



Cours Programmation Orientée Objet Avancée

A.U: 2020/2021

2LBC-BI



Chapitre 3 : Accès aux BD via Java DataBase Connectivity (JDBC)

Plan

- ▶ Introduction
- ▶ Principes et architecture de JDBC
- ▶ Processus d'accès à la base de données
- ▶ Compléments API JDBC

Introduction



- ▶ Java est de plus en plus utilisé pour développer des applications qui interagissent avec BD
 - ▶ Java: Monde des OBJETS
 - ▶ SGBDR : Monde des RELATIONS
 - ▶ Besoin d'un API Java pour interagir avec des BD relationnelles: Exécuter des requêtes SQL et récupérer des résultats.
- Un mapping O-R est nécessaire (ORM)



Java DataBase Connectivity (JDBC)



- ▶ JDBC est une API (Application Programming Interface) qui permet un accès homogène à des bases de données depuis un programme Java au travers du langage SQL.
- ▶ L'API JDBC:
 - Portable sur la plupart des OS
 - Indépendant du SGBD
 - Compatible avec la plupart des SGBDR : Oracle, Postgres, MySQL, Informix, Sybase, MS SQL Server, etc.

Java DataBase Connectivity (JDBC)

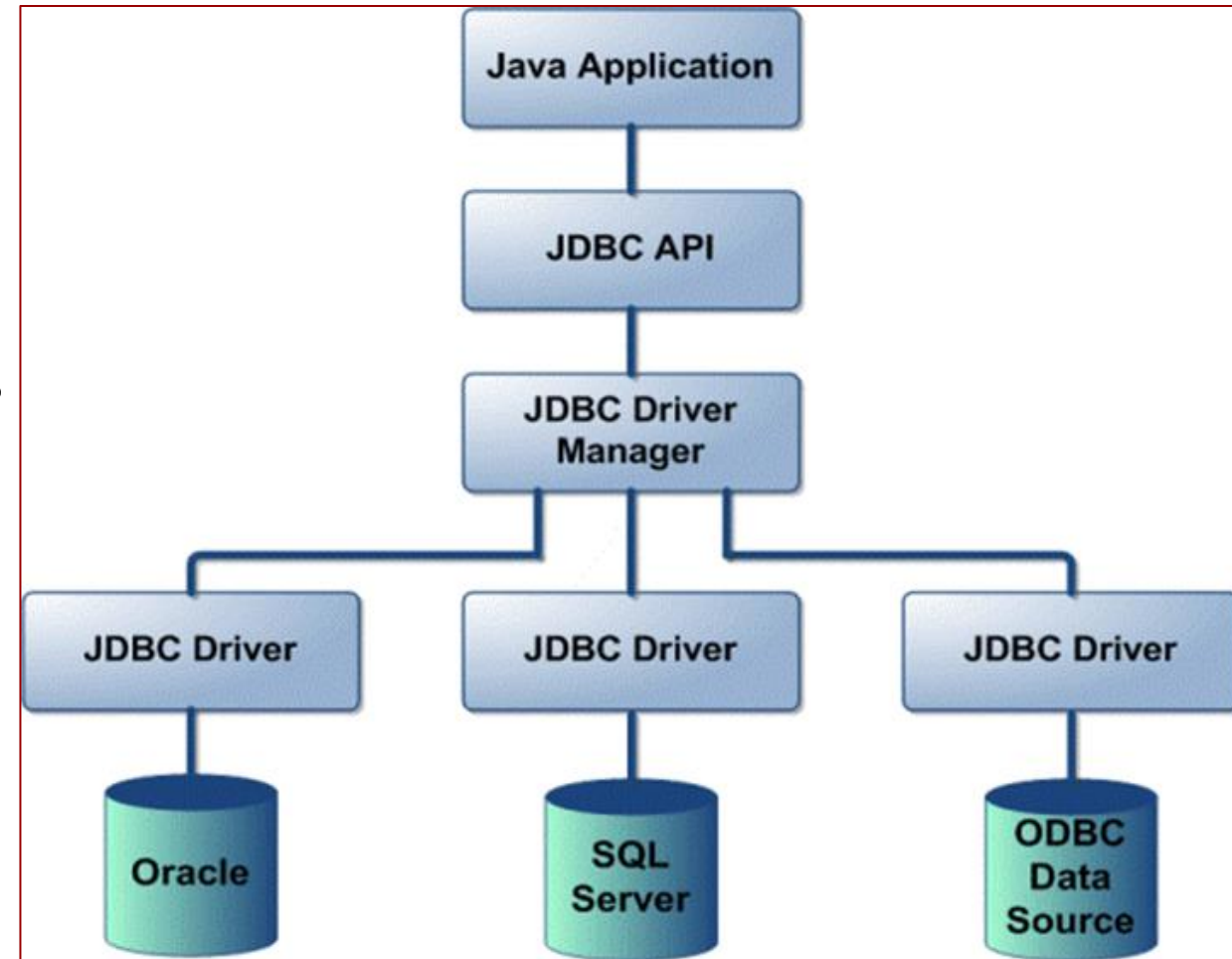
- ▶ JDBC fait partie du JDK (Java Development kit): définit un ensemble de classes et d'interfaces pour exécuter des instructions SQL
- ▶ Toutes les classes et interfaces sont dans le package java.sql

`import java.sql.*`

- ▶ JDBC définit un protocole de communication entre le programme Java client et le serveur de bases de données pour:
 - Ouverture/Fermeture de connexion à une base de données
 - Exécution de requêtes SQL
 - Exploitation des résultats: correspondance types SQL- types JAVA
 - Accès au méta-modèle: description des objets du SGBD

Java DataBase Connectivity (JDBC) Architecture

- ▶ Application Java: développée par le programmeur
- ▶ JDBC API: donné par SUN
- ▶ JDBC DriverManager: donné par SUN, classe Java à laquelle s'adresse le code de l'application Client, permet de charger et configurer les pilotes JDBC nécessaires à l'application
- ▶ JDBC Driver: donné ou vendu par un fournisseur d'accès au serveur BD.



