

Développement Java

SQL ET JAVA : JDBC



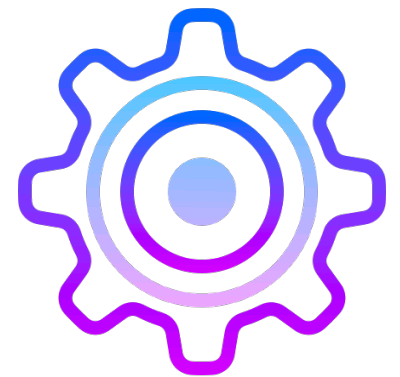
Objectifs pédagogiques

- Connecter une application à une base de données SQL grâce à l'API JDBC présente dans le package `java.sql`.



Une API, c'est quoi ?

- Java **A**pplication **P**rogramming **I**nterface
- Liste de classes faisant partie du JDK
- Inclut tous les packages, classes, interfaces, méthodes, champs, constructeurs



C'est quoi JDBC ?

- **Java DataBase Connectivity**
- API pour permettre un accès aux bases de données avec Java.



L' intérêt d'utiliser JDBC ?

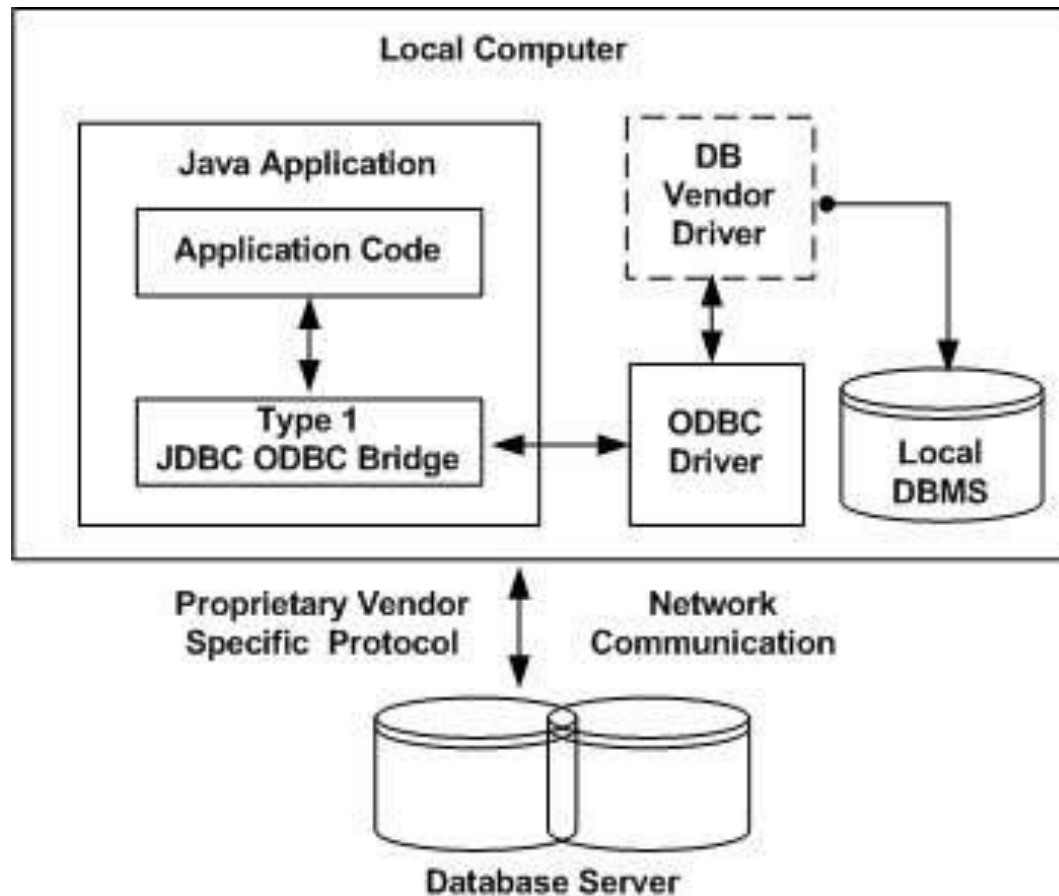
- Grâce à JDBC, les applications Java peuvent :
 - recevoir des données provenant de la base de données
 - envoyer des données provenant de Java
- JDBC a plusieurs type de pilotes

