

TCH055 — COURS 10

ACCÈS PROGRAMMATIQUE À UNE BASE DE DONNÉES

Pamella Kissok
Chargée de cours
École de Technologie Supérieure



PLAN DE LA SÉANCE

- Présentation de JDBC
- Utilisation de JDBC
 - Établissement et fermeture d'une connexion
 - Requêtes d'interrogation de données
 - Requêtes LMD
 - Requêtes pré-compilées
 - Gestion des exceptions
 - Gestion des transactions



JDBC



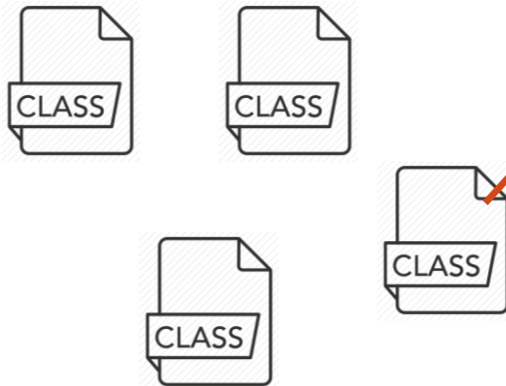
QU'EST-CE QUE JDBC?

- JDBC: Java Data Base Connectivity
- Cadre logiciel (framework) **universel** d'accès à une source de données
- Pas limité aux bases de données relationnelles
- Requiert un “Pilote” JDBC spécifique au SGBD utilisé

POURQUOI JDBC?

- Sans JDBC: utiliser une interface spécifique au SGBD utilisé.

Programme JAVA



Exemple fictif!

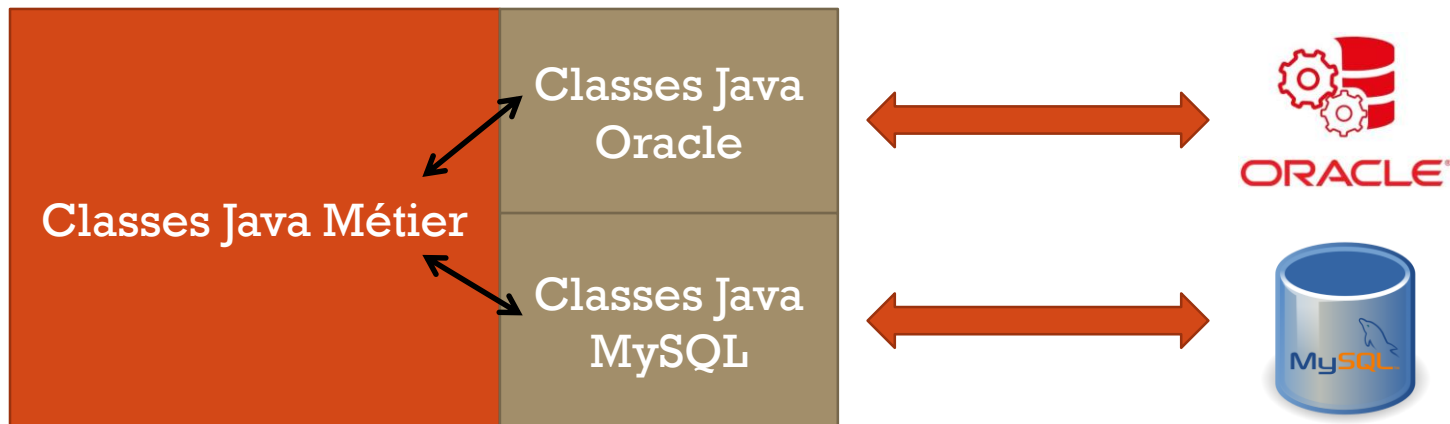
```
import com.oracle.*;  
...  
...  
Oracle or = new Oracle('localhost', 'user1', 'pass');  
or.requete("SELECT * FROM Produits");  
for(resultat: or.getResults()){  
    System.out.println(resultat.get(1));  
}  
...
```

Code dépend du SGBD (ici Oracle)

- ➡ Programme peu évolutif (difficile de passer sur un autre SGBD)
- ➡ Dupliquer le code si on souhaite supporter plus d'une source de données

