Java et les bases de données relationnelles : JDBC

Bernard ESPINASSE
Professeur à Aix-Marseille Université (AMU)
Ecole Polytechnique Universitaire de Marseille

Septembre 2015

1. Introduction
2. L'API JDBC
3. Déroulement d'un accès à la base de données
4. Compléments JDBC
5. Un exemple complet de programme en JDBC

Plan

1. Introduction
- Client serveur « classique » versus client serveur « Java »
- Pourquoi JDBC
- Qu'est ce que JDBC

2. L'API JDBC
- Architecture générale
- Les 4 classes de drivers
- Les classes du package java.sql
- Les interfaces du package java.sql

3. Déroulement d'un accès à la BD
- Procédure générale
- Connexion à la BD
- Création et exécution de la requête, instruction simple « Statement »
- Traitement de la requête, et fermeture de la connexion

4. Complements JDBC
- **Traduction des types de données SQL/Java**
- Accès aux méta-données
- Instructions élaborées « PreparedStatement » et « CallableStatement »

5. Un exemple complet de programme en JDBC

Principales sources du cours

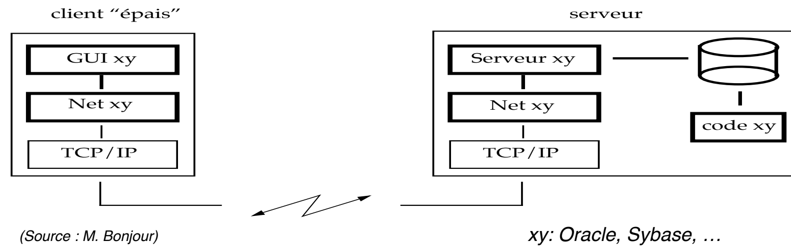
Webographie :
- PostgreSQL : https://jdbc.postgresql.org/ et http://docs.postgresql.fr/7.4/jdbc-use.html
- Oracle : http://www.oracle.com/technetwork/java/overview-141217.html
- Wikipedia : https://fr.wikipedia.org/wiki/Java_Database_Connectivity

Présentations :
- S. Rosmorduc, « Introduction à JDBC », Université Paris 8
- Cours de E. Cariou, « Introduction à JDBC : Accès aux bases de données en Java », Université de Pau et des Pays de l'Adour
- Cours de M. Bonjour, « Java et les bases de données », Université de Genève
- Cours de D. Fournier, « JDBC », Université de Le Havre
- …

1. Introduction à JDBC

- **Client serveur « classique » versus client serveur « Java »**
- **Pourquoi JDBC**
- **Qu'est ce que JDBC**

Client-serveur architecture « classique »



(Source : M. Bonjour)

xy: Oracle, Sybase, …

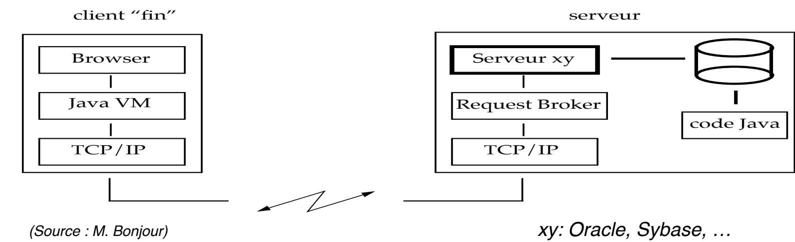
Côté client (client « épais ») :

- GUI : "moche", non-standard, runtime
- Net : protocole fermé, mais: sécurité, performances
- "Epaisseur": installation, configuration "câblée"
- Développement: langage "client", peu portable

Côté serveur :

- Net: protocole fermé, mais: répartition, homogénéité
- Développement: langage "serveur", pas portable

Client-serveur architecture « Java»



(Source : M. Bonjour)

xy: Oracle, Sybase, …

Côté client (client « fin ») :

- Browser: interface uniforme, bookmarks, plug-ins
- Java VM: sécurité
- "Finesse": TOUT se télécharge, pas de config locale
- Développement: Java, HTML (voire JavaScript)

Côté serveur :

- Request Broker: centralise les appels aux services
- Développement: Java
  - local: code exécuté par le serveur
  - mobile: code stocké dans la BD et envoyé au client