Les différents types de pilotes

- **Pilote de type 1**, un pont JDBC est utilisé pour accéder aux pilotes ODBC installés sur chaque ordinateur client.
- L'utilisation d'ODBC nécessite la configuration sur votre système d'un nom de source de données (DSN) qui représente la base de données cible.
- Lors de la première utilisation de Java, il s'agissait d'un pilote utile car la plupart des bases de données ne prenaient en charge que l'accès ODBC.
- Ce type de pilote est désormais recommandé uniquement pour un usage expérimental ou lorsqu'aucune autre solution n'est disponible.

Les différents types de pilotes

Pilote de type 1

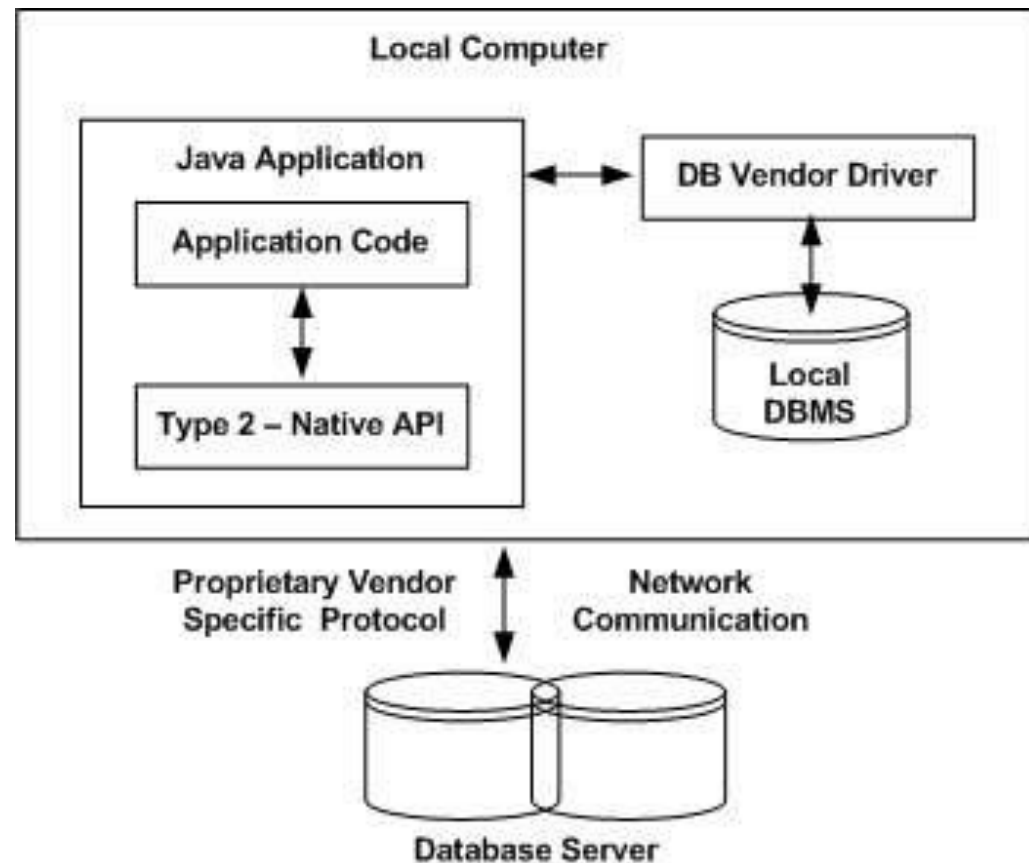


Les différents types de pilotes

- **Pilote de type 2**, les appels d'API JDBC sont convertis en appels d'API C / C ++ natifs, uniques à la base de données
- Ces pilotes sont généralement fournis par les fournisseurs de base de données et utilisés de la même manière que le pont JDBC-ODBC. Le pilote spécifique au fournisseur doit être installé sur chaque ordinateur client.
- Si nous changeons la base de données, nous devons changer l'API native, car elle est spécifique à une base de données et est pour la plupart obsolète à présent, mais vous pouvez réaliser une augmentation de vitesse avec un pilote de type 2, car elle élimine le temps système ODBC.
- Le pilote Oracle Call Interface (OCI) est un exemple de pilote de type 2

Les différents types de pilotes

Pilote de type 2



Les différents types de pilotes

- **Pilote de type 3**, une architecture 3-tier est utilisée pour accéder aux bases de données (Client - Serveur d'application – Serveur secondaire)
- Les clients JDBC utilisent des sockets réseau standard pour communiquer avec un serveur d'applications middleware.
- Les informations de socket sont ensuite traduites par le serveur d'applications middleware dans le format d'appel requis par le SGBD, puis transmises au serveur de base de données
- Ce type de pilote est extrêmement flexible, car il ne nécessite aucun code installé sur le client et un seul pilote peut en réalité fournir un accès à plusieurs bases de données

Architecture 3-tier ?

- **Le client**

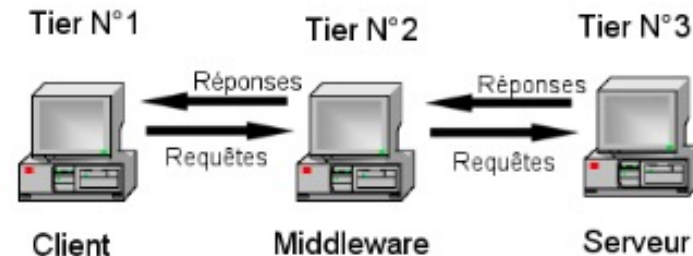
le demandeur de ressources

- **Le serveur d'application**

(appelé aussi *middleware*) le serveur chargé de fournir la ressource mais faisant appel à un autre serveur

