JDBC

Java Databases Connectivity

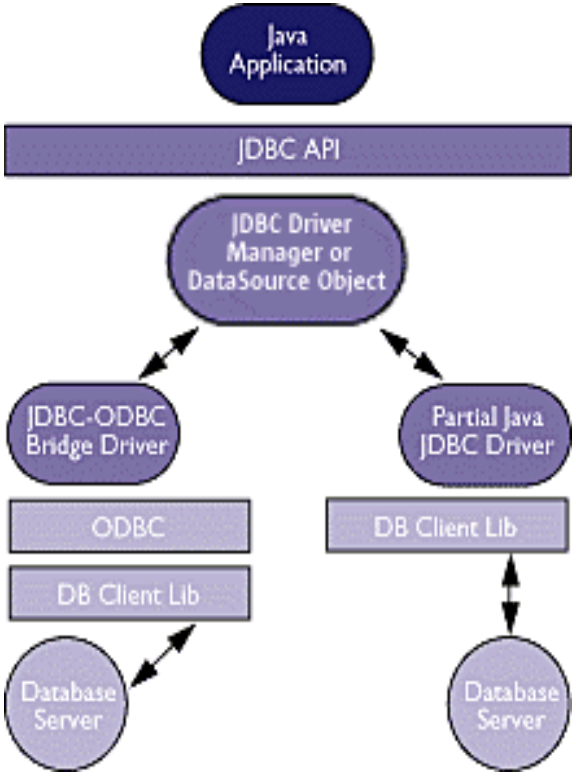
Objectifs

- Indépendance
  - Vision identique de SGBD différents
  - Utilisation de SQL
  - Définition des types adaptées à Java
- Simplicité
  - Connection, évaluation de requêtes, parcours des l'ensemble des résultats
- Adaptable
  - Possibilité d'utiliser les spécificités d'un SGBD (Attention)

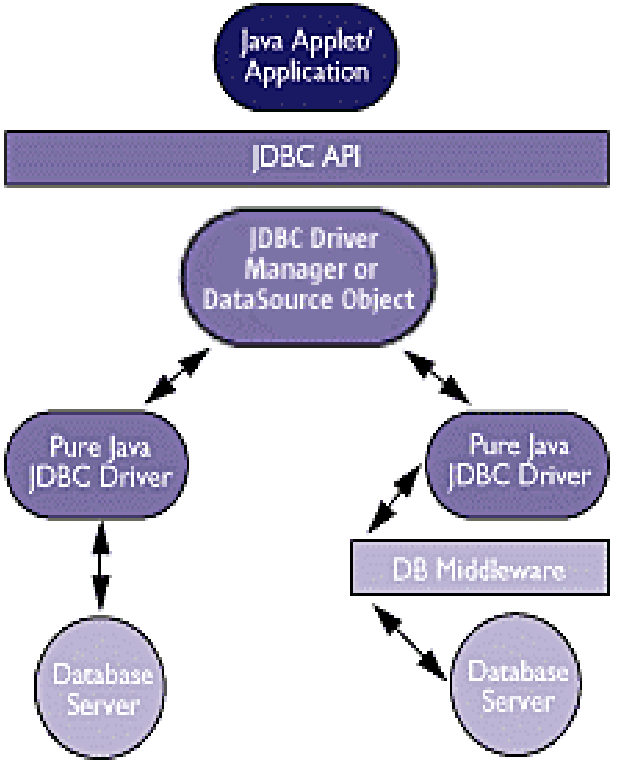
Architecture

- Java définit une interface générique (JDBC)
- Cette interface est implantée par des classes fournies par un *driver*
- On distingue 4 types de drivers :
  - *Type 1: JDBC-ODBC Bridge plus ODBC Driver*
  - *Type 2: A native API partly Java technology-enabled driver*
  - *Type 3: Pure Java Driver for Database Middleware*
  - *Type 4: Direct-to-Database Pure Java Driver*
    - Attention, les types 1. et 2. nécessite un adaptateur pour les applets
- Il existe des *drivers* pour bcp de SGBD (+ de 200)

