSet rs = stmt.getResultSet();
    //Exploitation du ResultSet
    ...
    rs.close();
} else {
    System.out.println("nombre de lignes modifiées " + stmt.getUpdateCount());
}
```



---

---

---

---

---

---

---

---

## JDBC

### Accès aux méta-données

- Permet de découvrir dynamiquement (au moment de l'exécution) des propriétés concernant la base de données ou les résultats de requêtes
- La méthode `getMetaData()` de la classe `Connection` permet d'obtenir les méta-données concernant la base de donnée.
  - Elle renvoie un `DatabaseMetaData`.
  - On peut connaître :
    - les éléments SQL supportés par la base
    - la structure des données de celle-ci
- La méthode `getMetaData()` de la classe `ResultSet` permet d'obtenir les méta-données d'un `ResultSet`.
  - Elle renvoie un `ResultSetMetaData`.
  - On peut connaître :
    - Le nombre de colonnes : `getColumnCount()`
    - Le nom d'une colonne : `columnName(int col)`
    - Le type d'une colonne : `getColumnType(int col)`
    - ...

---

---

---

---

---

---

---

---

## JDBC

### Gestion des transactions

- L'interface `Connection` offre des services de gestion des transactions
  - `setAutoCommit(boolean autoCommit)` définit le mode de la connexion (auto-commit par défaut)
  - `commit()` déclenche validation de la transaction
  - `rollback()` annule la transaction

```
try {
    con.setAutoCommit(false);
    // exécuter les instructions qui constituent la transaction
    stmt.executeUpdate("UPDATE INVENTORY SET ONHAND = 10 WHERE ID = 5");
    stmt.executeUpdate("INSERT INTO SHIPPING (QTY) VALUES (5)");
    ...
    // valide la transaction
    con.commit();
}
catch (SQLException e) {
    con.rollback(); // annule les opérations de la transaction
}
```

---

---

---

---

---

---

---

---

## JDBC

### A propos de SQLException

- `SQLException` définit les méthodes suivantes :
  - `getSQLState()` : --> un code d'état de la norme SQL ANSI-92
  - `getErrorCode()` : --> un code d'erreur spécifique (« vendor-specific »)
  - `getNextException()` : --> permet aux classes du JDBC de chaîner une suite de `SQLExceptions`

```
// du code très consciencieux
try {
    ...
}
catch (SQLException e) {
    while (e != null) {
        System.out.println("SQL 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();
    }
}
```

---

---

---

---

---

---

---

---

## JDBC

## SQLWarning

- Les classes du JDBC ont la possibilité de générer sans les lancer des exceptions quand un problème est intervenu mais qu'il n'est pas suffisamment grave pour interrompre le programme
  - Exemple : fixer une mode de transaction qui n'est pas supporté la base de données cible (un mode par défaut sera utilisé)
- **SQLWarning** encapsule même information que **SQLException**
- Pour les récupérer pas de bloc **try catch** mais à l'aide de méthode **getWarnings** des interfaces **Connection**, **Statement**, **ResultSet**, **PreparedStatement**, **CallableStatement**

```
void printWarnings(SQLWarning warn) {
    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.getNextException();
    }
}

...
ResultSet rs = stmt.executeQuery("SELECT * FROM CLIENTS");
printWarnings( stmt.getWarnings() );
printWarnings( rs.getWarnings() );
...
```

---

---

---

---

---

---

---

---

---

---

## JDBC

## Api JDBC 2.0

- JDBC 1.0
  - package supplémentaire (add-on) pour JDK 1.0
  - intégré l'API de base (core API) du JDK 1.1
- JDBC 2.0
  - spécification par SUN en mai 1998
    - extensions pour « meilleure » gestion des résultats (ResultSets « scrollables », « modifiables »)
    - support pour BLOBs (Binary Large Objects) et CLOBs (Character Large Objects)
    - ...
  - intégré à l'API de Java 2 (JDK 1.2)
  - compatibilité avec la version 1.0
    - code écrit pour JDBC 1.0 compile et fonctionne avec version 2.0 de l'API

---

---

---

---

---

---

---

---

---

---

## JDBC

## ResultSet JDBC 2.0

- Par défaut lorsque l'on crée un Statement les objets ResultSet sont en lecture seule (read only) et à accès séquentiel (forward only)

```
public Statement createStatement() throws SQLException
Statement stmt = conn.createStatement();
```

- Avec JDBC 2.0 possibilité de créer des ResultSet
  - « Scrollable »
    - plus de limitation à un parcours séquentiel
  - « Updateable »
    - possibilité de modifier les données dans la BD

```
public Statement createStatement(int resultSetType, int resultSetConcurrency)
```

```
ResultSet.TYPE_FORWARD_ONLY
ResultSet.TYPE_SCROLL_INSENSITIVE
ResultSet.TYPE_SCROLL_SENSITIVE
```

