JDBC

Java DataBase Connectivity

Programme détaillé ou sommaire

API JDBC et implémentations
Pilote JDBC
Gestion des pilotes
Chargement d'un pilote
Se connecter à la base
Dialoguer avec la base
INSERT/UPDATE/DELETE
SELECT
Requêtes préparées
Transactions

Objectifs pédagogiques

À l’issue de cette formation, vous serez en mesure de :

- ✓ D'échanger des données entre une application et une base de données relationelle

Chapitre 1
API JDBC

Rappels

1. Gestion des exceptions et le bloc finally

2. Les interfaces

API JDBC ?

Java DataBase Connectivity

API JDBC ?

API Java permettant de communiquer avec une base de données relationnelle :

- contient essentiellement des interfaces
- indépendante du type de base de données utilisée (MySQL, Oracle, Postgres, ...)

Packages → java.sql, javax.sql

API JDBC ?

Exemple

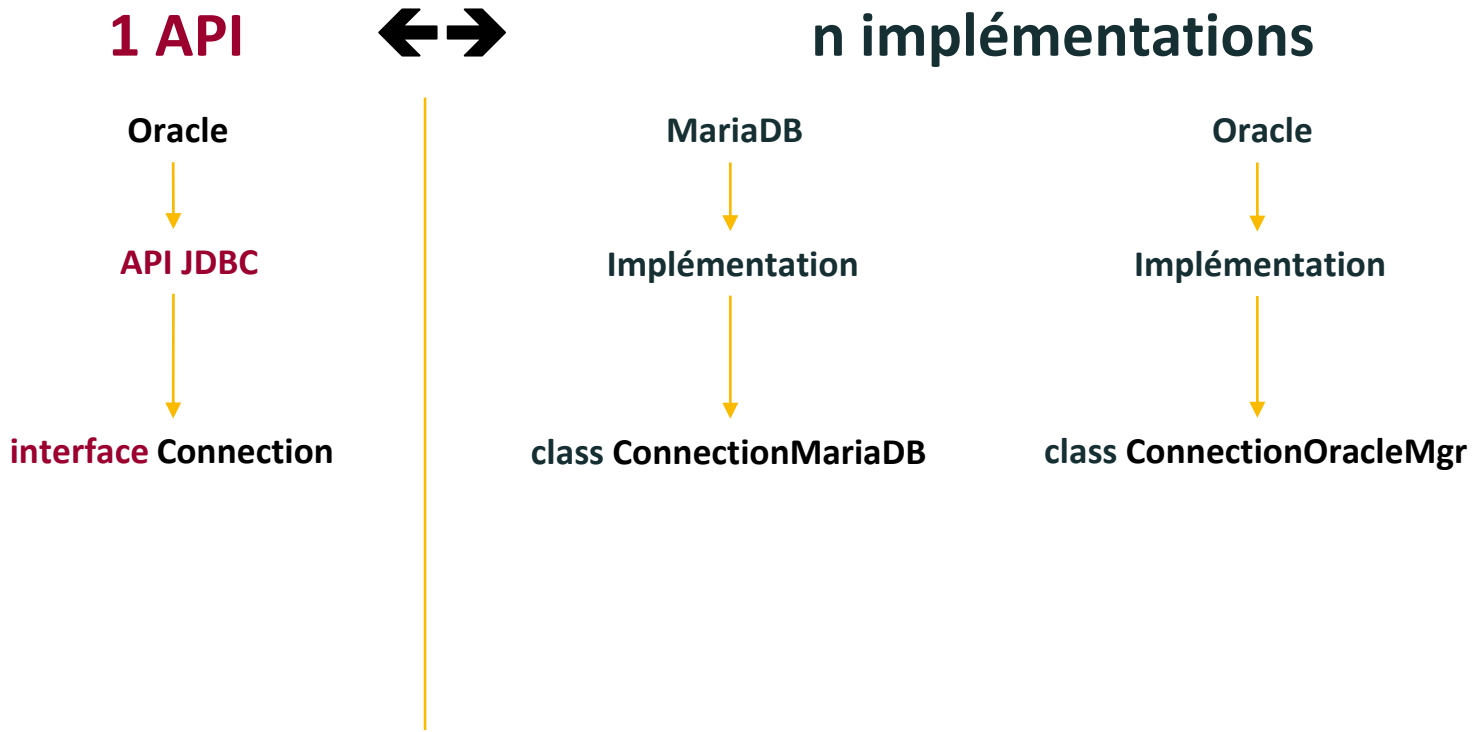
```
DriverManager.registerDriver(new org.mariadb.jdbc.Driver());

Connection connection = DriverManager.getConnection("....", "root", "");
```

Connection est une **interface** : utilisation du polymorphisme !!!