Java DataBase Connectivity (JDBC) Architecture

► Une architecture à deux niveaux:

■ Niveau supérieur: API JDBC

- c'est la couche visible et utile pour développer des applications Java accédant à des SGBD
- représentée par le package java.sql

■ Niveau inférieur

- destiné à faciliter l'implémentation de drivers pour des bases de données
- représente une interface entre les accès de bas niveau au moteur du SGBD et l'application

Java DataBase Connectivity (JDBC)

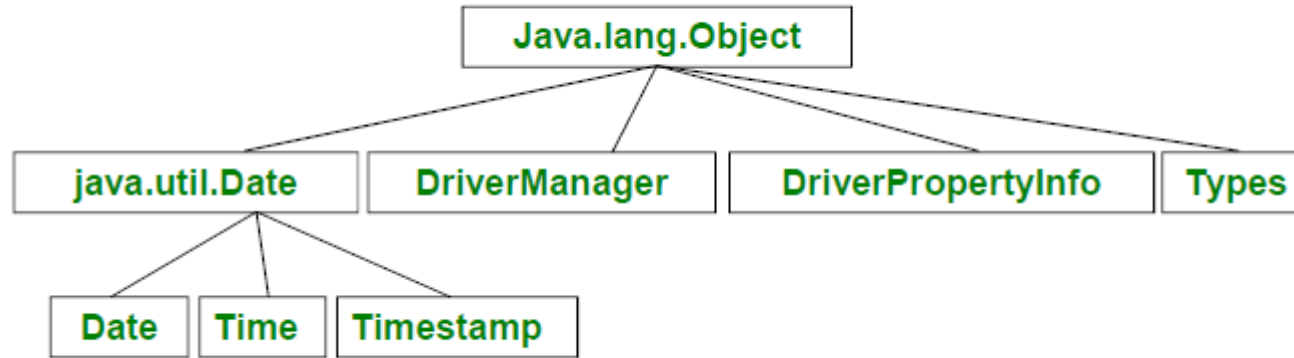
Architecture

► Drivers (pilote)

- Chaque SGBD utilise un driver (pilote) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBD.
 - Le driver est un ensemble de classes qui implantent les interfaces de JDBC
 - Le driver va établir le lien entre le programme Java et le SGBD
 - Des pilotes existent pour MySQL, postgresQL, ACCESS, Oracle, SQLServer
- Une même application peut utiliser plusieurs drivers pour accéder à plusieurs BD.

Java DataBase Connectivity (JDBC)

API JDBC

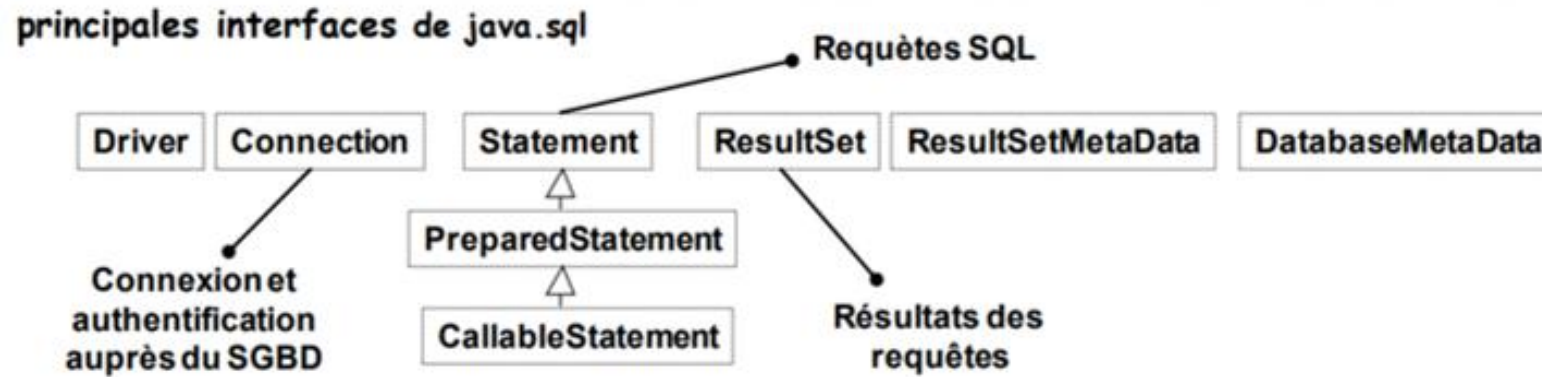


Les classes du package `java.sql`

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

Java DataBase Connectivity (JDBC)