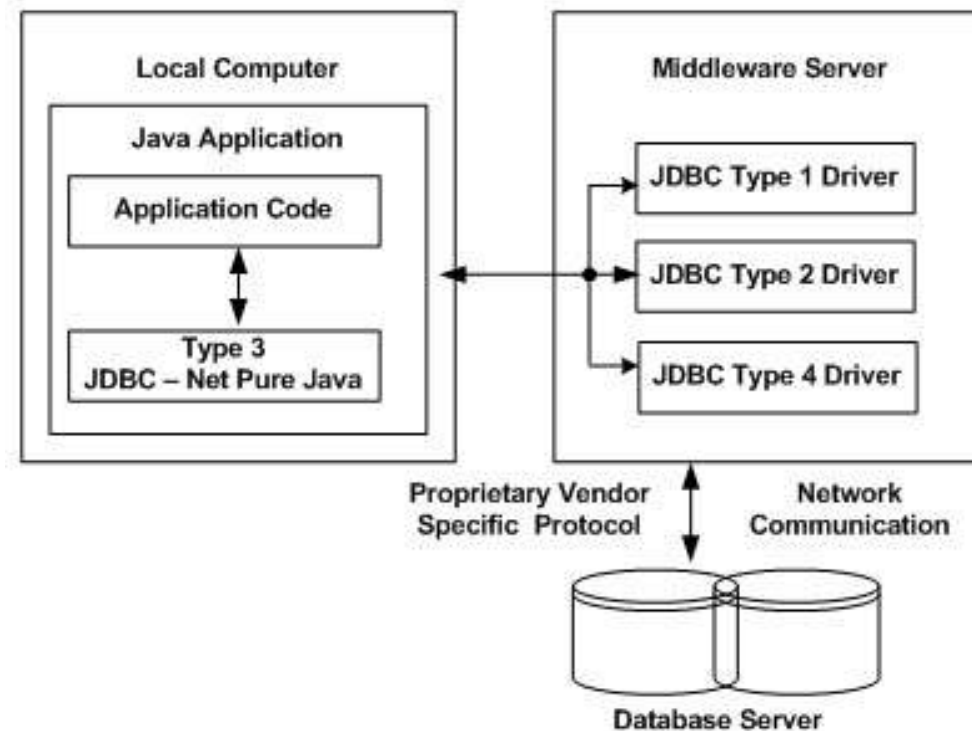
- **Le serveur secondaire**

(généralement un serveur de base de données), fournissant un service au premier serveur



Les différents types de pilotes

Pilote de type 3



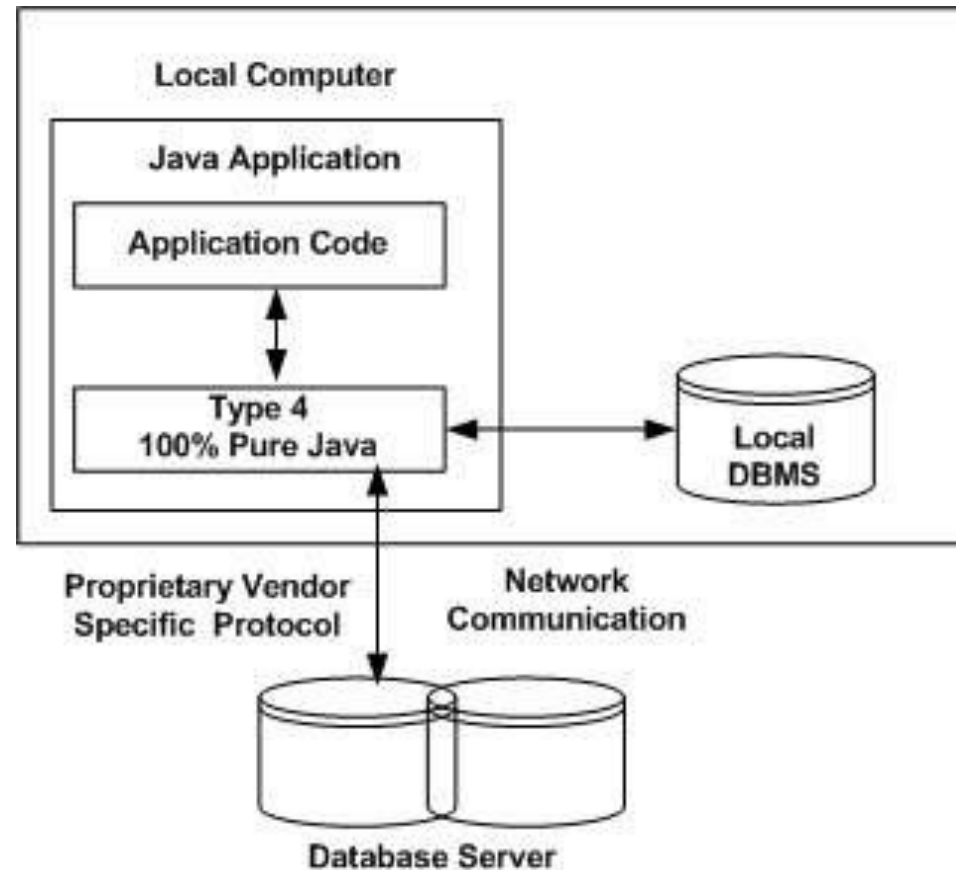
Les différents types de pilotes

- **Pilote de type 4**, un pilote Java pur communique directement avec la base de données du fournisseur via une connexion socket.
- Il s'agit du pilote le plus performant disponible pour la base de données. Il est généralement fourni par le fournisseur lui-même.
- Ce type de pilote est extrêmement flexible, vous n'avez pas besoin d'installer de logiciel spécial sur le client ou le serveur. De plus, ces pilotes peuvent être téléchargés dynamiquement.