Pourquoi JDBC

Constat :

- **Java** est de **plus en plus utilisé** pour développer des **applications BD en client-serveur** (Web, Browser, …)
- Besoin d'une **API Java** pour :
  - interagir avec des BD relationnelles :
    - exécuter des requêtes SQL
    - récupèrer les résultats
  - permettant de standardiser l'accès aux BD :
    - Spécification basée sur X/Open SQL CLI (comme ODBC)

Réponse de SUN à la pression des développeurs :

- **JDBC (Java Java Data Base Connectivity)**

Qu'est-ce que JDBC

- Une **API (Application Programming Interface)** Java permettant à un programme d'application en Java d'exécuter des instructions SQL pour accéder à une base de données relationnelle

- Une API permettant un accès uniforme à des BD relationnelles :
  - **portable sur la plupart des OS**
  - **indépendant du SGBD** (seule la phase de connexion est spécifique – driver)
  - **compatible avec la plupart des SGBDR** : Oracle, Postgres, MySQL, Informix, Sybase, MS SQL Server...

2. L'API JDBC

- **Architecture**
- **Les 4 classes de drivers**
- **Les classes et interfaces du package java.sql**

L'API JDBC

API JDBC :

- Actuellement **JDBC 4.0**
- API Java dont les ***classes*** et ***interfaces*** sont dans le **package java.sql** (import java.sql.*) et complété par le package ***javax.sql***
- Fait partie du **JDK** (Java Development Kit)
- Repose sur 8 classes et 8 interfaces définissant les objets nécessaires :
  - à la connexion à une BD distante
  - à la création et l'exécution de requêtes SQL
  - à la récupération et le traitement des résultats

Architecture de JDBC (1)

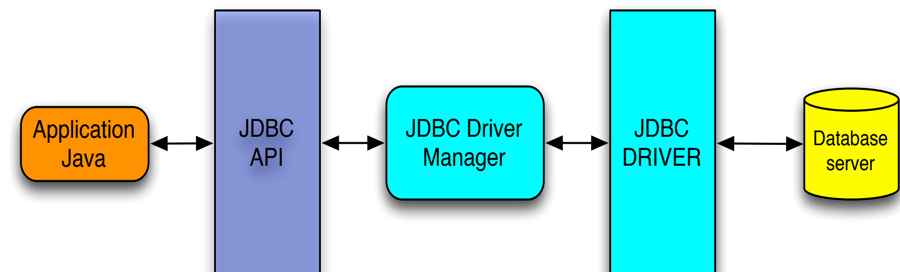
Architecture à 2 niveaux :

- **Niveau supérieur : API JDBC**
  - couche visible, nécessaire pour interfacer applications Java et SGBD
  - package java.sql
- **Niveau inférieur : DRIVERS**
  - implantation des drivers
  - interface entre les accès bas niveau au moteur du SGBD et l'application

Drivers (ou pilotes) :

- chaque SGBD utilise un **pilote (driver)** particulier
- permettent de traduire les requêtes JDBC dans le langage du SGBD
- constitués de ***classes*** implantant certaines ***interfaces*** de java.sql
- plus de **200 drivers** sont actuellement disponibles

Architecture de JDBC (2)



- **Application Java :** développée par le programmeur
- **JDBC API :** donné par SUN
- **JDBC Driver Manager:** donné par SUN
- **JDBC DRIVER :** donné ou vendu par un fournisseurs d'accès au serveur BD

Les 4 classes de drivers de JDBC (1)

4 classes de drivers (taxonomie de JavaSoft) :

