

# Java avancé et GL

## Cours 4 – Java et Bases de données

*Extrait du cours de « Pierre Charbit, Jean-Baptiste Yunès »*

*Master Informatique*

# Sommaire

I.Collections

II.Sérialisation

III.Accès aux données

# I.Collections

# I.Collections

## L'API des Collections

### ◆ ArrayList

- ◆ L'un des inconvénients des tableaux est le fait que l'on doit fixer sa taille à l'initialisation
- ◆ Il existe en Java des classes permettant de gérer des collections dynamiques, comme la classe « **ArrayList** ».
- ◆ On peut les utiliser de la façon suivante :
- ◆ `ArrayList<A> v = new ArrayList<A>();`
- ◆ Où « A » est le type d'objets stockés. Il s'agit nécessairement d'un type d'objet (comme String,...)
- ◆ Mais pas d'un type primitif (comme int, char, ...).
- ◆ Exemple
  - ◆ `ArrayList<String> = new ArrayList<String>()`

# I.Collections

## L'API des Collections - ArrayList

- ◆ On peut utiliser les méthodes :
  - ◆ **boolean add(A element)** // Ajoute un élément en fin de liste
  - ◆ **void add(int indice, A element)** // Ajoute un élément à un indice donné
  - ◆ **void clear()** // vide la structure de tous ces éléments
  - ◆ **void remove(int indice)** // retire l'élément d'indice donné
  - ◆ **void remove(A element)** // retiré l'élément
  - ◆ **A get(int indice )** // retrouve l'élément d'indice donné
  - ◆ **boolean contains(A element)** // teste l'existence d'un élément dans la collection
  - ◆ **int indexOf(T element)** // retrouve l'indice d'un élément dans la collection
  - ◆ **int size()** // retrouve le nombre d'éléments dans la collection

# I.Collections

## L'API des Collections - ArrayList

### ◆ Exemple

```
ArrayList<String> tab = new ArrayList<String>(); // équivalent String [] tab = new String[2];  
tab.add("Bonjour"); // équivalent tab[0] = "Bonjour";  
tab.add("Au revoir"); // équivalent tab[1] = "Au revoir";  
System.out.println(tab.get(0)); // équivalent: System.out.println(tab[0]);
```

### ◆ Boucles «for each»

```
String[] t = {'toto','titi', 'tata'};  
// Ancienne façon  
for(int i=0; i<t.length; i++) {  
    System.out.println(t[i]);  
}  
  
// nouvelle façon de faire  
for(String s : t) { // pour s variant sur tous les éléments de t  
    System.out.println(s);  
}
```

# I.Collections

## L'API des Collections - ArrayList

### ◆ Exemple

```
import java.util.ArrayList;
import java.util.List;

public class TestBoucle {

    public static void main(String[] args) {
        List<Personne> personnes = new ArrayList<>(6);
        personnes.add(new Personne("p1", Genre.HOMME, 176));
        personnes.add(new Personne("p2", Genre.HOMME, 190));
        personnes.add(new Personne("p3", Genre.FEMME, 172));
        personnes.add(new Personne("p4", Genre.FEMME, 162));
        personnes.add(new Personne("p5", Genre.HOMME, 176));
        personnes.add(new Personne("p6", Genre.FEMME, 168));

        long total = 0;
        int nbPers = 0;
        for (Personne personne : personnes) {
            if (personne.getGenre() == Genre.FEMME) {
                nbPers++;
                total += personne.getTaille();
            }
        }
        double resultat = (double) total / nbPers;
        System.out.println("Taille moyenne des femmes = " + resultat);
    }
}
```

## II.Sérialisation



# II.Sérialisation

## Sauvegarde des objets

- ◆ Java possède un mécanisme simple d'emploi pour sauvegarder ses objets dans des fichiers ou des BDs.
- ◆ Ceci afin qu'ils puisse exister encore une fois le programme terminé ou de pouvoir les échanger entre application.
- ◆ Il s'agit de la sérialisation. Cela peut s'avérer très pratique pour conserver des données à la fermeture de l'application afin de les recharger au lancement suivant.
- ◆ Pour pouvoir sauver un objet, il faut que leur classe implémente l'interface « Serializable »
- ◆ Cette interface ne requiert la redéfinition d'aucune méthode particulière, on doit juste préciser que l'on donne la possibilité à cette classe d'être sérialisée.