La classe concrète est en réalité une instance de **ConnectionMariaDB**, mais **on ne souhaite pas utiliser les classes spécifiques de l'éditeur**.

API JDBC vs Implémentations

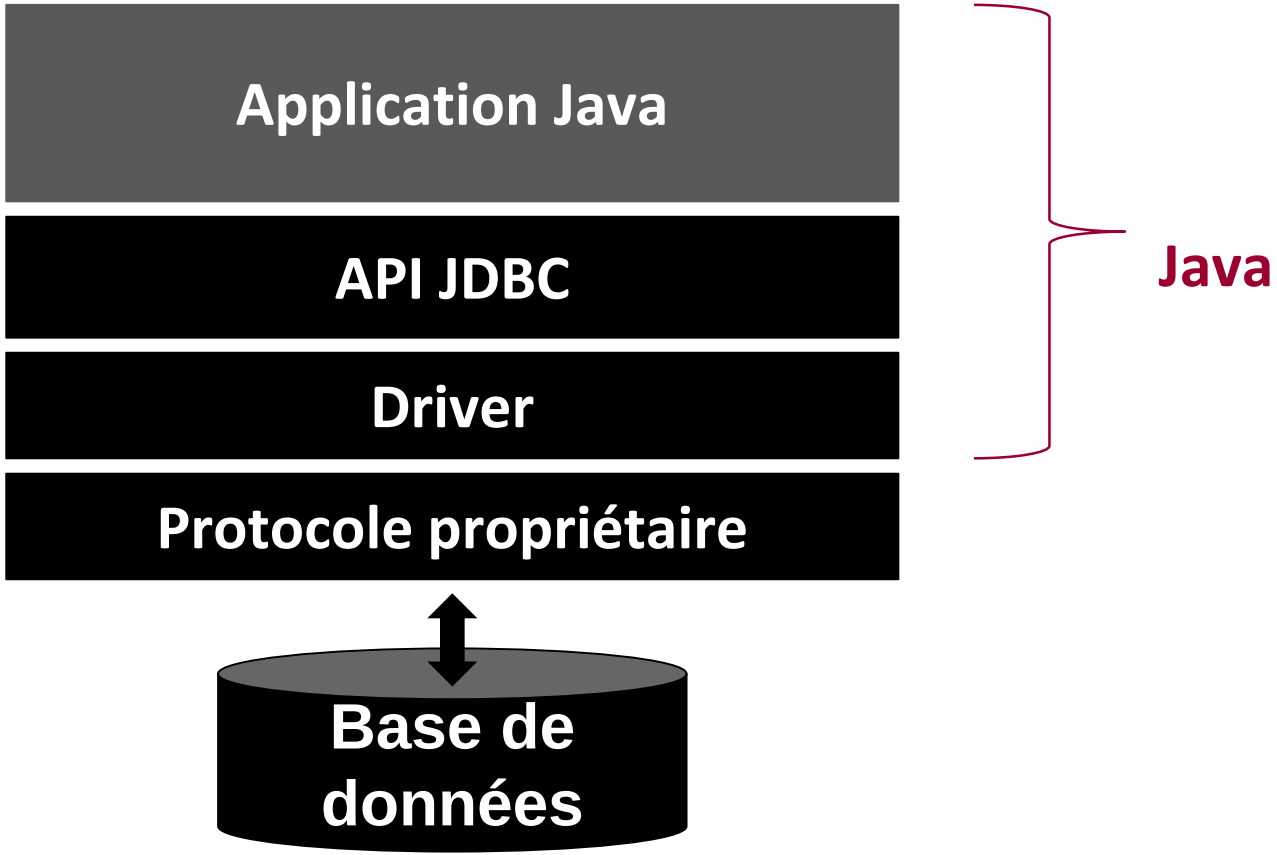


Implémentations JDBC

- L' **API JDBC** ne sert que pour rendre votre **code indépendant** de la base de données utilisées. Elle ne permet pas de se connecter à la base.

- Il faut utiliser une librairie de l'éditeur
  - qui implémente cette API
  - Qui fournit le **driver** de la base de données.
  - Permet de se connecter à la base
  - Permet d'échanger des données avec la base

JDBC ?