Le pilote Connector/J de MySQL est un bon exemple. En raison de la nature exclusive de leurs protocoles réseau, les fournisseurs de bases de données fournissent généralement des pilotes de type 4

Les différents types de pilotes

Pilote de type 4



Choisir le type du pilote

- **Le pilote de type 1** n'est pas considéré comme un pilote de niveau de déploiement et est généralement utilisé à des fins de développement et de test uniquement. **Absent dans Java 8**
- Si vous accédez à un type de base de données, tel qu'Oracle, Sybase ou IBM, le type de **pilote préféré est de type 4**.
- Si votre application Java accède à plusieurs types de bases de données en même temps, **le type 3 est le pilote préféré**.
- **Les pilotes de type 2** sont utiles dans les situations où un **pilote de type 3** ou de **type 4** n'est pas encore disponible pour votre base de données.

Les différentes base de données

- Il existe plusieurs bases de données :
 - PostgreSQL
 - MySQL
 - SQL Server
 - Oracle
 - Access
 - MariaDB
- Chacune a ses spécificités
 - Oracle est payant
 - MySQL est facile d'utilisation et flexible sur ses types
 - PostgreSQL est gratuit et opensource

Utilisation de MySQL avec JDBC

- Utilisation des classes java de JDBC :

```
import java.sql.*;
```

- 4 classes importantes :
 - DriverManager
 - Connection
 - Statement / PreparedStatement
 - ResultSet

Utilisation de MySQL avec JDBC

- Détails des classes importantes :
 - **DriverManager** : charge et configure le pilote de la base de données
 - **Connection** : réalise la connexion et l'authentification à la base de données
 - **Statement / PreparedStatement** : contient la requête SQL et la transmet à la base de données.
 - **ResultSet** : Contient les informations retournées par la base de données dans le cas d'une sélection de données

Les étapes JDBC pour requêter MySQL

- Nous allons voir plusieurs étapes en détails :
 - Etape 1 : Télécharger le driver MySQL
 - Etape 2 : Charger et configurer le driver
 - Etape 3 : Créer une connexion à la base de données
 - Etape 4 : Créer une requête
- Les pré-requis sont :
 - Avoir MySQL installé
 - Avoir une base de données existante (nommée **example**)
 - Avoir une table fruits (avec 3 colonnes **id**, **name**, **expirationDate**)

Etape 1 : télécharger le driver MySQL

- Sur le site MySQL, téléchargez le Connector J :
 - <https://dev.mysql.com/downloads/connector/>
 - Sélectionnez Connector/J
 - Sélectionnez Plateforme indépendante
 - Téléchargez l'archive ZIP
 - Dézippez l'archive ZIP
- Ajouter le fichier jar dans votre projet Java
- Ajouter ce fichier en tant que dépendance

Mettre le fichier jar au sein du projet

- Créer un dossier **lib** à la racine du projet Java
- Mettre le fichier **mysql-connector-java-x.x.xx.jar** dedans
- Ajouter le fichier jar en dépendances du projet
 - Clic-droit sur le projet / Module Settings / Librairies
 - Ajouter avec le (+) Java, Sélectionner le fichier .jar

Etape 2 : charger et configurer le driver

- Utilisation de la classe DriverManager pour charger le driver MySQL :

```
try {  
    Class.forName("com.mysql.cj.jdbc.Driver");  
} catch (Exception e) {  
    e.printStackTrace();  
    System.err.println("Driver MySQL introuvable");  
}
```