API JDBC

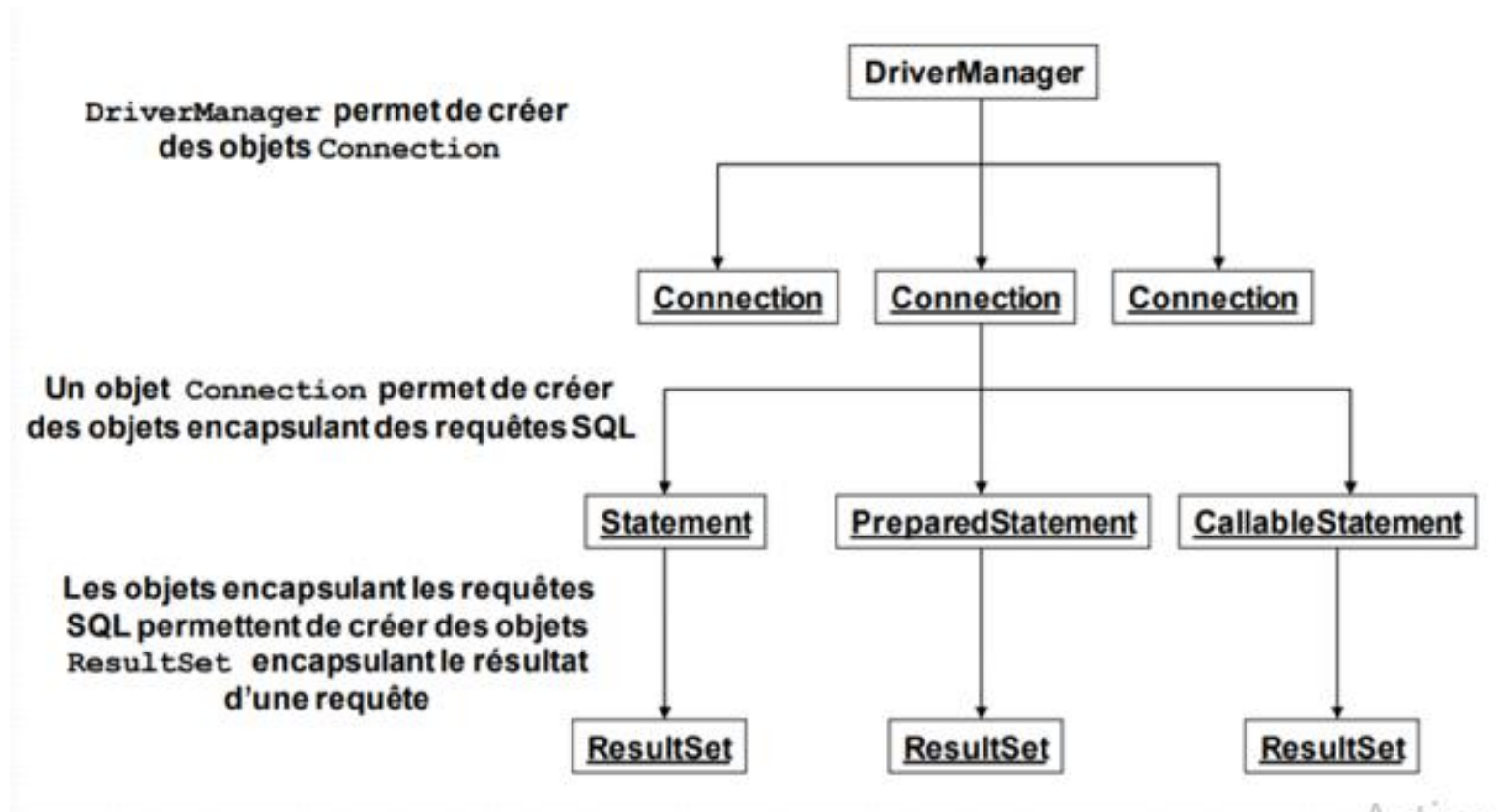


- ▶ **Driver**: envoie une instance de **Connection**
- ▶ **Connection**: connexion à une BD
- ▶ **Statement**: instruction SQL
- ▶ **PreparedStatement**: instruction SQL paramétrée
- ▶ **CallableStatement**: procédure stockée dans la base
- ▶ **ResultSet**: n-uplets récupérés par une instruction SQL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

Java DataBase Connectivity (JDBC)

API JDBC

- Objets instanciés à partir des types Java définis dans java.sql



Processus JDBC

Etapes d'un programme utilisant JDBC:

1. Mettre en place le pilote ou driver.
2. Établir une connexion avec une source de données.
3. Effectuer les requêtes.
4. Utiliser les données obtenues pour des affichages, des traitements statistiques, etc.
5. Mettre à jour les informations de la source de données.
6. Terminer la connexion
7. Éventuellement, recommencer en 1.

JDBC: Processus de connexion

```
import java.sql.DriverManager; // gestion des pilotes
import java.sql.Connection; // une connexion à la BD
import java.sql.Statement; // une instruction
import java.sql.ResultSet; // un résultat (lignes/colonnes)
import java.sql.SQLException; // les erreurs liées à la BD

public class JDBCExemple {
    // 0- chargement du pilote (pour JDBC antérieur à JDBC4)
    // 1- ouverture de connexion
    // 2- exécution d'une requête
    // 3- programme principal
}
```

JDBC: Processus de connexion

1 Chargement du Driver

- ▶ La classe DriverManager gère les différentes instances de Driver. Elle doit être chargée en mémoire:
 - Utiliser la méthode `Class.forName(String)` avec en paramètre le nom complet de la classe du driver
 - La classe crée une instance d'elle-même et enregistre cette instance auprès de la classe DriverManager
- ▶ Exemples:
 - `Class.forName("oracle.jdbc.driver.OracleDriver");`
 - `Class.forName("com.mysql.jdbc.Driver");`
 - `Class.forName("org.postgresql.Driver");`

JDBC: Processus de connexion


1 Chargement du Driver

► Exemple

```
import java.sql.*;

public class premiereConnexion {
    public static void main (String[ ] args) {
        try{
            Class.forName("oracle.jdbc.driver.OracleDriver");
        }
        catch(ClassNotFoundException e) {
            System.out.println("Impossible de charger le pilote");
            System.exit(1);
        }
        System.out.println("Pilote chargé");
    }
}
```

Chargement d'un pilote Oracle



JDBC: Processus de connexion

Chargement du Driver à partir de JDBC 4

- ▶ A partir de JDBC 4.0, il faut:
 - Connaître le nom de la base (Postgresql, MySQL, access, etc...)
 - Lui associer un pilote (ou driver): chercher sur votre moteur de recherche: JDBC + nom de votre BD
 - Télécharger le zip ou le .jar : Exp: mariadb-connector-java-[version]-bin.jar
- ▶ Sous IDE, ajouter l'archive du pilote (.jar) au projet
 - Sur le répertoire Librairies, ajouter .jar

JDBC: Processus de connexion

2 Connexion à la BD

- ▶ On utilise la méthode **getConnection()** de la classe **DriverManager** avec l'URL en argument:
 - `Connection maConnexion = DriverManager.getConnection(url);`
 - URL: Identification de la BD de forme générale: `jdbc:Driver:Base`
 - La forme exacte dépend de la BD avec des informations spécifiques pour établir la connexion (`jdbc:oracle:thin:@serveur:port:base`)
Si login et mot de passe: `login/motDePasse@ordinateur:port:base`
- ▶ Ou bien
 - `Connection conn=DriverManager.getConnection(url, user, password);`
- ▶ 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.

JDBC: Processus de connexion

2 Connexion à la BD en local

- ▶ (MariaDB est la BD issue de MySQL maintenue par le monde libre depuis qu'Oracle a racheté MySQL)
- ▶ `String url = "jdbc:mariadb://localhost:3306/maBD";`
`Connection conn = DriverManager.getConnection(url, "root", "root");`

Paramètres indiqués par votre
environnement de serveur
APACHE local

MySQL

MySQL peut être administrée via [phpMyAdmin](#).

Pour vous connecter au serveur MySQL dans vos propres scripts PHP, utilisez les paramètres suivants:

Hôte	localhost
Port	3306
Utilisateur	root
Mot de passe	root
Socket	/Applications/MAMP/tmp/mysql/mysql.sock

JDBC: Processus de connexion

Exemple de connexion à la BD

► Suite de l'exemple:

```
System.out.println("Pilote chargé");
String url = "jdbc:oracle:thin:user/mdp";
url+= " @://localhost/test ";

Connection maConnexion = null;

try {
    maConnexion = DriverManager.getConnection(url);
}

catch(SQLException e) {

    System.out.println("Impossible de se connecter à l'url :"+ url);

    System.exit(1);

}
```

JDBC: Processus de connexion

Exemple de connexion à la BD

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
    Connection connect = DriverManager.getConnection("jdbc:mysql://localhost/test", user, password);....  
}  
catch (ClassNotFoundException e){  
    System.out.println("Impossible de charger le pilote");  
    System.exit(1);  
}  
catch (SQLException e) {  
    System.out.println("Impossible de se connecter à l'url :"+ url);  
    System.exit(1);  
}  
}
```

JDBC

3 Déclaration et exécution de la requête

- ▶ Ouvrir la connexion
 - ▶ 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
 - ▶ Exécuter une requête sur ce `Statement`
 - ▶ Si `SELECT`, traiter le résultat obtenu (`ResultSet`)
 - ▶ Fermer le `Statement`
 - ▶ Fermer la connexion
-
- ▶ Un seul `ResultSet` par `Statement`

JDBC

3 Déclaration et exécution de la requête

- ▶ Une fois une Connection créée on peut l'utiliser pour créer et exécuter des requêtes (statements) SQL.
- ▶ La connexion retournée par la méthode getConnection() est une connexion ouverte, ce qui permet de passer les instructions SQL vers les SGBD.
- ▶ Il faut tout d'abord demander la création du statement: chaque requête s'exécute par un objet **Statement**:
`Statement monInstruction = maConnexion.createStatement();`
- ▶ Ensuite, il faut déclarer le code SQL de la requête:
`ResultSet monResultat = monInstruction.executeQuery(maRequête);`
- ▶ `excuteQuery(String q)` renvoie un objet de type ResultSet permet de décrire la table des résultats

JDBC

3 Déclaration et exécution de la requête

- ▶ 3 types (interfaces) d'objets Statement:
 - Statement : requêtes simples (SQL statique)
 - PreparedStatement: requêtes précompilés (SQL dynamique si supporté par SGBD) qui peuvent améliorer les performances
 - CallableStatement: encapsule procédures SQL stockées dans le SGBD
- ▶ 3 formes (méthodes) d'exécutions:
 - executeQuery: pour les requêtes qui retournent un résultat (SELECT) résultat accessible au travers d'un objet ResultSet
 - executeUpdate: pour les requêtes qui ne retournent pas de résultat (INSERT, UPDATE, DELETE, CREATE TABLE et DROP TABLE)
 - execute: quand on ne sait pas si la requête retourne ou non un résultat, procédures stockées

3 Déclaration et exécution de la requête

► 3 méthodes selon le type de la requête effectuée:

- Pour des select, retour d'un ResultSet:

```
ResultSet rs = ps.executeQuery();
```

On traite ensuite les lignes retournées, en séquence

- Pour des ordres DML (requêtes insert, delete ou update ou dropTable ou ordre create), retour d'un int:

```
int s = ps.executeUpdate();
```

Le nb de lignes traitées pour DML, zéro pour dropTable ou create

- Pour tout ordre SQL, retour d'un booléen:

```
boolean b = ps.execute(String req);
```

Renvoie vrai si le résultat est un ResultSet

JDBC: Exemple

3 Déclaration et exécution de la requête

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

```
ResultSet rs = stmt.executeQuery("SELECT nom, code_client FROM Clients");
```

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

Nom	Prénom	Code_client	Adresse
DUPONT	Jean	12345	135 rue du Lac
DUROND	Louise	12545	13 avenue de la Mer
...			
ZORG	Albert	45677	8 Blvd De la Montagne

Nom	Code_client
DUPONT	12345
DUROND	12545
...	
ZORG	45677

- Les rangées du `ResultSet` se parcourent ligne (row) par ligne avec la méthode `next()` qui renvoie `VRAI` s'il reste des lignes à lire et `FAUX` sinon.
`while (rs.next()) {...Exploiter les données}`

JDBC

3 Lecture des données

- ▶ Les colonnes sont référencées par leur numéro ou par leur nom
- ▶ L'accès aux valeurs des colonnes se fait par des méthodes `getXXX(String nomCol)` ou `getXXX(int numCol)` où XXX dépend du type de la colonne dans la table SQL.
- ▶ 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);
```

```
System.out.println("Row=" + i + " " + s); }
```

Attention! En SQL les numéros de colonnes débutent à 1

JDBC: Exemple

3 Déclaration et exécution de la requête

```
public static void main(String [ ] args) throws SQLException{  
    ...System.out.println("Pilote chargé");  
    String url = "jdbc:oracle:thin:ffioresen/mdp";  
  
    url+= "@oracle.iut-orsay.fr:1521:etudom";  
    Connection maConnexion = DriverManager.getConnection(url);  
  
    Statement monInstruction = maConnexion.createStatement();  
  
    ResultSet monResultat = monInstruction.executeQuery("SELECT numfilm, titre FROM ens2004.film where  
    titre like 'A%'");  
    while (monResultat.next()){  
        String titre = monResultat.getString("titre"); //monResultat.getString(2);  
        int numero = monResultat.getInt("numfilm"); // monResultat.getInt(1);  
        System.out.println(numero + "\t " + titre);  
        System.out.println(" " );  
    }  
}
```

JDBC

Equivalence des types Java-SQL

- **getXXX**: 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	getString	String
VARCHAR	getString	String
NUMERIC	getBigDecimal	java.Math.BigDecimal
DECIMAL	getBigDecimal	java.Math.BigDecimal
BIT	getBoolean	boolean <i>Boolean</i>
TINYINT	getByte	byte <i>Integer</i>
SMALLINT	getShort	short <i>Integer</i>
INTEGER	getInt	int <i>Integer</i>
BIGINT	getLong	long <i>Long</i>
REAL	getFloat	float <i>Float</i>
FLOAT	getDouble	double <i>Double</i>
DOUBLE	getDouble	double <i>Double</i>
DATE	getDate	java.sql.Date
TIME	getTime	java.sql.Time
TIME STAMP	getTimestamp	java.sql.Timestamp

Peut être appelée sur
n'importe quel type de valeur

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

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

JDBC

Traitement des valeurs nulles

- ▶ Quand les valeurs nulles sont acceptées, il y a conversion automatique vers une valeur “acceptable ” selon le type retourné par getXXX()
- ▶ **null** si getXXX() retourne un type objet (getString(), getDate(), ...)
- ▶ **0** si getXXX() retourne un type numérique (getInt(), getDouble(),....)
- ▶ **false** pour getBoolean()
- ▶ **wasNull()** : méthode de ResultSet

JDBC

Traitement des valeurs nulles

```
while (rs.next( )) {
```

```
    System.out.print(rs.getString("NOM"));
```

```
    System.out.print(" "+rs.getString("PRENOM")+ " ");
```

```
    System.out.println(rs.getBoolean("MARIE")? "Marié" : "Non Marié");}
```

NOM	PRENOM	ADRESSE	CODE_POSTAL	DATE_NAISS	MARIE
TOTO	Riri	NULL	38920	NULL	NULL
TITI	Fifi	NULL	73550	1961-03-14	1
TUTU	Mimi	Rue Chose	73350	1957-06-10	0

► **wasNull** true si on vient de lire une valeur NULL, false si non

```
while (rs.next( )) {
```

```
    System.out.print(rs.getString("NOM"));
```

```
    System.out.print(" "+rs.getString("PRENOM")+ " ");
```

```
    boolean marié = rs.getBoolean("MARIE");
```

```
    if (rs.wasNull()) System.out.println("?");
```

```
    else
```

```
        System.out.println(marié? "Marié" : "Non Marié");}
```

```
TITI Fifi Marié
TOTO Riri Non Marié
TUTU Mimi Non Marié
...
```

```
TITI Fifi Marié
TOTO Riri ?
TUTU Mimi Non Marié
...
```

JDBC

Déclaration et exécution de plusieurs requêtes

- ▶ On peut faire plusieurs requêtes sur un même 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);
```

```
//exploitation des résultats de myQuery1
```

```
while(rs1.next()){...}
```

```
ResultSet rs2 = stmt.executeQuery(myQuery2);
```

```
while(rs2.next()) {...}
```

- ▶ Si plus qu'une requête simultanément alors créer et utiliser autant d'objets Statement
- ▶ Un seul ResultSet par Statement peut être ouvert à la fois

JDBC

ResultSet et modification des données

- ▶ Si on veut par exemple modifier les données d'un ResultSet avant de les afficher :

- ▶ Méthodes `setXXX()` du `ResultSet`

- `setType (indice | " nom_colonne ", valeur)`