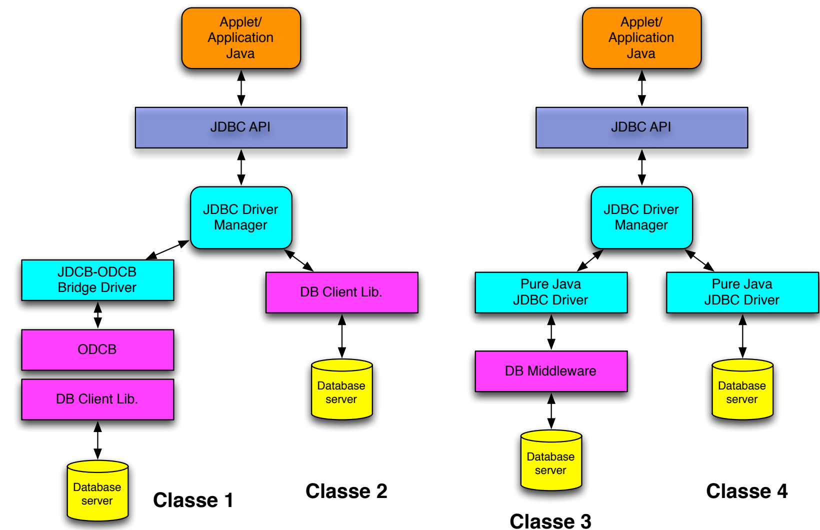
Drivers « natifs » :

- utilisent une partie écrite dans du ***code spécifique non Java*** (souvent en langage C) et appelé par ces implantations
- ***rapides*** mais doivent être ***présents sur le poste client*** car ne peuvent pas être téléchargés par le ClassLoader de Java (pas des classes Java mais des librairies dynamiques)
- ne peuvent donc ***pas être utilisés*** par des ***applets*** dans des ***browsers*** classiques :
  - **Classe 1 : JDBC-ODBC bridge driver** (pont JDBC-ODBC)
  - **Classe 2 : Native-API, partly-Java driver** (driver faisant appel à des fonctions natives non Java de l'API du SGBD)

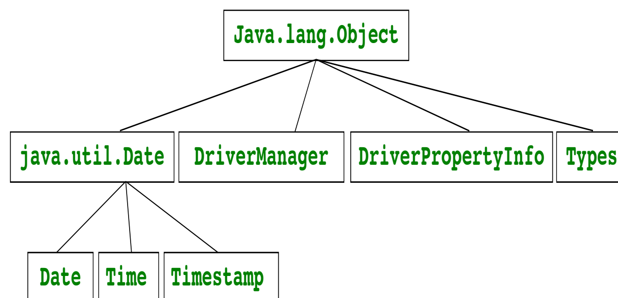
Drivers « Java » :

- **Classe 3 : Net-protocol, all-Java driver** (driver qui permet l'utilisation d'un middleware qui fait l'interface avec la BD
- **Classe 4 : Native-protocol, all-Java driver** (driver écrit entièrement en Java qui utilise le protocole réseau du SGBD pour accéder à la BD)

Les 4 classes de drivers de JDBC (2)

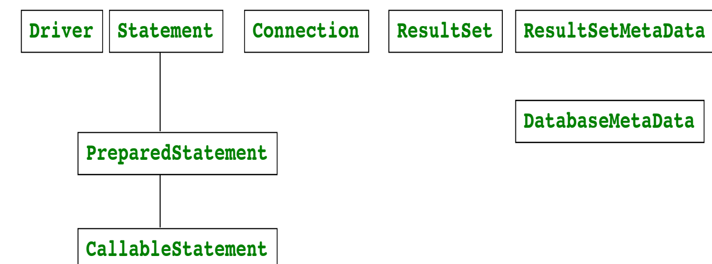


Les classes du Package « java.sql »



- **DriverManager** : gère les drivers, lance les connexions aux BD
- **Date** : date SQL
- **Time** : heures, minutes, secondes SQL
- **TimeStamp** : comme Time, avec une précision à la microseconde
- **Types** : constantes pour désigner les types SQL (conversions)

Les interfaces du Package « java.sql »



- **Driver** : renvoie une instance de Connection
- **Connection** : connexion à une BD
- **Statement** : instruction SQL
- **PreparedStatement** : instruction SQL paramétrée
- **CallableStatement** : procédure stockée dans la BD
- **ResultSet** : n-uplets récupérés par une instruction SQL
- **ResultSetMetaData** : description des n-uplets récupérés
- **DatabaseMetaData** : informations sur la BD

3. Déroulement d'un accès à la BD

- **Procédure générale**
- **Connexion à la BD**
- **Création de la requête**
- **Exécution de la requête**
- **Traitement de la requête**
- **Fermeture de la connexion**

Principe général d'un accès à un SGBD

Première étape :

- Préciser le type de driver que l'on veut utiliser
- Le driver permet de gérer l'accès à un type particulier de SGBD