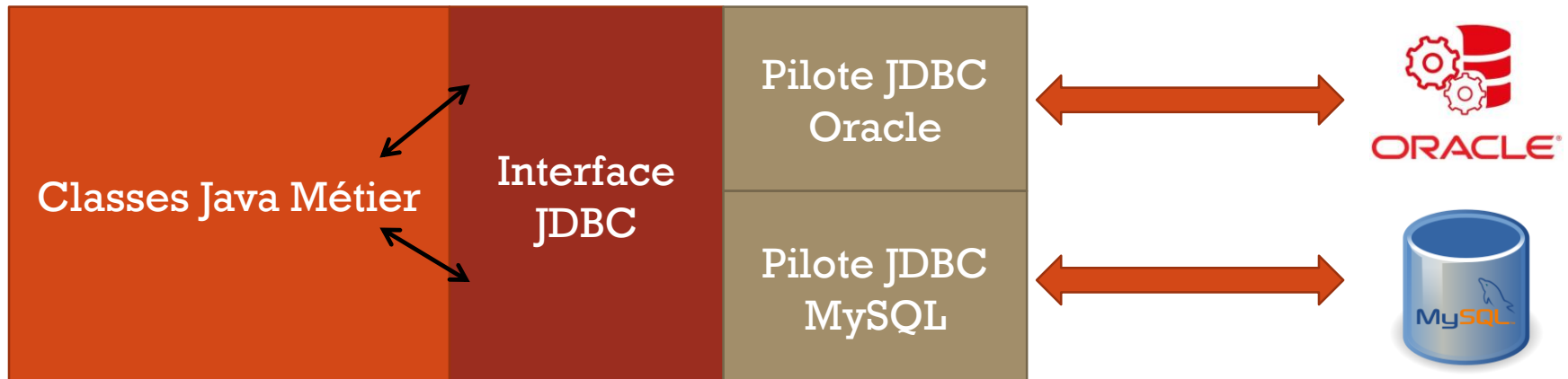
SANS JDBC...

- Sans JDBC: utiliser une interface spécifique au SGBD utilisé.



AVEC JDBC...

- Avec JDBC: utiliser une interface commune, qui communique avec des interfaces spécifiques.
- Notre programme dépend uniquement d'une interface universelle, peu importe la source de données.



UTILISATION DE JDBC

DRIVER JDBC

- Tel qu'évoqué plus haut, nous devons disposer du driver JDBC spécifique au SGBD que nous utilisons.
- Le driver doit se trouver dans le CLASSPATH.
- Chargement du driver en vue d'une connexion:

```
try{  
  
    //Chargement du driver JDBC  
    Class.forName("oracle.jdbc.driver.OracleDriver");  
  
} catch (ClassNotFoundException e){  
    e.printStackTrace();  
}
```

ÉTABLIR UNE CONNEXION

- Nous créons une instance de `java.sql.Connection`
- Pour créer l'instance, on utilise la « fabrique » `DriverManager` (méthode `getConnection`)
- La méthode `getConnection` requiert trois paramètres
 - l'**URI** de connection JDBC
 - Le **nom d'utilisateur** et le **mot de passe**
- **URI de connection JDBC pour oracle (version *thin*)**

`jdbc:oracle:thin:@url_serveur:num_port:sid`

Exemple:

`jdbc:oracle:thin:@192.168.10.1:1521:XE`

ÉTABLIR UNE CONNEXION - EXEMPLE

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

```
...
```

```
Connection connJdbc;
```

```
try{
```

```
    //Chargement du driver JDBC
```

```
    Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
    //Création de la connexion
```

```
    connJdbc = DriverManager.getConnection(  
        "jdbc:oracle:thin:@tch054oral2c.logti.etsmtl.ca:1521:TCH054",  
        "equipe4000", "leMotd3Pass3");
```

```
    } catch (ClassNotFoundException | SQLException e){  
        e.printStackTrace();
```

```
    }
```

```
...
```

Compte personnel:

jdbc:oracle:thin:@localhost:1521:xe

CRÉER UNE REQUÊTE

- Pour créer une requête, nous avons besoin d'une instance de la classe `java.sql.Statement` (ou une sous-classe)
- L'objet **Connection** permet de créer des instances de **Statement** grâce à la méthode **createStatement**.

```
#import java.sql.*;
...
try{
    ...
    Statement requete = connJdbc.createStatement();

} catch ( SQLException e ){
    e.printStackTrace();
}
...
```

EXÉCUTER UNE REQUÊTE D'INTERROGATION DE DONNÉES (1/2)

- On définit et on exécute la requête avec la méthode `executeQuery`.
- Retourne un objet de la classe **`java.sql.ResultSet`**

```
#import java.sql.*;
...
try{
    ...
    Statement requete = connJdbc.createStatement();

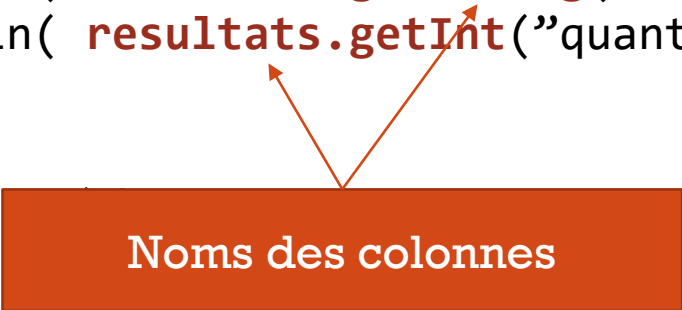
    ResultSet resultats = requete.executeQuery(
        " SELECT * From Sortie WHERE code_produit='R47'" );
    ...
} catch ( SQLException e ){
    e.printStackTrace();
}
...
```