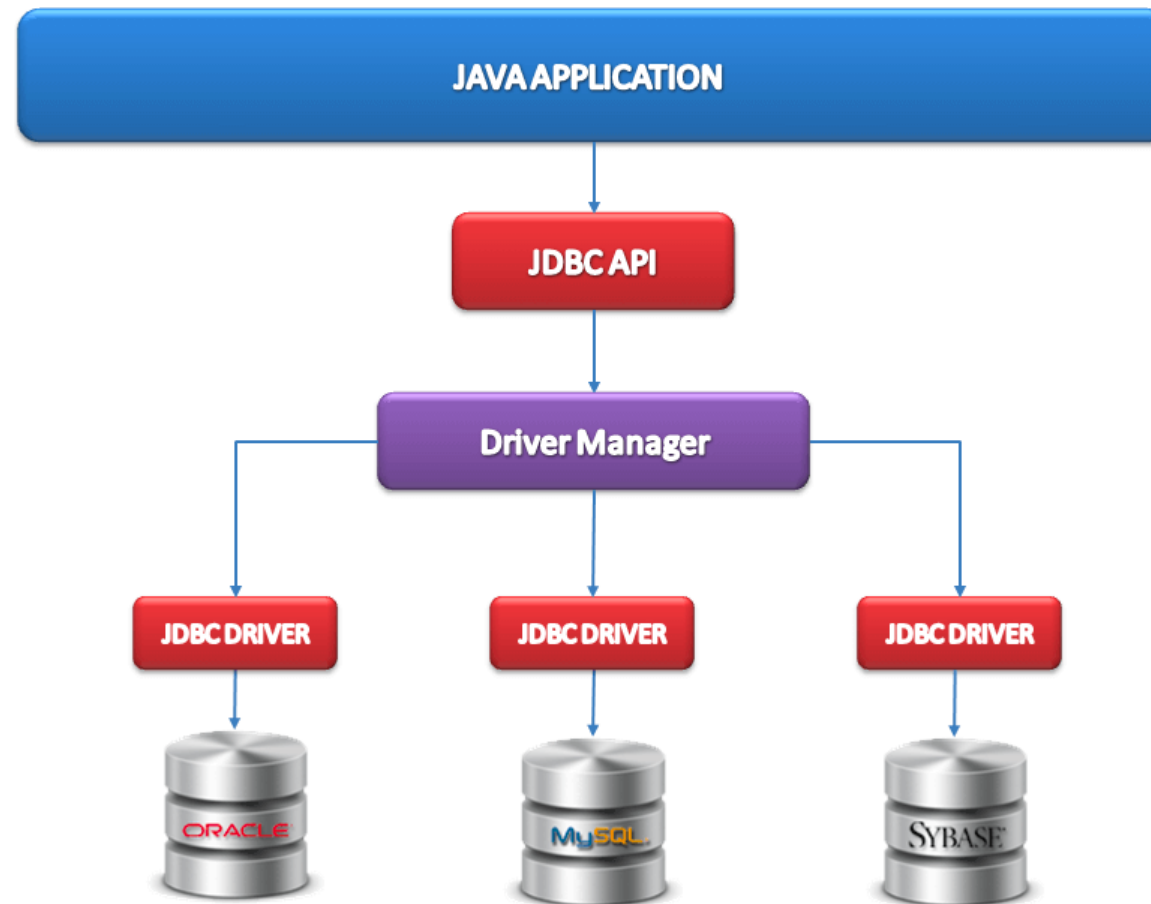
Etape 3 : créer une connexion

- Utilisation de la classe DriverManager pour créer une connexion à votre base de données :

```
try {  
    Connection connection = DriverManager.getConnection(url, user, password);  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

```
url = "jdbc:mysql://localhost:3306/example?serverTimezone=UTC";  
user = "root";  
password = "";
```

Les différents drivers JDBC



Base de données exemple

- Vous pouvez utiliser le schéma suivant :

```
CREATE TABLE `fruits` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `name` varchar(80) NOT NULL,  
  `expirationDate` date NOT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
INSERT INTO `fruits` (`id`, `name`,  
  `expirationDate`) VALUES  
  ('Pomme', '2022-02-11'),  
  ('Orange', '2022-02-11');
```

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input type="checkbox"/>	1	id	int(11)			No	None		AUTO_INCREMENT	Change Drop More
<input type="checkbox"/>	2	name	varchar(80)	utf8_general_ci		No	None			Change Drop More
<input type="checkbox"/>	3	expirationDate	date			No	None			Change Drop More

				id	name	expirationDate
<input type="checkbox"/>	Edit	Copy	Delete	1	Pomme	2022-02-11
<input type="checkbox"/>	Edit	Copy	Delete	2	Orange	2022-02-11

Etape 4 : Création d'une requête

- Utilisation de Statement pour une sélection :