```
rs.setInt(1, 75);
```

```
rs.setString(2, " Columbian ");
```

- ▶ **Remarques**

- L'indice commence à 1, il fait référence au numéro de colonne du ResultSet (celui défini dans l'ordre SELECT) et nom au numéro de colonne de table.

JDBC

Accès BD en mise à jour

- Pour exécuter un update, un insert ou un delete :

```
Statement monInstruction = maConnexion.createStatement();
```

```
int monResultat = monInstruction.executeUpdate(maRequête);
```

- Exemple: création de table

```
final String MA_REQUETE = " create table Employees as select * from scott.emp ";
```

```
conn = new Connection();
```

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

```
int r = st.executeUpdate(MA_REQUETE);
```

JDBC

Accès BD en mise à jour

```
public static void main(String [ ] args) throws SQLException{  
    ....while(monResultat.next())  
    {...}  
    int num =1112;  
    String client = "charlie ";  
    int nbLignes = monInstruction.executeUpdate("INSERT INTO  
location(numExemplaire, login, dateLocation) VALUES(num, client, SYSDATE)  
");  
    System.out.println(nbLignes + " ligne(s) insérée(s) ");  
    ....}
```

JDBC

Modification de la définition de données

- ▶ Pour modifier la structure de la base:
- ▶ La classe Statement a : `execute(ordreSQL)`
 - L'ordre SQL correspond à la chaîne de caractères contenant l'ordre à exécuter
 - Elle retourne un boolean qui est vrai si il n'y a pas eu d'erreur à l'exécution

JDBC

Fermeture de la connexion

- ▶ Quand une Connection n'a plus d'utilité prendre soin de la fermer!
- ▶ Libération de mémoire et surtout des ressources de la base de données détenues par la connexion
- ▶ Libérer les objets ResultSet et Statement : `monObjet.close();`
- ▶ **Remarque** : Si erreur lors du dialogue avec la BD, l'instruction `close` n'est pas exécutée. La connexion reste ouverte!

```
try{
Connection conn =
DriverManager.getConnection("jdbc:odbc:companydb ",user, passwd);
//utilisation de la connexion pour dialoguer avec la BD
conn.close(); // fermeture de la connexion
}
catch(SQLException e){...}
```

JDBC

Fermeture de la connexion

- ▶ Garantir la fermeture de la connexion: **clause finally**

```
Connection conn = null;
try {
    Connection conn = DriverManager.getConnection("jdbc:odbc:companydb ",user, passwd);
    ...//utilisation de la connexion pour dialoguer avec la BD...
    // fermeture de la connexion
    conn.close();
} catch(SQLException e) {...}
finally{
    try{ if (conn!=null)
        conn.close();}
    catch(SQLException e) {e.printStackTrace();}}
```

Pour que conn soit connue dans le bloc finally
Le compilateur impose d'initialiser conn

conn peut ne pas avoir été initialisé

close peut provoquer une SQLException

JDBC

Fermeture de la connexion

- ▶ Pour garantir la fermeture de la connexion: utilisation d'un try avec ressources (Java7) au lieu de la clause finally