Drivers JDBC (http://java.sun.com/products/jdbc)



Type 1 Type 2



Type 4 Type 3

Différentes versions de JDBC

- Dès l'origine : support de SQL-2
- JDBC 2
  - Différents type de resultSet choisi à la connection
    - Scroll-insensitive, Scroll-sensitive, Read-only
  - Déplacement du curseur
- JDBC 3
  - SQL3
  - Pool de connections, séquences, nouveaux types, SQL 3
- JDBC 4 (http://jcp.org/aboutJava/communityprocess/pr/jsr221/index.html)
  - Détection automatique du driver (en fct de l'URL)
  - XML
  - Blob, ...
  - Exceptions

Forme générale de code JDBC

(1) Charger la classe du *driver*
- Class.forname(), DriverManager.registrerDriver(), auto à partir de jdbc 3

(1) Ouvrir une connexion (ou un pool) sur une base de données
- Connection, Datasource

(2) Créer les requêtes ou les script SQL
- Statement, PreparedStatement, CallableStatement

(3) Exécuter
- interrogation (executeQuery()), mise à jour (executeUpdate()), cas complexe (execute())

(4) Parcourir les résultats (ResultSet)

(5) Fermer la connexion (close())

java.sql et javax.sql

- Ces paquetages contiennent un grand nombre d'interfaces (API JDBC) et quelques classes
- Les classes d'implantation des interfaces sont fournies par le *driver*

Interfaces Importantes

- **Driver** : Permet de créer une instance de Connexion
- **Connection** : Représente une connexion à une base et permet de créer des
  - requêtes SQL (**Statement**), éventuellement paramétrées (**PreparedStatement**) ou stockées sur le SGBD (**CallableStatement**)
- L'exécution d'une requête `select` retourne un **ResultSet** qui permet de parcourir le résultat
- Les informations sur le résultat sont représentées par un **ResultSetMetaData**
- **DatabaseMetaData** donne accès à des informations sur la base de données

Classes Utiles

- **DriverManager** : gestion des drivers et des connexions
- Type de données
  - **Date** : date SQL
  - **Time** : heures, minutes, secondes SQL
  - **TimeStamp** : comme **Time**, avec une précision à la microseconde
  - **Types** : constantes pour désigner les types SQL (cf. conversions vers Type Java et inverse)

1 – Le Drivers (Chargement - théorie)

- Trois méthodes de Chargement
  - Avec la classe DriverManager (simple mais « en dur »)
    - `DriverManager.registerDriver(new org.postgresql.Driver());`
  - Explicite en Java standard (chargement dynamique)
    - `Class.forName("com.mysql.jdbc.Driver").newInstance();`
    - `Class.forName("org.postgresql.Driver").newInstance();`
    - `L'appel au DriverManager est fait par le driver`
  - Automatique lors de la connection à partir JDBC 3

1 – Le Drivers (Chargement - Pratique)

- L'API JDBC est dans le JDK mais pas le Drivers
- Il faut donc :
  - Trouver l'archive qui contient le driver (ex. http://jdbc.postgresql.org/)
  - Permettre au compilateur et/ou à la JVM (et à eclipse !!) d'y accéder
    - Ajout dans le *classpath* (variable d'env. ou paramètre)
    - Utilisation d'un script ant qui charge automatique un répertoire entier
    - Configure le *buildpath* de eclipse
- Connaître le nom de la classe du driver ou laisser faire JDBC 3 (il faut alors connaître le format de l'URL de connection)

1 – Le Drivers (Connection 1/2)

- On utilise la méthode **connect()** de **Driver**
  - 1er paramètre une URL qui précise le driver et indique le SGBD et la base données
  - 2 autres paramètres qui indiquent le login et le mot de passe
- Retourne une instance de l'interface **Connection** qui permettra de lancer des requêtes vers le SGBD