Deuxième étape :

- Récupérer un objet « Connection » en s'identifiant auprès du SGBD
- Préciser la BD utilisée

Etapes suivantes :

- A partir de la connexion, créer un « statement » (état) correspondant à une requête particulière
- Exécuter ce statement au niveau du SGBD
- Fermer le statement

Dernière étape

- Se déconnecter de la base en fermant la connexion

Procédure d'un accès à un SGBD

Procédure :

- **0 - Importation du package java.sql**
- **1 - Enregistrement du BON driver JDBC** : recherche et chargement du driver approprié à la BD
- **2 - Connexion au SGBD** : établissement de la connexion à la BD
- **3 - Création d'une requête** (ou instruction SQL)
- **4 - Exécution de la requête** : envoi de cette requête et récupération des réponses
- **5 - Traitement des résultats**
- **6 - Fermeture de la connexion**

Programmation avec JDBC :

- On utilise le paquetage java.sql
- La plupart des méthodes lèvent l'exception java.sql.SQLException.

1 - Enregistrement du driver

- La classe DriverManager gère les différentes instances de Drivers
- La classe du driver doit être chargée en mémoire :
  `Class.forName("nomDriver");`
- la classe crée une instance d'elle même et enregistre cette instance auprès de la classe DriverManager
- Ex :

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Class.forName("com.mysql.jdbc.Driver");
Class.forName("org.postgresql.Driver");
```

2 - Connexion à la base

- Utilisation de la méthode Connection getConnection(String url, String user, String password) renvoie SQLException de DriverManager
- Le DriverManager essaie tous les drivers enregistrés (chargés en mémoire avec Class.forName()) jusqu'à ce qu'il trouve un driver qui lui fournisse une connexion
- Ex : `Connection connexion = DriverManager.getConnection(url, toto, password);`

3 - Création d'une requête

- La requête est créée avec l'interface **Statement** qui possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion
- Il existe 3 types de Statement :
  - **Statement** : **instruction simple** (requêtes statiques simples)

    Ex : Statement req1 = connexion.createStatement();
  - **PreparedStatement** : **requêtes paramétrables (ou dynamiques) précompilées** (requêtes avec paramètres d'entrée/sortie)

    Ex : PreparedStatement req2 = connexion.prepareStatement(str);
  - **CallableStatement** : **procédures stockées**

    Ex : CallableStatement req3 = connexion.prepareCall(str);

3 – Requête simple : Statement

- **Instruction simple : Classe** Statement

  Statement String = connexion.createStatement();

  ResultSet executeQuery(String ordre)
  - Exécute un ordre de type SELECT sur la BD
  - Retourne un objet de type ResultSet contenant tous les résultats de la requête

  int executeUpdate(String ordre)
  - Exécute un ordre de type INSERT, UPDATE, ou DELETE

  void close ()
  - Ferme l'état

4 - Exécution d'une requête (1)

Il y 3 types d'exécutions différents :

- en **consultation** (requêtes de type SELECT) :
  - avec la méthode executeQuery()
  - qui retourne un ResultSet (n-uplets résultants)
- en **modification** (requêtes de type INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE) :
  - avec la méthode executeUpdate()
  - qui retourne un entier (nombre de n-uplets traités)
- les exécutions de **nature indéterminée** avec la méthode execute()

Remarques : les méthodes executeQuery() et executeUpdate() de l'interface Statement prennent comme argument la requête SQL à exécuter

Ex :
```
Statement stmt = connexion.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM auteur WHERE aut_id > 60");
```


5 - Traitement des résultats (1)

- Seules les données demandées sont transférées en mémoire par le driver JDBC
- L'interface **ResultSet** permet d'accéder ligne par ligne au résultat retourné par une requête SELECT grâce à ses méthodes :
  - **méthode booléenne next() : fait avancer le curseur sur la ligne suivante**
    - Retourne true si le déplacement a été fait, false s'il n'y avait pas d'autre ligne
    - Traitement de chaque ligne : while(rs.next()) {// Traitement de chaque ligne}
  - **méthode booléenne previous() : recule d'une ligne**
    - Retourne true si le déplacement a été fait, false s'il n'y avait pas de ligne précédente
  - **méthode absolute(int ligne)** : se place à la **ligne n° ligne**
  - **autres méthodes : first(), last()**

5 - Traitement des résultats (2)

- Les **colonnes** sont identifiées par leur numéro (à partir de 1) ou par leur nom
- **Accès aux colonnes/données dans une ligne** par la méthode **[type] get[Type](int col)** :
  - Retourne le contenu de la colonne col dont l'élément est de type [type] qui est un type Java **compatible** avec le type de la colonne, pouvant être String, int, float, boolean ...

    Ex : `String getString(int col)`
- **Ex :**