```
String query = "SELECT * FROM fruits";  
try (Statement st = connection.createStatement()) {  
    ResultSet resultSet = st.executeQuery(query);  
  
    while (resultSet.next()) {  
        System.out.println(resultSet.getInt("id"));  
        System.out.println(resultSet.getString("name"));  
    }  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

```
id : 1  
name : Pomme  
id : 2  
name : Orange
```

Etape 4 : Création d'une requête

- Utilisation de PreparedStatement pour un SELECT id :

```
String querySql = "SELECT * FROM fruits WHERE id = ?";  
try (PreparedStatement preparedStatement = connection.prepareStatement(querySql)) {  
    preparedStatement.setLong(1, 10L);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    resultSet.next();  
    System.out.println(resultSet.getInt("id"));  
    System.out.println(resultSet.getString("name"));  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

Statement vs PreparedStatement

- Il est possible d'utiliser la classe **Statement** ou **PreparedStatement** en JDBC. Voici les différences :

Statement	PreparedStatement
Aucun paramètre	Requête avec paramètres
Compilée à chaque appel	Précompilée
	Protège contre les injections SQL

1. Utilisez **PreparedStatement** pour les requêtes avec paramètres
2. Si un **Statement** est exécuté plusieurs fois, utilisez un **PreparedStatement** pour réduire le temps d'exécution

Fonctionnement du ResultSet

- Représente une abstraction d'une table qui se compose de plusieurs enregistrements constitués de colonnes qui contiennent les données

Méthodes	Rôles
getInt(String)	retourne sous forme d'entier le contenu de la colonne dont le nom est passé en paramètre.
getFloat(String)	retourne sous forme d'un nombre flottant le contenu de la colonne dont le nom est passé en paramètre.
getDate(String)	retourne sous forme de date le contenu de la colonne dont le nom est passé en paramètre.
next()	se déplace sur le prochain enregistrement : retourne false si la fin est atteinte
close()	ferme le ResultSet
getMetaData()	retourne un objet de type ResultSetMetaData associé au ResultSet.

Fonctionnement du ResultSetMetaData

- La méthode `getMetaData()` d'un objet `ResultSet` retourne un objet de type `ResultSetMetaData`. Cet objet permet de connaître le nombre, le nom et le type des colonnes

Méthodes	Rôles
<code>int getColumnCount()</code>	Retourner le nombre de colonnes du <code>ResultSet</code>
<code>String getColumnName(int)</code>	Retourner le nom de la colonne dont le numéro est donné
<code>String getColumnLabel(int)</code>	Retourner le libellé de la colonne donnée
<code>boolean isCurrency(int)</code>	Retourner <code>true</code> si la colonne contient un nombre au format monétaire
<code>boolean isReadOnly(int)</code>	Retourner <code>true</code> si la colonne est en lecture seule
<code>boolean isAutoIncrement(int)</code>	Retourner <code>true</code> si la colonne est auto incrémentée
<code>int getColumnType(int)</code>	Retourner le type de données SQL de la colonne

Fonctionnement du DatabaseMetaData

- DatabaseMetaData permet d'obtenir des informations sur la base de données dans son ensemble : nom des tables, nom des colonnes dans une table, méthodes SQL supportées

Méthodes	Rôles
ResultSet getCatalogs()	Retourner la liste du catalogue d'informations
ResultSet getTables(catalog, schema, tableNames, columnNames)	Retourner une description de toutes les tables
ResultSet getColumns(catalog, schema, tableNames, columnNames)	Retourner une description de toutes les colonnes
String getURL()	Retourner l'URL de la base à laquelle on est connecté
String getDriverName()	Retourner le nom du driver utilisé

Fonctionnement des transactions (1/2)

- Une transaction est gérée à partir de l'objet *Connection*.
- Par défaut, une connexion est en mode *auto-commit* : Dans ce mode, chaque opération est validée unitairement, chacune dans sa propre transaction
- Une transaction permet de valider un ensemble de traitements sur la base de données uniquement s'ils se sont tous effectués correctement

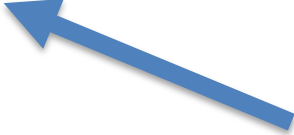
Fonctionnement des transactions (2/2)

- Pour pouvoir rassembler plusieurs traitements dans une transaction, il faut tout d'abord désactiver le mode *auto-commit*.
- La classe *Connection* possède la méthode *setAutoCommit()* qui attend un booléen qui précise le mode de fonctionnement.

Exemple pour désactiver l'auto-commit

- `connection.setAutoCommit(false);`