```
try(Connection conn =  
    DriverManager.getConnection("jdbc:odbc:companydb ",user, passwd);){  
    ...//utilisation de la connexion pour dialoguer avec la BD ...}  
catch(SQLException e){...}
```

- ▶ Les ressources sont automatiquement fermées à la fin de l'instruction try

Exemple d'accès à une base de données MySQL via JDBC

► Etape 1: Bases de données MySQL

- Créer une base de données MySQL 'mabase' , y ajouter une table Personne (avec 3 champs: id, nom et prénom) et insérer des données.

	Champ	Type	Interclassement	Attributs	Null	Défaut	Extra
<input type="checkbox"/>	<u>id</u>	int(11)			Non	Aucun	auto_increment
<input type="checkbox"/>	nom	varchar(20)	latin1_swedish_ci		Non	Aucun	
<input type="checkbox"/>	prenom	varchar(20)	latin1_swedish_ci		Non	Aucun	


Exemple d'accès à une base de données MySQL via JDBC

Etape2: MySQL JDBC Driver

- Télécharger le driver JDBC Java mysql connecter et décompresser l'archive .zip ou .jar
<https://downloads.mysql.com/archives/c-j/>

MySQL Product Archives

MySQL Connector/J (Archived Versions)

 Please note that these are old versions. New releases will have recent bug fixes and features!
To download the latest release of MySQL Connector/J, please visit [MySQL Downloads](#).

Product Version:

Operating System:

Platform Independent (Architecture Independent), Compressed TAR Archive

Apr 20, 2020

3.2M

[Download](#)

(mysql-connector-java-5.1.49.tar.gz)

MD5: e7bc11a55398bad0ea8548163deabaa8 | [Signature](#)

Platform Independent (Architecture Independent), ZIP Archive

Apr 20, 2020

3.5M

[Download](#)

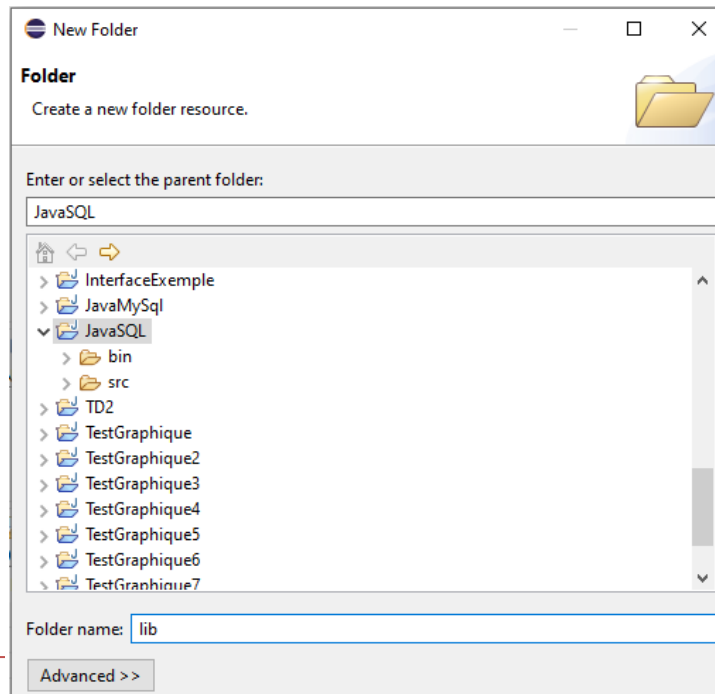
(mysql-connector-java-5.1.49.zip)

MD5: 5ecd588e13f14de07faa5c67f5caf3f1 | [Signature](#)

Exemple d'accès à une base de données MySQL via JDBC

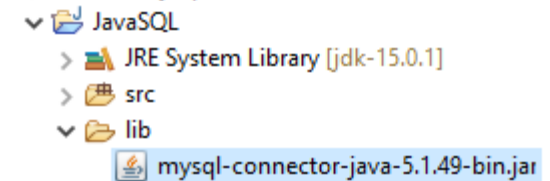
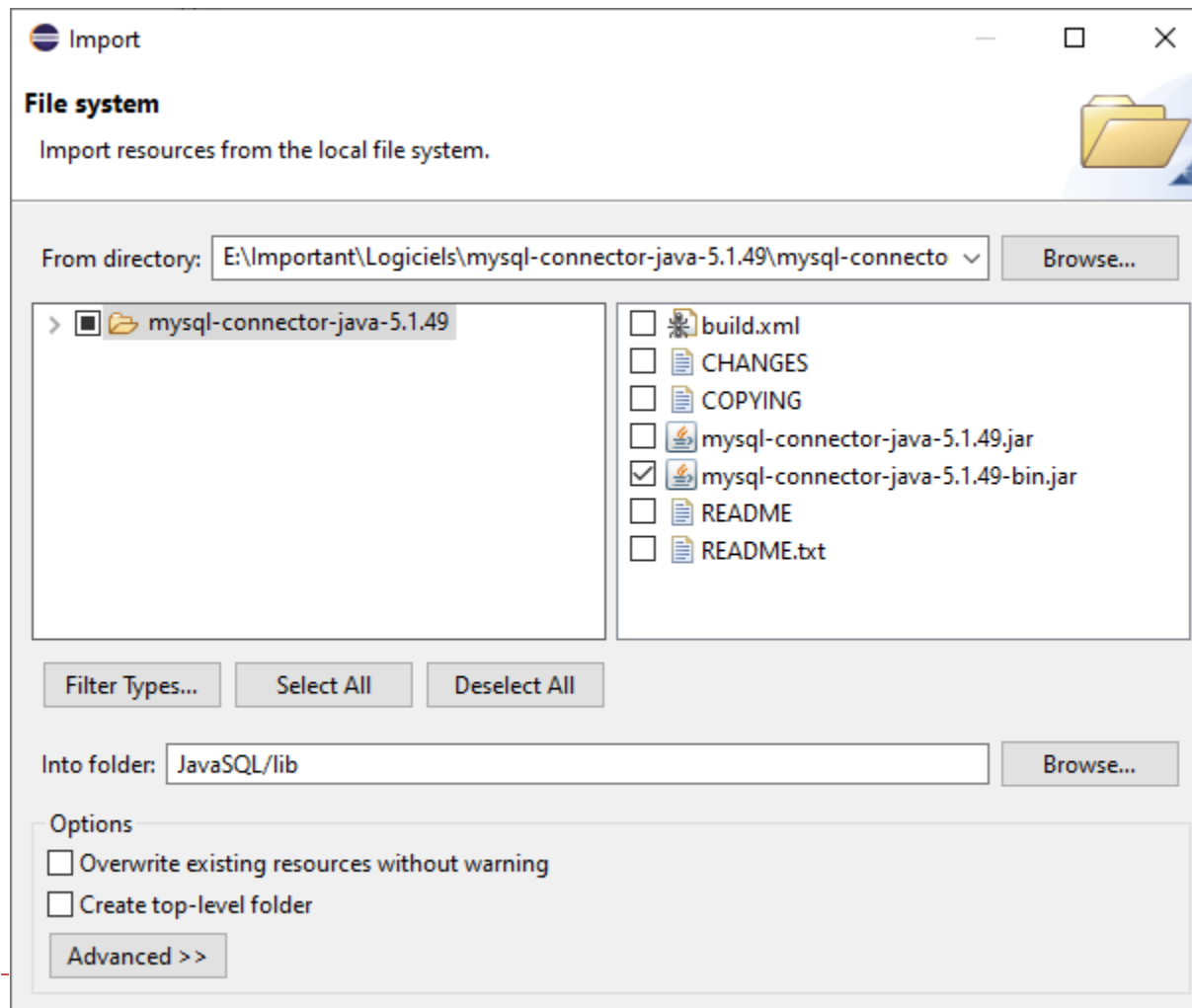
Etape2: MySQL JDBC Driver

- ▶ Intégrer le driver dans votre projet
 - Faire un clic droit sur le nom de votre projet et aller dans New > Folder
 - Renommer le répertoire lib puis valider



Exemple d'accès à une base de données MySQL via JDBC

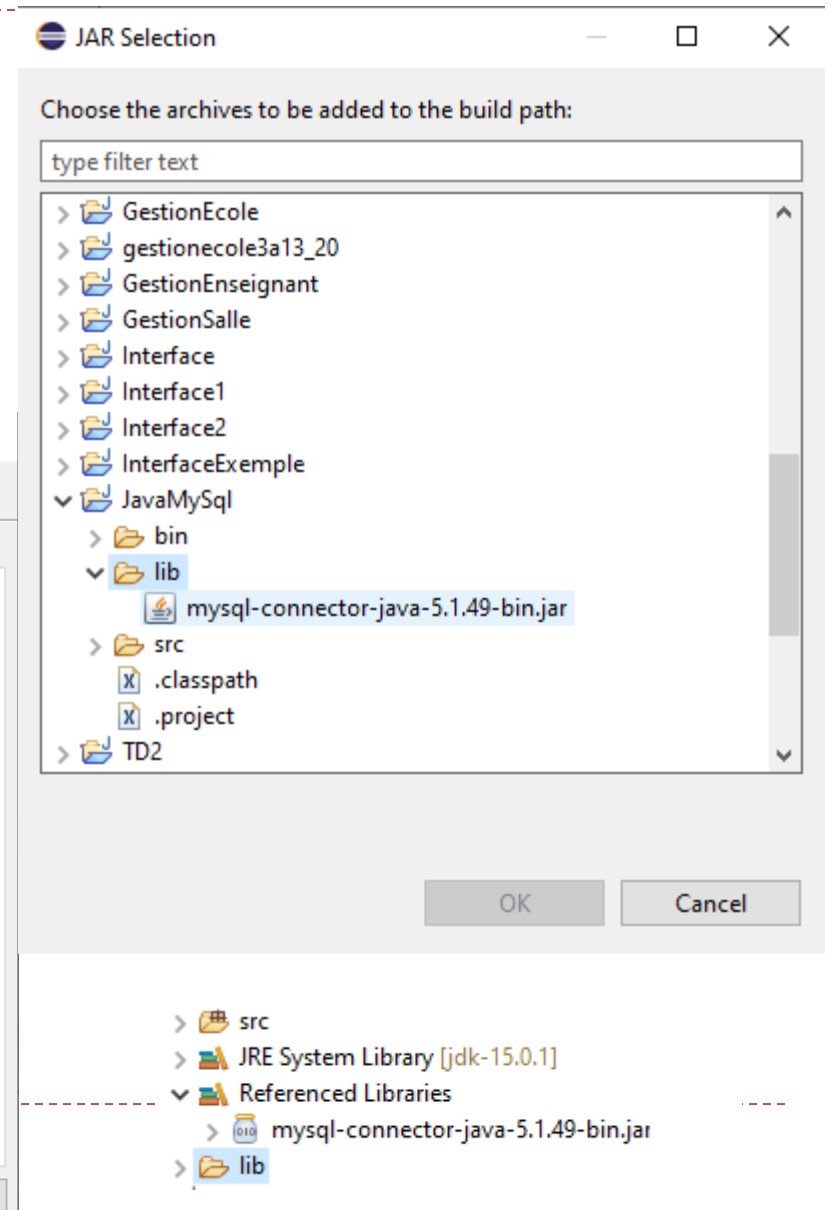
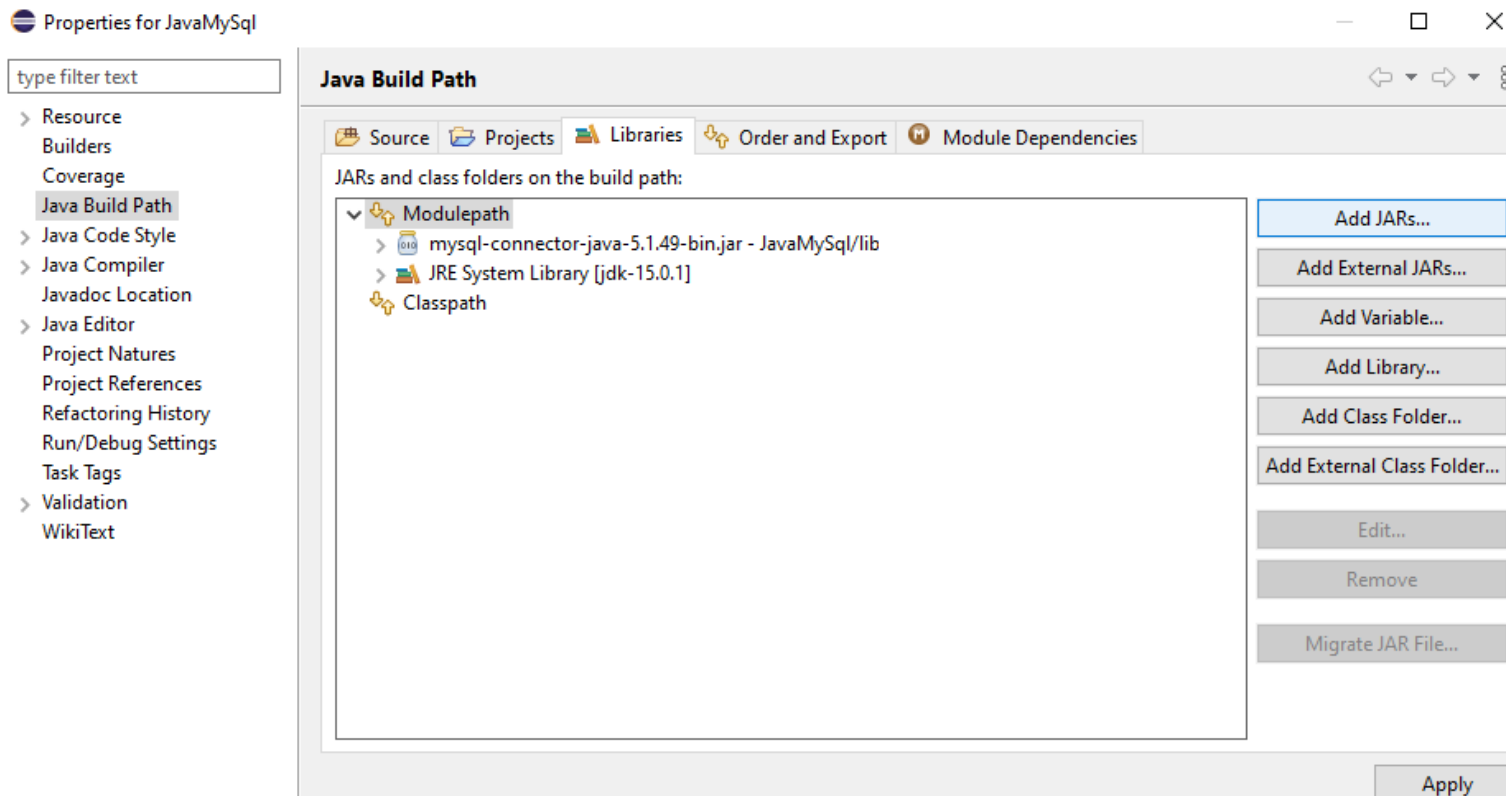
- Copier le .jar de l'archive décompressée dans lib



Exemple d'accès à une base de données MySQL via JDBC

Etape2: MySQL JDBC Driver

- ▶ Ajouter JDBC au path du projet
 - Faire clic droit sur .jar qu'on a placé dans le lib
 - Aller dans Build Path et choisir Configure Build Path



Exemple d'accès à une base de données MySQL via JDBC

Etape3:

- Charger le driver JDBC (pour MySQL dans notre cas)
- Etablir la connexion avec la base de données
- Créer et exécuter des requêtes SQL

Exemple d'accès à une base de données MySQL via JDBC