```
int id = rs.getInt("aut_id");
String nom = rs.getString("aut_nom");
String prenom = rs.getString(3);
```


5 - Traitement des résultats (3)

L'interface **ResultSet** possède des méthodes pour **gérer les NULL:**

- La méthode **wasNull()** renvoie true si l'on vient de lire NULL
- Toutes les méthodes précédentes **getType() convertissent** une valeur NULL SQL en une valeur compatible Java pour le type d'objet demandé :
  - null Java pour les méthodes retournant un objet,
  - 0 pour les méthodes numériques,
  - false pour getBoolean()

5 - Traitement des résultats (4)

Si une **requête ne peut s'exécuter** ou qu'une **erreur de syntaxe SQL** a été détectée, **l'exception SQLException** est générée :

- Toutes les **méthodes** présentées précédemment sont **concernées**
- **Opérations possibles** sur cette exception :
  - int getErrorCode() : retourne le code de l'erreur renvoyé par le SGBD (et dépendant du type du SGBD)
  - SQLException getNextException() : si plusieurs exceptions sont chaînées entre elles, retourne la suivante ou null s'il n'y en a pas
  - String getSQLState() : retourne « l'état SQL » associé à l'exception
- **Nombreuses exceptions (voir API):**
  - SQLException : erreurs SQL
  - SQLWarning : avertissements SQL
  - DataTruncation : lorsqu'une valeur est tronquée lors d'une conversion SGBD Java
  - …

6 - Fermeture de la connexion

- Fermer les connexions ouvertes permet de libérer les ressources mobilisées
- Chaque objet possède une **méthode close() :**

resultset.close();

statement.close();

connection.close();

Exemple (1/2)

```
String url = "jdbc:postgresql://localhost:5433/dominique"; //5432 port par défaut
Connection con;
Statement stmt;
String query = "SELECT * FROM auteur WHERE aut_id > 60 ORDER BY aut_nom,
aut_prenom";
try {
        Class.forName("org.postgresql.Driver");
} catch(java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException (try): ");
        System.err.println(e.getMessage());
}
try {
    Properties props = new Properties();
    props.setProperty("user","dom");
    props.setProperty("password","toto");
    con = DriverManager.getConnection(url,props);
    stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    System.out.println("Quelques auteurs :");
```


Exemple (2/2)