Driver (pilote) JDBC

- Un pilote JDBC est un composant logiciel qui permet à une application Java de communiquer avec une base de données relationnelle.

- Un pilote JDBC est une classe Java qui implémente l'interface **java.sql.Driver**
  - Pour MySQL : com.mysql.jdbc.Driver
  - Pour MariaDB : org.mariadb.jdbc.Driver

java.sql.DriverManager

- Une application Java peut se connecter à plusieurs base de données.
- Prend en charge le chargement **des pilotes** et permet de créer de nouvelles connexions à des bases de données.
- Tient à jour la liste des pilotes JDBC.

Charger un pilote

- Se fait par la méthode :

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

- Usuellement :

```
Class.forName("com.mysql.jdbc.Driver");
```

Class.forName("com.mysql.jdbc.Driver")

- L'instruction **Class.forName(String className)** charge la classe dont le nom est passé en paramètre.

- Or cette classe possède un bloc **static** qui est exécuté au chargement.

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    //
    // Register ourselves with the DriverManager
    //
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
```

Pourquoi utiliser Class.forName

- Permet de supprimer l'import de la classe
- Le code est totalement indépendant de la base de données
- Le nom du pilote, au format String, peut être externalisé dans un **fichier de configuration**

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    //
    // Register ourselves with the DriverManager
    //
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
```

Se connecter à la base de données

- Etape qui suit le chargement du pilote
- On créé une connexion à la base de données
  - méthode DriverManager.*getConnection*
- Exemple: ici url, user et password sont des chaines de caractères

Connection **maConnection** = DriverManager.*getConnection*(**url, user, password**);

URL d'accès à une base de données

- **Format : [PROTOCOLE]:[SOUS PROTOCOLE]:[COMPLEMENTS]**
  - **jdbc** : le protocole
  - **[SOUS PROTOCOLE]** : dépend de la base de données et permet de sélectionner le pilote.
  - [COMPLEMENTS] : information de connexion à la base de données avec adresse IP, port et nom de la base de données

- **Exemples d'URL**

jdbc:mysql://localhost:3306/pizzadb

jdbc:postgres://localhost:8090/pizzadb

Atelier (TP)

OBJECTIFS : Créer une connexion à une base de données

DESCRIPTION :

- Dans le TP n°1 vous allez vous connecter à une base de données locale.

Chapitre 2
Fichier de configuration

Pourquoi un fichier de configuration

- On ne met pas en dur dans le code les informations d'accès à la base de données.

- **Comment ferez vous lorsque vous passerez votre application en production ?**

Avantage du fichier de configuration

- L'équipe qui met en production votre application à juste à modifier ce fichier pour renseigner le **user** et **mot de passe de la base de production**.

Qu'utilise t'on en Java ?

- Le fichier de configuration **le plus simple** à utiliser en Java est le fichier **properties**
  - Stockage sous la forme **clé** = **valeur**.

- Un fichier properties est un fichier :
  - de type texte
  - avec une extension **.properties** au lieu de .txt

- On le stocke dans les ressources du projet : **src/main/resources**

Exemple de fichier properties

- Exemple de contenu d'un fichier

Contenu de **database.properties**

```
database.driver=org.mariadb.jdbc.Driver
database.url=jdbc:mariadb://localhost:3306/pizzeria
database.user=user127
database.password=wsgskj74&!
```

Extraire une valeur dans un fichier properties ?

- Utilisation de la classe java.util.ResourceBundle

- On récupère le fichier properties :
  - en passant son **nom** en paramètre de la méthode static getBundle.
  - on ne précise pas l'extension .properties car elle est implicite.

- On récupère une valeur avec la méthode getString(key):

```
ResourceBundle monFichierConf = ResourceBundle.getBundle("database");
String driverName = monFichierConf.getString("database.driver");
```

Atelier (TP)

OBJECTIFS : Créer un fichier de configuration pour l'accès aux données

DESCRIPTION :

- Dans le TP n°2 vous allez mettre en place un fichier de configuration pour accéder à votre base et également créez une base distante chez un founisseur de service d'infrastructure CLOUD.

Chapitre 3
Persistence des données

La persistence des données

- La persistence désigne tout ce qui est en rapport avec les échanges avec une base de données.

- On distingue 4 opérations distinctes:
  - L'insertion (CREATE)
  - La lecture (READ)
  - La mise à jour (UPDATE)
  - La suppression (DELETE)

- Ces opérations sont souvent associées à l'acronyme CRUD

Echanger des données avec la BDD

- Les échanges ne se font pas directement sur l'objet de type Connection. On utilise un **Statement**.