1 – Le Drivers (Connection 2/2)

- On désigne la base de données avec une URL qui dépend du SGBD (cf. transparent suivant)
  - `String url="jdbc:odbc:ma_base";`
  - `String url="jdbc:postgresql://pc-bruno.univ-tln.fr/brunodatabase";`
- Puis on ouverture la connexion :

```
Connection connection =
  DriverManager.getConnection(url, user,
  password);
```

Format de l'URL JDBC

- Une URL pour une base de données est de la forme :
  - jdbc:*sous-protocole*:*base de donnée*
- Par exemple, pour Oracle :
  - jdbc:oracle:thin:@pc-bruno.univ-tln.fr:1521:brunodatabase
    - oracle:thin est le sous-protocole (driver « *thin* » ; Oracle fournit aussi un autre type de driver)
    - @sinfo1.univ-tln.fr:1521:brunodatabase désigne la base de données INFO située sur la machine sinfo1 (le serveur écoute sur le port 1521)
  - jdbc:oracle:oci8:pc-bruno@brunodatabase
- La forme exacte des parties *sous-protocole* et *base de données* dépend du SGBD cible

3- Création d'un *Statement*

- 3 types de *statement* :
  - `statement` : requêtes simples
  - `prepared statement` : requêtes précompilées
  - `callable statement` : procédures stockées
- Création d'un *statement* à partir la connection :

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

4 - Exécution de requêtes SQL simples (1/2)

- Instance de l'interface **Statement**
- La méthode à appeler est différente suivant la nature de la requêtes SQL que l'on veut exécuter :
  - Consultation (select)
    - executeQuery() On parcours les t-uples avec un ResultSet
  - Mise à jour (update, insert, delete) ou gestion de la base de données (create table,…)
    - executeUpdate() renvoie le nombre de lignes modifiées
  - Type inconnu (ex. donné par un fonction sous forme de String) ou si la requêtes peut renvoyer plusieurs résultats (procédures stockées)
    - execute()

4 - Exécution de requêtes SQL simples (2/2)

- Construction de la requête :
- ```
  String email = "dupont@monuniversite.demo";
  String query = "select nom, prenom from enseignant
    where email='"+
  +email+"'";
  ```
- Execution
  - `ResultSet rs = stmt.executeQuery(query);`
  ```

5 – Parcours des résultats Interface ResultSet

- **executeQuery()** retourne de ResultSet
- L'interface ResultSet définit les méthodes pour accéder au valeur des attributs
  - getXXX(int numéroDeColonne)
  - getXXX(String nomDeColonne)
  - XXX désigne le type Java de la valeur que l'on va récupérer (Byte, Boolean, AsciiStream, Short, String UnicodeStream, Int Bytes, BinaryStream, Long, Date, Object, Float, Time, BigDecimal,TimeStamp)
- A Noter : données volumineuses (ex. Blob)
  - Ouverture d'un flux

5 – Parcours des résultats - en pratique