```
ResultSet.CONCUR_READ_ONLY
ResultSet.CONCUR_UPDATABLE
```

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

---

---

---

---

---

---

---

---

---

---



- Tous les **ResultSet** ne sont pas nécessairement modifiables
  - En général la requête ne doit référencer qu'une seule table sans jointure
- Tous les drivers JDBC ne supportent pas nécessairement et entièrement les **ResultSet** « scrollable » et « updateable »
  - l'objet **DataBaseMetaData** fournit de l'information quant au support proposé pour les **ResultSet**
  - Il faut être prudent si le logiciel que l'on écrit doit interagir avec une grande variété de drivers JDBC

---

---

---

---

---

---

---

---

---

---

---

---

- **javax.sql** package d'extension standard de JDBC
  - Pour les applications J2EE (Java 2 Enterprise Edition)
  - Inclus en standard dans J2SE (Java 2 Standard Edition) depuis version 1.4
- **DataSource** : Obtention du nom de la base à partir de serveurs de noms plutôt que d'avoir le nom de la base de données codé « en dur » dans l'application.
  - Utilisation de JNDI (Java Naming and Directory Interface) pour connexion à une base de donnée
- **PooledConnection** : support pour gestion d'un « pool » de connexion
  - gestion d'un cache des connexion ouvertes
  - évite la création de nouvelles connexions (ce qui est coûteux)
- **RowSet** : permet de traiter les résultats des requêtes comme des composants JavaBeans
- Support pour les transactions distribuées

---

---

---

---

---

---

---

---

---

---

---

---

Un exemple « complet »

```
import java.sql.*;
public class TestJDBC {
    public static void main(String[] args) throws Exception {
        Class.forName("postgres95.pgDriver");

        Connection conn = DriverManager.getConnection("jdbc:pg95:mabase",
                                                    "dedieu", "");
        Statement stmt = conn.createStatement();

        ResultSet rs = stmt.executeQuery("SELECT * from employee");
        while (rs.next()) {
            String nom = rs.getString("nom");
            String prenom = rs.getString("prenom");
            String email = rs.getString("email");
        }

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

---

---

---

---

---

---

---

---

---

---

---

---

# JDBC

## bibliographie

- *Java Entreprise in a Nutshell*, David Flanagan, Jim Farley, William Cawford et Kris Magnusson, Ed. O'Reilly, 1999
- *Database programming with JDBC and Java*, George Reese, Ed. O'Reilly, 1998
- Tutoriaux en ligne
  - *JDBCshort course*  
<http://developer.java.sun.com/developer/onlineTraining/Database/JDBCShortCourse/index.html>
- Applications : **JDBCTest** permet de tester via une interface graphique les différentes fonctionnalités proposées par le JDBC et de voir le code Java correspondant.  
(<http://developer.java.sun.com/developer/onlineTraining/Database/JDBCShortCourse/jdbc/exercises/JDBCTestConnect/index.html>)

Home: [dbcmanger.sourceforge.net](http://dbcmanger.sourceforge.net)

JDBC Explorer is an open source front-end for Data Base Management Systems that supports visualization and editing (if the corresponding JDBC driver supports ResultSets). Any JDBC database is supported, in the screenshots below you can see the app in action browsing MySQL, MS-SQL Server 2000, SAP DB - Ver:7.3.



Home: [qform.sourceforge.net](http://qform.sourceforge.net)

QueryForm is a GPL'd open source GUI database front-end that, in the words of developer Dave Glasser (email: [dglasser@pobox.com](mailto:dglasser@pobox.com)), "uses table metadata to build forms on-the-fly through which you can enter queries, browse results, and add, update or delete rows".

---

---

---

---

---

---

---

---

---

---

# JDBC

## Drivers JDBC

- 4 catégories de drivers JDBC
- type 1 : Pont JDBC-ODBC (Open Data Base Connectivity)

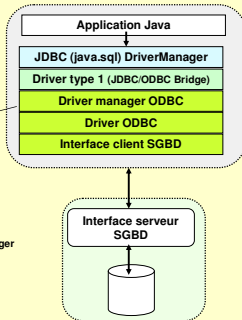
### ODBC

- interface d'accès (C) aux SGBD définie par Microsoft
- standard de fait, très grand nombre de SGBD accessibles

impose chargement dans la mémoire vive de la plateforme d'exécution de librairies dynamiques

### code binaire ODBC sur le client

- alourdit processus d'installation et de maintenance
- problème de sécurité pour les applets
  - applets « untrusted » n'ont pas l'autorisation de charger en mémoire du code natif



---

---

---

---

---

---

---

---

---

---

# JDBC

## Drivers JDBC

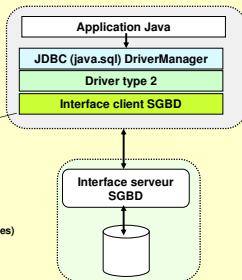
- Type 2 : API native

interface d'accès entre le driver manager JDBC et l'interface cliente du SGBD

impose chargement dans la mémoire vive de la plateforme d'exécution de librairies dynamiques (code binaire de l'interface client spécifique au SGBD par exemple librairies OCI, Oracle Call Interface, conçues initialement pour programmeurs C/C++)

### Driver dédié à un SGBD particulier

- moins ouvert que pont JDBC:ODBC
- potentiellement plus performant (moins de couches logicielles)
- mêmes problèmes qu'avec pont JDBC-ODBC
- code natif sur plateforme d'exécution



---

---

---

---

---

---

---

---

---

---