EXÉCUTER UNE REQUÊTE D'INTERROGATION DE DONNÉES (2/2)

- On parcourt les résultats ligne par ligne (comme un curseur) avec la méthode `next()`.
- On utilise les méthodes ***getInt***, ***getString***, ***getDouble***, ***getDate***, ... pour obtenir la valeur d'une colonne.

```
#import java.sql.*;
...
try{
    ...
    ResultSet resultats = requete.executeQuery(
        " SELECT * From Sortie WHERE code_produit='R47'" );
    while(resultats.next()){
        System.out.println( resultats.getString("code_produit");
        System.out.println( resultats.getInt("quantite");
    }

} catch ( SQLException
    e.printStackTrace
}...
```



Noms des colonnes

EXÉCUTER UNE REQUÊTE DE MODIFICATION DE DONNÉES (LMD)


- On définit et on exécute la requête avec la méthode `executeUpdate`.
- Retourne le nombre de lignes affectées (insérées, MAJ ou supprimées)

```
#import java.sql.*;
...
try{
    ...
    int nbAffectees = requete.executeUpdate(
        "DELETE FROM Sortie WHERE code_produit='R47'" );
    if(nbAffectees==0){
        System.out.println("Aucune ligne supprimée!");
    }
    ...
} catch ( SQLException e ){
    e.printStackTrace();
}
...
```

REQUÊTES PRÉCOMPILÉES (1/3)

- Une requête précompilée est une requête générique paramétrée. Pour exécuter la requête, on doit spécifier la valeur des paramètres.
- **Étape 1:** On instancie la classe PreparedStatement.

```
#import java.sql.*;
...
try{
    ...
    PreparedStatement requete = connJdbc.prepareStatement(
        "Select * FROM Produit WHERE classe = ? AND quantite = ?");
    ...
} catch ( SQLException e ){
    e.printStackTrace();
}
...
```



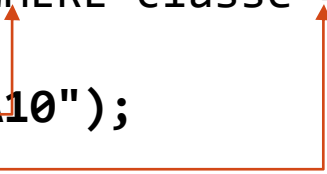
Paramètres de la requête

REQUÊTES PRÉCOMPILÉES (2/3)

- **Étape 2:** Avant d'exécuter la requête, on spécifie la valeur des paramètres (méthodes *setInt*, *setDouble*, *setString*, ...)

```
#import java.sql.*;
...
try{
    ...
    PreparedStatement requete = connJdbc.prepareStatement(
        "Select * FROM Produit WHERE classe = ? AND quantite = ?");

    requete.setString( 1, "A10");
    requete.setInt( 2, 25 );
} catch ( SQLException e ){
    e.printStackTrace();
}
...
```



Les paramètres sont numérotés selon l'ordre de leur apparition dans la requête.

REQUÊTES PRÉCOMPILÉES (3/3)

- **Étape 3:** on exécute la requête en utilisant les méthodes ***executeQuery*** ou ***executeUpdate***, selon le type de requête.

```
#import java.sql.*;
...
try{
    ...
    PreparedStatement requete = connJdbc.prepareStatement(
        "Select * FROM Produit WHERE classe = ? AND quantite = ?");

    requete.setString( 1, "A10");
    requete.setInt( 2, 25 );

    ResultSet resultat = requete.executeQuery();
    while(resultat.next()){
        System.out.println("Classe:" + resultat.getString("classe"));
    }

} catch ( SQLException e ){
    e.printStackTrace();
}
...
```

EXÉCUTIONS LMD PAR LOT

- Disponible pour les requêtes précompilées (**PreparedStatement**):

```
#import java.sql.*;

...
try{
    ...
    PreparedStatement requete = connJdbc.prepareStatement(
        "INSERT INTO Produit (classe, quantite) VALUES (?,?)");
    requete.setString( 1, "A10");
    requete.setInt( 2, 25 );
    requete.addBatch();

    requete.setString( 1, "A45");
    requete.setInt( 2, 34 );
    requete.addBatch();

    int[] resultat = requete.executeBatch();

} catch ( SQLException e ){
    e.printStackTrace();
}

...
```

REQUÊTE PRÉCOMPILÉES VS NORMALES

- Une requête précompilée ***PreparedStatement*** est utile uniquement lors de la construction de requêtes dynamiques (une requête qui doit être complétée par des valeurs à l'exécution).
- Il est possible de construire des requêtes dynamiquement en utilisant des ***Statement*** (avec des concaténations), cependant ***PreparedStatement*** :
 - Est plus efficace si la même requête doit être effectuée plus d'une fois dans la méthode;
 - Nous protège face aux attaques par injection SQL en échappant automatiquement les chaînes de caractères lues.
 - Permet des exécutions par lot.