```
import java.sql.*;
public class JavaMySQL {
    public static void main(String[] args) {
        //Information d'accès à la base de données
        String url = "jdbc:mysql://localhost:3306/mabase";
        String login = "root";
        String passwd = "root";
        Connection conn = null;
    try {
        //1: Chargement du driver
        Class.forName("com.mysql.jdbc.Driver");
        //2: connexion à la base
        conn = DriverManager.getConnection(url, login, passwd);
        //3: Création d'un Statement
        Statement st = conn.createStatement();
        //4 : créer une requête SQL
        ResultSet rs = st.executeQuery("select * from Personne");
        //5: parcours des données
        while(rs.next()) {
            System.out.println("id:" + rs.getInt(1) + "\t Nom:" + rs.getString(2) +
                               "\t Prenom:" + rs.getString(3));
        }
        conn.close();
    }
    catch(ClassNotFoundException e) {
        System.out.println("Impossible de charger le driver");
        System.exit(1);
    }
    catch(SQLException e) {
        System.out.println("Impossible de se connecter à l'url");
        System.exit(1);
    }
    finally {
        try {if (conn!=null)
            conn.close();}
        catch(SQLException e) {
            e.printStackTrace();}
        }
    }
}
```

Complément JDBC

- ▶ Requêtes paramétrables « PreparedStatement »
- ▶ Requêtes procédures stockées « CallableStatement »
- ▶ Accès aux méta-données

JBDC

Préparer/exécuter une requête paramétrable/précompilé

- ▶ Utilisation de SQL préformaté via un objet `PreparedStatement` ou envoi d'une requête paramétrable au SGBD pour précompilation (exécute une requête SQL dynamique).
- ▶ Objectif: Méthode plus rapide que Statement standard car SGBD n'analyse qu'une seule fois la requête, plus fiable ...
 - Interface: `PreparedStatement`
 - Méthode: `conn.prepareStatement(req)`
 - A utiliser pour des requêtes qui sont exécutées plusieurs fois (à quelques variables près)
 - Compilés (parsed) par le SGBD une seule fois
 - Au lieu de valeurs, on utilise '?'
 - Ce sont donc des statements avec variables, dont les valeurs réelles sont données dans un 2^{ème} temps.

JBDC

Préparer/exécuter une requête paramétrable

- ▶ Création d'un `PreparedStatement` paramètres formels à l'aide ?

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

- ▶ Passage des paramètres effectifs: `setXXX(indice, valeur)` où `XXX` représente le type du paramètre
- ▶ `ps.setString(1, "Person ");`
- ▶ Invocation et exploitation des résultats (SQL statique)
- ▶ Exécution de `prepareStatement`
 - `executeQuery()` pour une requête
 - `executeUpdate()` pour tous les autres ordres SQL

JBDC

Exemple requête précompilée PreparedStatement

- ▶ Exemple Select: on récupère un ou plusieurs ResultSet