```
try {  
    connection = DriverManager.getConnection(url, user, password);  
    connection.setAutoCommit(false);  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```



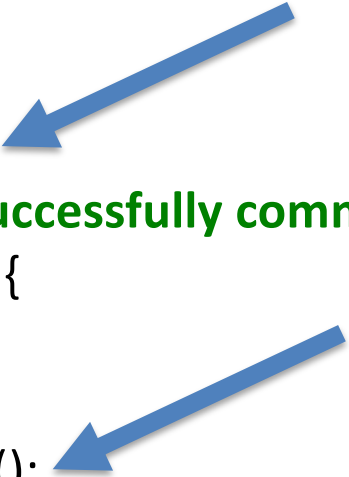
- Toutes nos requêtes DML devront donc être commit (validées et envoyées à la base de données)

Transaction, Commit et Rollback

- Une fois le mode auto-commit désactivé, un appel à la méthode ***commit()*** de la classe *Connection* permet de valider la transaction courante. L'appel à cette méthode valide la transaction courante et crée implicitement une nouvelle transaction
- Si une anomalie intervient durant la transaction, il est possible de faire un retour en arrière pour revenir à la situation de la base de données au début de la transaction en appelant la méthode ***rollback()*** de la classe *Connection*.

Exemple de transaction

```
try {  
    // Do SQL updates...  
    // Commit updates  
    connection.commit();  
    System.out.println("Successfully committed changes to the database !");  
} catch (SQLException e) {  
    try {  
        // Rollback update  
        connection.rollback();  
        System.out.println("Successfully rolled back changes from the database !");  
    } catch (SQLException e1) {  
        System.out.println("Could not rollback updates " + e1.getMessage());  
    }  
}
```



Récupérer la colonne id auto générée

- **getGeneratedKeys()** est la méthode à utiliser si vous devez extraire des clés **AUTO_INCREMENT** via **JDBC**

```
Statement stmt = connection.createStatement();
stmt.executeUpdate(insertQuery, Statement.RETURN_GENERATED_KEYS);
// Using Statement.getGeneratedKeys() to retrieve the value of an auto-increment value
int autoIncrKey = -1;
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next()) {
    autoIncrKey = rs.getInt(1);
} else {
    // throw an exception from here
}
```

Fermeture des ressources

- L'accès aux données doit toujours être fermé une fois effectué
- La fermeture de l'Objet Connection libère la connexion et la replace dans le pool

```
try {  
    // Do SQL queries...  
} catch (SQLException e) {  
    // Rollback  
} finally {  
    resultSet.close();  
    statement.close();  
    connection.close();  
}
```

Pool de connexion

- La création systématique de nouvelles instances de *Connection* est trop lourd

Solution : Le pool de connexion et sa source de données

- Mécanisme permettant de réutiliser les connexions créées
- L'appel à la méthode *close()* de l'Objet Connection ne ferme pas la connexion
 - Elle est « retournée » au pool et peut être utilisée ultérieurement
- La gestion du pool se fait en général de manière transparente pour l'utilisateur

<https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-usagenotes-j2ee-concepts-connection-pooling.html>

<https://www.baeldung.com/java-connection-pooling>

- FIN -