```
    while (rs.next()) {
        int id = rs.getInt("aut_id");
        String nom = rs.getString("aut_nom");
        String prenom = rs.getString("aut_prenom");
        System.out.println(id + " " + nom+ " " + prenom );
    }
    rs.close();
    stmt.close();
    con.close();
} catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
}
```


4. Compléments JDBC

- Traduction des types de données SQL/Java
- Accès aux méta-données
- Requêtes paramétrables « PreparedStatement »
- Requêtes procédures stockées « CallableStatement »

Traduction des types de données SQL/Java (1)

Correspondances entre les types SQL et les types Java :

Type SQL	Types Java
CHAR, VARCHAR, LONGCHAR	string
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	bite[ ]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Traduction des types de données SQL/Java (2)

Traduction des types par le driver JDBC :

Tous les SGBD n'ont pas les mêmes types SQL (même pour types de base, il peut y avoir des différences importantes) : le driver JDBC spécifique au SGBD **traduit** le type JDBC retourné par le SGBD en un type Java par des méthodes :

Type SQL	Méthode Java	Type SQL	Méthode Java
CHAR, VARCHAR	getString()	DOUBLE, FLOAT	getDouble()
LONGVARCHAR	getAsciiStream()	DATE	getDate()
NUMERIC, DECIMAL	getBigDecimal()	TIME	getTime()
BINARY, VARBINARY	getBytes()	TIME STAMP	getTimeStamp()
LONGVARBINARY	getBinaryStream()	ARRAY	getArray()
BIT	getBoolean()	BLOB	getBlob()
INTEGER	getInt()	CLOB	getClob ()
BIGINT	getLong()	REF	getRef ()
SMALLINT	getShort()	REAL	getFloat()
TINYINT	getByte()	AUTRE	getObject()

Types de méta-données

Méta-données : informations décrivant :
- la BD (interface DatabaseMetaData) ou
- les types de données des résultats retournés par une requête (interface ResultSetMetaData).

- **DatabaseMetaData :** méthode **getMetaData()** de la classe **Connection** :
  - Permet d'accéder aux informations :
    - les tables de la BD : getTables()
    - le nom de l'utilisateur : getUserName()
    - les procédures stockées : getProcedures()
- **ResultSetMetaData :** méthode **getMetaData()** de la classe **ResultSet** :
  - Permet d'accéder aux informations :
    - nombre de colonne : getColumnCount()
    - nom d'une colonne : getColumnName(int numcol)
    - type d'une colonne : getColumnTypeName(int numcol)
    - si un NULL SQL peut être stocké dans une colonne : isNullable()

Types de méta-données

Méthodes de l'interface ResultSetMetaData :

getCatalogName()
getColumnClassName()
getColumnCount()
getColumnDisplaySize()
getColumnLabel()
getColumnName()
getColumnType()
getColumnTypeName()
getPrecision()
getScale()
getSchemaName()
getTableName()
isAutoIncrement()
isCaseSensitive()
isCurrency()
isDefinitelyWritable()
isNullable()
isReadOnly()
isSearchable()
isSigned()
isWritable()

Accés aux meta-données : exemple

```
ResultSet rs = st.executeQuery(
              "SELECT * FROM livre");
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);
}
```

Requêtes paramétrables : PreparedStatement (1)

- Via un objet **PreparedStatement**, on envoie une **requête paramétrable** au SGBD pour **précompilation**
- Méthode plus **rapide** que le Statement standard car le SGBD **n'analyse qu'une seule fois la requête**
- Utile lorsqu'on exécute souvent la même requête (à quelques variables près)
- **Remarque** :
  - pas disponible sur tous les SGBD.

Requêtes paramétrables : PreparedStatement (2)

- **Avant d'exécuter l'ordre, on remplit les champs avec :**

  void set[Type](int index, [Type] val)
  - Remplit le champ en ième position définie par index avec la valeur val de type [Type]
  - [Type] peut être : String, int, float, long ...
  - Ex : void setString(int index, String val)

  ResultSet executeQuery()
  - Exécute un ordre de type SELECT sur la base
  - Retourne un objet de type ResultSet contenant tous les résultats de la requête

  int executeUpdate()
  - Exécute un ordre de type INSERT, UPDATE, ou DELETE

Requêtes paramétrables : PreparedStatement (3)

- **Création d'une instance de PreparedStatement par la Méthode prepareStatement() de Connection :**
  - Paramètres spécifiés par un ?
  - Instanciés ensuite par une méthode setInt(), setDate(), setString(), ...
  - Toutes les méthodes set*Type* ont 2 paramètres :
    - le **numéro de l'argument** dans la requête et
    - sa **valeur**
  - Ex 1 : Consultation

```
PreparedStatement ps = c.prepareStatement("SELECT * FROM auteur WHERE
aut_nom = ? ");
```

  - Ex 2 : Mise à jour

```
PreparedStatement ps = c.prepareStatement("UPDATE Stade SET sta_capacite = ?
WHERE sta_nom = ?");
...
for(int i = 0; i < 31; i++) {     ps.setInt(1, capacite[i]);
     ps.setString(2, nom[i]);
     int count = ps.executeUpdate();
}
```


Requête exécutant une procédure stockée : CallableStatement (1)

- **Via un objet CallableStatement, on peut exécuter une procédure stockée et manipuler son résultat en utilisant la méthode prepareCall() de Connection :**

```
CallableStatement ps1 = c.prepareCall("{? = call nomProcedure(?,?)}");
CallableStatement ps2 = c.prepareCall("{call nomProcedure(?,?)}");
CallableStatement ps3 = c.prepareCall("{call nomProcedure}");
```

  Passage de paramètres identique à PreparedStatement (héritage)
- **Ex :**