- Statement **monStatement** = **maConnexion**.createStatement();

Type d'instruction SQL	Méthode associée	Type retour	Valeur retour
SELECT	executeQuery	ResultSet	Lignes de résultat
INSERT, UPDATE, DELETE	executeUpdate	Int	Nombre de lignes
CREATE TABLE, etc..	execute	Boolean	true si OK

INSERT

- Méthode **executeUpdate** de la classe **Statement**
- Cette méthode retourne le nombre de lignes insérées

```
Statement monStatement = maConnexion.createStatement();
int nb = monStatement.executeUpdate("INSERT INTO CLIENT (ID,NOM,PRENOM) VALUES (1,'Untel','Bob')");
```

UPDATE

- Méthode **executeUpdate** de la classe **Statement**
- Cette méthode retourne le nombre de lignes **modifiées**

```
Statement monStatement = maConnexion.createStatement();
int nb = monStatement.executeUpdate("UPDATE COMPTE SET SOLDE=SOLDE+1000.0 WHERE PRENOM='Robert'");
```

DELETE

- Méthode **executeUpdate** de la classe **Statement**
- Cette méthode retourne le nombre de lignes **supprimées**

```
Statement monStatement = maConnexion.createStatement();
int nb = monStatement.executeUpdate("DELETE FROM COMPTE WHERE PRENOM='Robert'");
```

SELECT

- Méthode **executeQuery** de la classe **Statement**
- Cette méthode retourne une instance de type **ResultSet**
- Un **ResultSet** est un **curseur** qui permet de parcourir les données

```
Statement monStatement = maConnexion.createStatement();
ResultSet curseur = monStatement.executeQuery("SELECT * FROM CLIENT");
```

ResultSet

- Pourquoi un curseur ?
  - La base de données ne retourne pas tous les résultats d'un coup.
  - Imaginez ce qui se passerait si vous vouliez parcourir une table possédant plusieurs centaines de millions de lignes

ResultSet

- Le **ResultSet** possède :
  - La méthode **next()** qui permet d'avancer ligne par ligne dans les résultats
    - La méthode retourne **true** si elle a réussi à avancer d'une ligne
    - La méthode retourne **false** dans le cas contraire.
  - Possède des **méthodes getInt, getString, etc..** pour récupérer les résultats d'une ligne donnée.

ResultSet

- Zoom sur le fonctionnement du **ResultSet**

ResultSet **curseur** = statement.executeQuery(**"SELECT * FROM CLIENT"**);

Côté Java

Par défaut le ResultSet ne pointe pas sur la première ligne de résultats.

Côté base de données

ID	NOM	PRENOM
127	Untel	Bob
348	Doe	John
19	Durand	Marcel

ResultSet

- Première itération

Côté Java

Premier curseur.next():

- La base de données déplace le curseur sur la ligne 1 et retourne **true** pour indiquer qu'on peut récupérer des résultats.

On peut alors récupérer les résultats :

```
Integer id = curseur.getInteger("ID");          // 127
String nom = curseur.getString("Nom");          // Untel
```

Côté base de données

ID	NOM	PRENOM
127	Untel	Bob
348	Doe	John
19	Durand	Marcel

ResultSet

- Seconde itération

Côté Java

Second curseur.next():

- La base de données déplace le curseur sur la ligne 2 et retourne **true** pour indiquer qu'on peut récupérer des résultats.

On peut alors récupérer les résultats :

Integer id = **curseur**.getInteger("ID"); // 348

String nom = **curseur**.getString("Nom"); // Doe

Côté base de données

ID	NOM	PRENOM
127	Untel	Bob
348	Doe	John
19	Durand	Marcel

ResultSet

- Troisième itération

Côté Java

Troisième curseur.next():

- La base de données déplace le curseur sur la ligne 3 et retourne **true** pour indiquer qu'on peut récupérer des résultats.

On peut alors récupérer les résultats :

```
Integer id = curseur.getInteger("ID");          // 19
String nom = curseur.getString("Nom");          // Durand
```

Côté base de données

ID	NOM	PRENOM
127	Untel	Bob
348	Doe	John
19	Durand	Marcel

ResultSet

- Dernière itération

Côté Java

Dernier curseur.next():

- la méthode retourne **false** pour indiquer que la base n'a plus de données disponibles.