```
PreparedStatement ps= conn.prepareStatement("select * from client where  
nocli =? ");
```

```
ps.setInt(1, numcli); //variable numcli avec une valeur
```

```
ResultSet rs = ps.executeQuery();
```

- ▶ Exemple de Delete: on récupère un entier

```
ps = conn.prepareStatement("delete from client where nocli =? ");
```

```
ps.setInt(1, numcli);
```

```
int nbLigne = ps.executeUpdate();
```

JBDC

Exemple requête précompilée PreparedStatement

► Update utilisé dans une boucle

```
PreparedStatement updateVentes;
```

```
String updateString = "update CAFE" + "set VENTES =? WHERE  
NOM_CAFE LIKE ? ";
```

```
updateVentes = conn.prepareStatement(updateString);
```

```
int[ ] VentesDeLaSemaine = {175, 150, 60, 155, 90};
```

```
String [ ] cafes = {"Colombian ", "French_Roast ", "Espresso ", "Colombian_Decaf ", "French_Roast_Decaf "};
```

```
for(int i =0; i<cafes.length; i++) {
```

```
    updateVentes.setInt(1, VentesDeLaSemaine[i]);
```

```
    updateVentes.setString(2, cafes[i]);
```

```
    updateVentes.executeUpdate();}
```

JBDC

Exemple requête précompilée PreparedStatement

```
PreparedStatement pst = maConnexion.prepareStatement("update tarif set  
prix = prix+? where numvaoyage =? and datedeb>? ");  
  
for(int i =0; i++) { //10 fois update  
    pst.setInt(1, saisieMajoration());  
    pst.setInt(2, saisieVoyage());  
    pst.setString(3, saisieDate());  
    int compteur = pst.executeUpdate();  
    System.out.println(compteur + "ligne (s) mise (s) à jour. ");}  
  
pst.close();
```

Types de Méta-données

- ▶ Méta-données: informations décrivant:
 - La BD (interface `DatabaseMetaData`)
 - 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`

JDBC

Meta informations du ResultSet

- ▶ `ResultSetMetaData m = rs.getMetaData();`
- ▶ Informations disponibles:
 - nombres de colonnes : `getColumnCount()`
 - type associé à une colonne : `getColumnType(int col)`
 - Le nom d'une colonne: `columnName(int col)`, etc

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

← Renvoie true si requête de type Query false sinon (Update)

← Accès au ResultSet produit par la requête

JDBC

Contenu de l'interface ResultSetMetaData

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

JDBC

Exemple Meta informations du ResultSet

```
ResultSet monResultat = monInstruction.executeQuery("select numfilm, titre from  
ens2004.film ");
```

```
ResultSetMetaData rsmd = monResultat.getMetaData();
```

```
int nombreDeColonnes = rsmd.getColumnCount();
```

```
for(int i = 1; i <= nombreDeColonnes; i++) {
```

```
String nomColonne = rsmd.getColumnName(i);
```

```
String nomType = rsmd.getColumnTypeName(i);
```

```
System.out.println("La colonne "+ i + "est " + nomColonne + "dont le nom de type  
Oracle est "+ nomType) ; }
```


JDBC

Procédures stockées / CallableStatement

- ▶ 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
- ▶ Utilisation des procédures stockées depuis JDBC via l'interface CallableStatement
 - Syntaxe unifiée indépendante de la manière dont celles-ci sont gérées par le SGBD (chaque SGBD a sa propre syntaxe)
 - Utilisation possible de la valeur de retour
 - Gestion des paramètres IN, OUT, INOUT

JDBC

Procédures stockées / CallableStatement

- ▶ Comme pour la méthode `prepareStatement`, les paramètres sont définis par ?.
- ▶ 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(?, ?) } ");
```

- ▶ Appel sans valeur de retour et sans paramètres

```
CallableStatement proc = conn.callableStatement (" {call  
maProcédure } ");
```

JDBC

Procédures stockées: préparation de paramètres

- ▶ Paramètre en sortie : `registerOutParameter(indice, type)` avec
indice : position du paramètre et son type

```
proc.registerOutParameter(1, Types.DECIMAL, 3);
```

- ▶ Passage des paramètres IN: méthode `setXXX(indice, valeur)` identique à `prepareStatement`.

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

- ▶ Appel : `proc.execute()` : renvoie true si l'exécution a produit un ResultSet et false : pas de retour, ou mise à jour.

```
ResultSet rs = proc.executeQuery();
```

- ▶ Exploitation du ResultSet (=Statement et PreparedStatement)
- ▶ Récupération des paramètres OUT := `getType(indice/ "nom_colonne")`:
`int nb = cs.getInt(1);`

JDBC

Procédure Stockée

- ▶ Soit la fonction définie par :
CREATE OR REPLACE FUNCTION entitule (numero IN voyage.numVoyage%TYPE)
RETURN voyage.nomVoyage%TYPE IS BEGIN select nomvoyage into nom FROM
voyage where numvoyage = numero;
return nom;
- ▶ numero sera de type INTEGER et la fonction renverra du VARCHAR2(60)

CallableStatement cst = maConnexion.prepareCall("{? = call entitule (?,?)}
"); //Création de CallableStatement

cst.setInt(1, 6); //Premier paramètre en entrée valeur = 6

cst.registerOutParameter(1, java.sql.Types.VARCHAR); //Paramètre out

boolean retour = cst.execute(); //Exécution de CallableStatement

String nom= cst.getString(1); // Récupération du résultat