```
java.sql.Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT x, y, z
  FROM uneTable");
while (rs.next())
{
        int i = rs.getInt("a");
        String s = rs.getString("b");
        byte b[] = rs.getBytes("c");
        ...
}
```

Exemple complet :

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

    Connection conn;
    conn = DriverManager.getConnection("jdbc:postgresql://pc-bruno/bruno",
    "test", "");

    Statement stmt = conn.createStatement();

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

Exceptions à traiter

- Erreur dans le code SQL : SQLException
- Avertissement lors de l'exécution (SQLWarning)
  - Problèmes de conversion de données (DataTruncation - sous-classe de SQLWarning)

Types JDBC/SQL

- Malgré SQL les SGBD présentent des différences de types
- JDBC masque ces différences en définissant ses propres types SQL (constantes de la classe **Types**)
- Le driver assure la conversion
  - SQL vers Java lors de la lecture
  - Java vers SQL lors du passage de paramètres

Liens en Types SQL et Java

- Utilisés explicitement avec les methodes getXXX() (et setXXX())
- Parfois plusieurs choix (presque tous les types SQL peuvent être retrouvés par getString())
  - CHAR et VARCHAR : getString()
  - LONGVARCHAR : getAsciiStream() et getCharacterStream()
  - BINARY et VARBINARY : getBytes()
  - LONGVARBINARY : getBinaryStream()
  - REAL : getFloat(), DOUBLE et FLOAT : getDouble()
  - DECIMAL et NUMERIC : getBigDecimal()
  - DATE : getDate(), TIME : getTime(), TIMESTAMP :getTimestamp()

Test un résultat vide : NULL

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery("SELECT * FROM
  ue");
while (rset.next()) {
  code = rset.getString(1);
  description = rset.getString(2);
  if (rset.wasNull()) description="Pas de
  description";
}
```

Les transactions

- Par défaut une connexion est ouverte en « *auto-commit* » :
  - un commit est automatiquement lancé après chaque requete SQL qui fait une mise à jour
- Pour un contrôle plus fin on utilise
  - **conn.setAutoCommit(false) pour le désactiver**
  - **conn.commit()** pour valider la transaction
  - **conn.rollback()** pour annuler la transaction

Précompilation des requêtes

- Si les requêtes fabriquées à partir de String changent (paramètres) :
  - Elles sont compilées à chaque appel d'où une perte de performances
- JDBC permet de ne compiler la requête qu'une fois (si le SGBD le supporte)
  - En indiquant les paramètres de façon générique
  - En fixant leur valeur (sans changer la requête) au moment de l'exécution
- Deux **Statement** particuliers :
  - Les requêtes paramétrées (**PreparedStatement**)
  - Les procédures stockées (**CallableStatement**)

Avantages des PreparedStatement

- Leur traitement est plus rapide si ils sont utilisés plusieurs fois avec des paramètres différents
- La portabilité est meilleures. L'utilisation des méthodes **setXXX()** (fournies par le driver) intégrent les spécifités du SGBD
  - Formats de date (JJ/MM/AA, ‘AAAA-MM-JJ, ...)
  - Formats de chaînes de caractères (échappement, ...)
  - ...

Création d'une requête paramétrée

- Interface PreparedStatement

```
PreparedStatement pstmt =
  conn.prepareStatement("UPDATE enseignant SET nom =
  ? WHERE email = ?");
```

- Les "?" indiquent les emplacements des paramètres

Définition des paramètres

- Valeurs des paramètres
  - Les valeurs des paramètres sont données par les méthodes setXXX(*n*, valeur)
- On choisit la méthode setXXX suivant le type SQL de la valeur que l'on veut mettre dans la base de données
- C'est au programmeur de passer une valeur Java du bon type à la méthode setXX

Requête paramétrée - Exemple

```
PreparedStatement pstmt =
   conn.prepareStatement("UPDATE enseignant SET nom =
   ? WHERE email = ?");

for (Enseignant e:listeEnseignant) {
   pstmt.setString(1, e.getNom());
   pstmt.setString(2, e.getEmail());
   pstmt.executeUpdate();
}
```

- Pour passer la valeur NULL à la base de donnée, on peut
  - utiliser la méthode setNull(n, type) (type de la classe **Types**)
  - ou passer la valeur Java **null** si la méthode setXXX() attend un objet en paramètre

Création d'une procédure stockée

- Les procédures stockées sont associées aux instances de l'interface *CallableStatement* qui hérite de l'interface *PreparedStatement*
- La création d'une instance de *CallableStatement* se fait par l'appel de la méthode *prepareCall* de la classe *Connection*
- On passe à cette méthode une chaîne de caractères qui décrit comment sera appelée la procédure stockée, et si la procédure renvoie une valeur ou non