```
CallableStatement testCall;
testCall = conn.prepareCall(
   "{ call supprimeAuteur(?,?) }");
testCall.setString(1, "Asimov");
testCall.setString(2, "Isaac");
testCall.execute();
```

Création du Statement CallableStatement (2)

- **Récupération du Résultat :**
  - Identique pour le résultat renvoyé par une fonction ou des paramètres OUT et IN/OUT pl/sql
  - methode **get*Type*()**

- **Ex :**

```
CallableStatement testCall;
testCall = conn.prepareCall(
     "{ ? = call qteAuteur(?,?) }");
testCall.setString(2, "Asimov");
testCall.setString(3, "Isaac");
testCall.execute();
int nbLivres = testCall.getInt(1);
String nom =  testCall.getString(2);
String prenom =  testCall.getString(3);
```


Gestion des transactions

- **Transaction :**
  - Un ensemble d'action est effectué en entier ou pas du tout
- **Gestion des transactiond en JDBC :**
  - Par défaut : **autocommit on**
  - Pour modifier ce mode on appel la méthode **setAutoCommit()** de la classe Connection (paramètres true et false)
  - Pour valider ou annuler on appel les méthodes **commit()** et **rollback()** de la classe Connection
- **Pour plus de détails, voir documentation spécialisée**

5. Un exemple complet d'un programme JDBC

•

Exemple complet (1/6)

(Source : A. Lacayrelle cité par E. Cariou)

- **Accès à une base Oracle contenant 2 tables :**
  - produit (codprod, nomprod, *codecat**) *table produits*
  - categorie (codecat, libellecat) *table catégorie de produits*
- **Paramètres de la base :**
  - Fonctionne sur la machine ladybird sur le port 1521
  - La BD s'appelle « test »
  - L'utilisateur qui se connecte : « étudiant »,
  - Son mot de passe : « mdpetud »

Exemple complet (2/6)

- **Création de la connexion à la base :**

```
Connection con;

// chargement du driver Oracle
DriverManager.registerDriver(
  new oracle.jdbc.driver.OracleDriver());

// création de la connexion
con = DriverManager.getConnection(
  "jdbc :oracle :thin :@ladybird :1521
  :test, "etudiant", "mdpetud");
```

//note: la syntaxe du premier argument dépend du type du SGBD

Exemple complet (3/6)

- **Exécution d'une instruction simple de type SELECT :** Lister toutes les caractéristiques de toutes les catégories

```
Statement req;
ResultSet res;
String libelle;
int code;

req = con.createStatement();
res = req.executeQuery(
   "select codcat, libellecat from categorie");

while(res.next()) {
    code = getInt(1);
    libelle = getString(2);
    System.out.println(
         " produit : "+code +","+ libelle);
}
req.close();
```


Exemple complet (4/6)

- **Exécution d'une instruction simple de type UPDATE :** Ajouter une catégorie « céréales » de code 5 dans la table catégories

```
Statement req;
int nb;

req = con.createStatement();

nb = req.executeUpdate("
  insert into categories values (5, 'cereales')");

System.out.println(
     " nombre de lignes modifiées : "+nb);

req.close();
```


Exemple complet (5/6)

- **Instruction paramétrée de type SELECT :** Retourne tous les produits de la catégorie 'céréales'

```
PreparedStatement req;
ResultSet res;
String nom;
int code;

req = con.prepareStatement("select codprod, nomprod
    from categorie c, produit p where c.codcat=p.codcat
    and libellecat = ?");

req.setString(1, "cereales");

res = req.executeQuery();

while(res.next()) {
     code = getInt(1);
     libelle = getString(2);
     System.out.println( " produit : "+code +","+ libelle); }
req.close();
```

Exemple complet (6/6)

- **Instruction paramétrée de type UPDATE :** Ajout de 2 nouvelles catégories dans la table catégorie

```
PreparedStatement req;
int nb;

req = con.prepareStatement("insert into categories values (?,?)");

req.setInt(1, 12);
  req.setString(2, "fruits");
  nb = req.executeUpdate();

req.setInt(1, 13);
  req.setString(2, "légumes");
  nb = req.executeUpdate();
  req.close();
```