On ne peut récupérer de résultats.

Les méthodes getInteger(..) et getString(…) retourne une **SQLException**

Côté base de données

ID	NOM	PRENOM
127	Untel	Bob
348	Doe	John
19	Durand	Marcel

→ Parcours terminé.

Exemple d'utilisation d'un ResultSet

- Exemple complet

```
ResultSet curseur = monStatement.executeQuery("SELECT ID, NOM, PRENOM FROM CLIENT");


ArrayList<Client> clients = new ArrayList<>();


while (curseur.next()) {
    Integer id = curseur.getInt("ID");
    String nom = curseur.getString("NOM");
    String prenom = curseur.getString("PRENOM");

    Client clientCourant = new Client(id, nom, prenom);
    clients.add(clientCourant);
}
```

Précautions sur le ResultSet

- Un ResultSet doit être fermé après usage
  - Sinon la base de données ne ferme pas le curseur de son côté
  - Au bout d'un certain nombres de curseurs non fermés, la base peut refuser d'ouvrir de nouveaux curseurs.
  - Elle génère alors une SQL Exception avec le message: **maximum open cursors exceeded**

- La fermeture d'un ResultSet se fait généralement dans le bloc finally
- Exemple pour fermer un ResultSet

curseur.close();

Précautions sur le Statement

- Un Statement doit être fermé après usage
  - Sinon la base de données ne ferme pas le statement de son côté
  - Au bout d'un certain nombres de statements non fermés, la base peut refuser d'ouvrir de nouveaux statements.
  - Elle génère alors une SQL Exception avec le message: **too many statements.**

- La fermeture d'un statement se fait généralement dans le bloc finally
- Exemple pour fermer un Statement

monStatement.close();

Précautions sur le Statement

- Une Connection doit être fermée après usage
  - Sinon la base de données ne ferme pas la connexion de son côté
  - Au bout d'un certain nombres de connexions non fermées, la base peut refuser d'ouvrir de nouveaux statements.
  - Elle génère alors une SQL Exception avec le message: **too many connections.**

- La fermeture d'une connexion se fait généralement dans le bloc finally
- Exemple pour fermer une connexion

maConnexion.close();

Gestion des exceptions

- Toutes les méthodes que vous allez utiliser lèvent des exceptions.
- Comment les gérer ?
  - Solution 1: bloc **try/catch** dans la méthode qui accède aux données (DAO)
  - Solution 2: clause **throws SQLException** dans la signature de la méthode pour transférer la gestion de l'exception à la méthode appelante…mais ce n'est pas une très bonne pratique.

Gestion des exceptions

- Exemple solution 1

```
public List<Client> getClients() {

	ArrayList<Client> clients = new ArrayList<>();

	try {
	    // création de la connexion, du statement, du resultSet et traitement des données.
	} catch (SQLException e) {
	    // Traitement de l'exception si elle se produit
	} finally {
	    // Fermeture des ressources: resultSet, statement, connexion.
	}
	return clients;
}
```

Gestion des exceptions

- Exemple solution 2: dans ce cas c'est la méthode appelante qui **catch** l'exception,
- Par contre le **try/finally est obligatoire** pour garantir la fermeture des ressources

```
public List<Client> getClients() throws SQLException {

      ArrayList<Client> clients = new ArrayList<>();

      try {
         // création de la connexion, du statement, du resultSet et traitement des données.
      } finally {
         // Fermeture des ressources: resultSet, statement, connexion.
      }
      return clients;
}
```

Que mettre dans le catch ?

- C'est la question que les développeurs se posent tout le temps.
  - C'est une décision de conception, donc il n'y a pas une bonne réponse unique.
  - Mauvaises pratiques à éviter:
    - Pas de **e.printStackTrace();**
    - Pas de **System.out.println(e.getMessage());**
  - Bonnes pratiques:
    - On **logue** l'erreur pour le journal de l'application
    - On **throw** une exception de type **RuntimeException** si l'exception qui se produit est susceptible de rendre l'application inutilisable.

Chapitre 4
Transactions

Mode auto-commit

- Sur MySQL quand une connexion est créée, elle est par défaut en mode « auto-commit »
- Chaque requête est traitée comme une transaction qui est validée à la fin de la requête.
- Pour désactiver le mode « auto-commit »

```
connection.setAutoCommit(false)
```

Gérer une transaction

- Pour valider une transaction

  connection.commit()

- Pour annuler une transaction (par exemple si une exception est catchée)

  connection.rollback()

Atelier (TP)

OBJECTIFS : Mettre en place des échanges d'informations en JDBC avec une base de données

DESCRIPTION :

- Dans le TP n°3 vous allez créez une table et réaliser les 4 opérations du CRUD.