GESTION DES TRANSACTIONS

- En JDBC, nous sommes par défaut en mode « commit automatique »: une requête **commit** est envoyée automatiquement après chaque requête LMD (**executeUpdate**).
- La méthode **setAutoCommit(boolean)** de la classe **Connection** permet de changer le mode:

Compte personnel:
jdbc:oracle:thin:@localhost:1521:xe

```
#import java.sql.*;
...
Connection connJdbc;
try{
    //Création de la connexion
    connJdbc = DriverManager.getConnection(
        "jdbc:oracle:thin:@tch054oral2c.logti.etsmtl.ca:1521:TCH054", "equipe4000", "leMotd3Pass3");

    connJdbc.setAutoCommit(false);

} catch (ClassNotFoundException | SQLException e){
    e.printStackTrace();
}
...
```

GESTION DES TRANSACTIONS

- Si le mode « auto-commit » est désactivé, utiliser les méthodes ***commit()*** et ***rollback()*** de la classe ***Connection*** pour valider ou annuler une transaction.
- Exemple avec commit:

```
...
try{
    ...
    PreparedStatement requete = connJdbc.prepareStatement(
        "INSERT INTO Produit (classe, quantite) VALUES (?,?)");
    requete.setString( 1, "A10");
    requete.setInt( 2, 25 );

    requete.executeUpdate();
    connJdbc.commit();

} catch ( SQLException e ){
    e.printStackTrace();
} ...
```



INTERFACE EN LANGAGE C AVEC OCILIB



INTERFAÇAGE AVEC OCILIB

- Approche similaire à JDBC
- Particularités du C:
 - Pas d'objets mais des (pointeurs vers des) structures ;
 - Pas de gestionnaire d'exception (try/catch) mais un pointeur vers une fonction qui gère les exceptions.

INITIALISATION ET CONNEXION

- Inclure le fichier .h de la librairie:

```
#include "OCILIB.h"
```

- Créer une fonction de gestion d'erreurs:

```
void err_handler(OCI_Error *err)
{
    printf("%s\n", OCI_ErrorGetString(err));
}
```

INITIALISATION ET CONNEXION

- Initialisation de la librairie:

```
if (!OCI_Initialize(err_handler, NULL, OCI_ENV_DEFAULT))  
{  
    return EXIT_FAILURE;  
}
```

- Établir la connexion avec OCI_ConnectionCreate:

```
OCI_Connection *cn;  
  
cn = OCI_ConnectionCreate(  
    "tch054ora12c.logti.etsmtl.ca:1521/tch054ora12c.logti.etsmtl.ca",  
    "nom_utilisateur",  
    "le_mot_de_passe",  
    OCI_SESSION_DEFAULT);
```

INITIALISATION D'UNE STRUCTURE DE REQUÊTE

- Afin d'exécuter une requête, il est nécessaire de créer et d'initialiser une structure OCI_Statement
- On utilise la fonction OCI_StatementCreate(OCI_Connection*)

```
OCI_Statement *st;
```

```
st = OCI_StatementCreate(cn);
```

EXÉCUTION D'UNE REQUÊTE

- **OCI_ExecuteStmt(OCI_Statement*, char*)**

```
OCI_ExecuteStmt(  
    st,  
    "SELECT code_produit, desc_produit, quantite from Produit");
```

- **Récupérer les résultats:**

```
OCI_Resultset *rs;  
rs = OCI_GetResultset(st);
```

- **Parcourir les résultats (fetch)**

```
while( OCI_FetchNext(rs) ) {  
    ...  
}
```

RÉCUPÉRER LES VALEURS

- En parcourant les résultats avec `OCI_FetchNext`, on accède à la valeur d'une colonne de la ligne courante en utilisant les fonctions `OCI_GetXXX`, en remplaçant XXX par le type de donnée attendu.

- Ex.: pour récupérer un entier depuis le `OCI_ResultSet* rs`:

```
printf("Quantite: %i\n", OCI_GetInt(rs, 3));
```

- Autres fonctions Get, selon le type:

`OCI_GetString`, `OCI_GetDouble`, `OCI_GetShort`, `OCI_GetFloat`...

- Pour une liste complète, la documentation à:

http://vrogier.github.io/ocilib/doc/html/group_oci_c_api_fetching.html

LIBÉRER LA MÉMOIRE

- Libérer le OCI_Statement avec OCI_StatementFree:

```
OCI_StatementFree(st);
```

- Déconnection et libération du OCI_Connection avec OCI_ConnectionFree:

```
OCI_ConnectionFree(cn);
```

- Libération de la mémoire résiduelle utilisée par la librairie:

```
OCI_Cleanup();
```