Syntaxe pour les procédures stockées

- Syntaxe propre à JDBC :
  - si la procédure renvoie une valeur :
    - { ? = call nom-procédure(?, ?,...) }
  - si elle ne renvoie aucune valeur :
    - { call nom-procédure(?, ?,...) }
  - si on ne lui passe aucun paramètre :
    - { call nom-procédure }
- Exemple :
  - **CallableStatement cstmt = conn.prepareCall("{? = call augmenter(?,?)}");**

Lancement d'une procédure stockée

- Avant l'appel de la procédure les paramètres « in » et « in/out » sont fixés (setXXX() , cf . requêtes paramétrées)
- On précise les paramètres « out » et « in/out » par la méthode registerOutParameter()
- On exécute la procédure (executeQuery(), executeUpdate() ou execute())
- On récupère les paramètres « out » et « in/out » par les méthodes getXXX()

Exemple de procédure stockée

```
create or replace procedure augmentation (unDept in
  integer, pourcentage in number, cout out number) is
begin
  update emp
   set sal = sal * (1 + pourcentage / 100)
   where dept = unDept;
  select sum(sal) * pourcentage / 100
   into cout
   from emp
   where dept = unDept;
end;
```

Utilisation d'une procédure stockée

```
CallableStatement csmt = conn.prepareCall( "{ call
  augmentation(?, ?, ?) }");

// 2 chiffres après la virgule pour 3ème paramètre
csmt.registerOutParameter(3, Types.DECIMAL, 2);

// Augmentation de 2,5 % des salaires du dept 10
csmt.setInt(1, 10);
csmt.setDouble(2, 2.5);
csmt.executeQuery();
double cout = csmt.getDouble(3);
System.out.println("Cout total augmentation : "
+ cout);
```

Procédures stockées contenant plusieurs ordres SQL

- Une procédure stockée peut contenir plusieurs ordres SQL de divers types
- Pour retrouver tous les résultats de ces ordres, on utilise la méthode *getMoreResults()* de la classe **Statement**
- Ainsi, si elle contient 2 ordres SELECT, on commence par la lancer par la méthode *execute*, puis on récupère le 1er *ResultSet* par *getResultSet*, puis on récupère le 2ème résultat par *getMoreResults* et ensuite getResultSet

Interrogation des méta-données

Les Meta données

- JDBC permet de récupérer des informations sur le type de données que l'on vient de récupérer par un SELECT (interface ResultSetMetaData),
- mais aussi sur la base de données elle-même (interface DatabaseMetaData)
- Les données que l'on peut récupérer avec DatabaseMetaData dépendent du SGBD avec lequel on travaille

Accès aux méta-données

- La méthode `getMetaData()` permet d'obtenir les méta-données d'un `ResultSet`.
- Elle renvoit des `ResultSetMetaData`.
- On peut connaitre :
  - Le nombre de colonne : `getColumnCount()`
  - Le nom d'une colonne : `getColumnName(int col)`
  - Le type d'une colonne : `getColumnType(int col)`
  - ...

ResultSetMetaData

```
ResultSet rs = stmt.executeQuery("SELECT *
  FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
  String typeColonne =
  rsmd.getColumnTypeName(i);
  String nomColonne = rsmd.getColumnName(i);
  System.out.println("Colonne " + i + " de
  nom «  + nomColonne + " de type «  +
  typeColonne);
}
```

DatabaseMetaData

```
DatabaseMetaData dbmd = c.getMetaData();
System.err.println("Connected to " +
  dbmd.getDatabaseProductName()
+ " " + dbmd.getDatabaseProductVersion() + " as "
+ dbmd.getUserName());

java.util.List<String> listTables = new
  ArrayList<String>();
String[] types = { "TABLE", "VIEW" };
ResultSet rs = dbmd.getTables(null, null, "%", types);
while (rs.next()) {
listTables.add(rs.getString(3));
}
System.err.println("Tables et Vues :"+listTables);
```