Chapitre 5
Les DAO

Qu'est ce qu'une DAO ?

- **DAO** est l'acronyme pour **D**ata **A**ccess **O**bject.
- Ce sont des interfaces et classes qui prennent en charge les opérations de CRUD pour un type d'objet donné.
- En général il y a :
  - une **interface** qui définit les méthodes obligatoires
  - une **classe** qui **implémente l'interface**.
  - Il peut y avoir plusieurs classes différentes d'implémentation différentes.

Exemple d'interface

- Exemple d'interface avec ses 4 méthodes de base: lecture, création, mise à jour, suppression

```
public interface FournisseurDao {

    List<Fournisseur> extraire();

    void insert(Fournisseur fournisseur);

    int update(String ancienNom, String nouveauNom);

    boolean delete(Fournisseur fournisseur);
}
```

Exemple de classe d'implémentation

```
public class FournisseurDaoJdbc implements FournisseurDao {

  public List<Fournisseur> extraire() {
       // TODO accéder ici à la base de données et créez la liste de
       // fournisseurs
       return null;
  }
  public void insert(Fournisseur fournisseur) {
       // TODO Accédez ici à la base de données et insérez le fournisseur
       // en base de données
  }
  public int update(String ancienNom, String nouveauNom) {
       // TODO Accédez ici à la base de données et mettre à jour le nom du fournisseur
       // dont l'ancien nom est passé en paramètres
       // TODO le chiffre retourné en fin de méthodes indique le nb de lignes mis à jour.
       return 0;
  }
  public boolean delete(Fournisseur fournisseur) {
       // TODO Accédez ici à la base de données et supprime le fournisseur passé en
       // paramètres
       // TODO le boolean retourné en fin de méthode indique si oui ou non la suppression
       // a bien eu lieu.
       return false;
  }
}
```

Enrichissement des DAO

- Les DAO peuvent accueillir de nouvelles méthodes pour couvrir les besoins applicatifs.
- Exemple de 2 nouvelles méthodes:

```
public interface FournisseurDao {

  …

  List<Fournisseur> select(String chaine);

  …

  int deleteAll(String chaine);

}
```

Atelier (TP)

OBJECTIFS : Utilisez une DAO pour échanger des informations avec la base de données

DESCRIPTION :

- Dans le TP n°4 vous allez écrire la DAO pour la classe Fournisseur et l'utilisez dans vos 4 classes TestInsertion, TestUpdate, TestSelect et TestDelete.

Chapitre 6
Requêtes préparées

PreparedStatement

- Se créé à partir d'une connexion
- Contient une instruction SQL déjà compilée => gain de performance si plusieurs exécutions
- Peut contenir des paramètres (mis à jour avant l'exécution de la requête)

Exemple d’utilisation

- Exemple 1

```
PreparedStatement updatePizza = conn.prepareStatement("UPDATE PIZZA SET PRICE=20.0
WHERE ID=? AND PIZZA_NAME=?");
updatePizza.setInt(1,1);
updatePizza.setString(2, "Regina");
updatePizza.executeUpdate();
```

- Exemple 2

```
PreparedStatement selectPizzaSt = conn.prepareStatement("SELECT * FROM PIZZA WHERE ID=?");
selectPizzaSt.setInt(1,1);
ResultSet resultats = selectPizzaSt.executeQuery();
```

Exemple d’utilisation avec des paramètres nommés

- Exemple 1

```
PreparedStatement updatePizza = conn.prepareStatement("UPDATE PIZZA SET PRICE=20.0 WHERE ID=:id AND PIZZA_NAME=:name");
updatePizza.setInt("id",1);
updatePizza.setString("name", "Regina");
updatePizza.executeUpdate();
```

- Exemple 2

```
PreparedStatement selectPizzaSt = conn.prepareStatement("SELECT * FROM PIZZA WHERE ID=:id");
selectPizzaSt.setInt("id",1);
ResultSet resultats = selectPizzaSt.executeQuery();
```

Atelier (TP)

OBJECTIFS : Mettre en place des requêtes préparées pour mettre en base de données de gros fichiers.

DESCRIPTION :

- Dans le TP n°5 vous allez migrer vos DAO afin qu'elles utilisent des **PreparedStatement**.