# II.Sérialisation

## Sauvegarde des objets - Exemple

```
package com.sdzee.tp.beans;

import java.io.Serializable;

public class Client implements Serializable {

    private Long id;
    private String nom;
    private String prenom;
    private String adresse;
    private String telephone;
    private String email;
    private String image;

    public void setId( Long id ) {
        this.id = id;
    }

    public Long getId() {
        return id;
    }

    public void setNom( String nom ) {
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }

    public void setPrenom( String prenom ) {
        this.prenom = prenom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setAdresse( String adresse ) {
        this.adresse = adresse;
    }
}
```

## II.Sérialisation

### Sauvegarde des objets – Exemple(Suite)

```
    }  
  
    public void setTelephone( String telephone ) {  
        this.telephone = telephone;  
    }  
  
    public String getTelephone() {  
        return telephone;  
    }  
  
    public void setEmail( String email ) {  
        this.email = email;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setImage( String image ) {  
        this.image = image;  
    }  
  
    public String getImage() {  
        return image;  
    }  
}
```

### III. Accès aux données

# III.Accès aux données

## Java et BD

- ◆ Il existe plusieurs bases de données différentes (PostgreSQL, MySQL, SQL server, Oracle, Access....).
- ◆ On choisit ici « MySQL » mais le fonctionnement des interactions avec java est très similaire.
- ◆ Dans tous les cas, java a besoin d'un « Driver » pour communiquer avec la base de donnée, et ce driver est spécifique à la base.
- ◆ Une recherche sur Google permet de télécharger le « driver » en question qui est un « .jar ».
- ◆ Ajouter « .jar » au « projet java » et tester par la suite la connexion à la base de données en donnant le nom de la base, le login et le mot de passe pour accéder à cette base.

# III. Accès aux données

## Se connecter à la base - Exemple

```
public static void main(String[] args) {  
  
    try {  
        String url = "jdbc:mysql://localhost/mabase";  
        String user = "root";  
        String passwd = "";  
  
        Connection conn = DriverManager.getConnection(url, user, passwd);  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

- ◆ Pour pouvoir communiquer il faut créer « la connection » avec notre base. La méthode « getConnection » prend en argument l'url, le nom d'utilisateur et le password.
- ◆ l'url est constituée du protocole (jdbc:mysql), suivi de l'adresse de la machine hébergeant la base (ici, localhost), suivi du nom de la base de données (ici mabase).
- ◆ Les méthodes manipulant des bases de données sont susceptibles de générer des exceptions, c'est pourquoi les instructions sont entourées d'un bloc « try catch ».

# III. Accès aux données

## Lire dans la base : Statement et ResultSet

- ◆ Pour pouvoir exécuter des requêtes SQL, on a besoin d'un objet Statement. On le crée avec l'instruction suivante :

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

- ◆ où « conn » est notre objet de type Connection.
- ◆ Ensuite on peut effectuer une requête. Cela s'effectue via la méthode executeQuery qui prend en argument une chaîne de caractères correspondant à la requête SQL.
- ◆ Supposons que la base contienne la table table1, on pourra donc écrire :

```
ResultSet result = st.executeQuery("SELECT * FROM table1");
```

- ◆ L'objet de type « ResultSet » est celui qui contient le résultat de la requête et se comporte comme une tête de lecture qui est positionnée au début avant la première ligne de la base.
- ◆ Avec la commande « result.next() », on passe à la ligne suivante. Cette commande renvoie « false » si on est à la dernière ligne.

# III.Accès aux données

## Lire dans la base : Statement et ResultSet

- ◆ On peut récupérer les infos de la ligne courante avec des instructions de type get.
- ◆ Si on ne connaît pas le nom des colonnes on utilisera « getObject(int i) ». Cela renvoie l'objet contenu dans « la i-ème » colonne de la ligne courante (que l'on pourra convertir avec la méthode « toString() » si on veut l'afficher).
- ◆ Si la table a deux colonnes on pourra pour afficher la table écrire par exemple (attention les indices commencent à 1 dans mysql) :

```
while(result.next()){  
    System.out.print(result.getObject(1).toString());  
    System.out.print("|");  
    System.out.println(result.getObject(2).toString());  
}
```

- ◆ A la fin il ne faut pas oublier de fermer les objets :

```
result.close();  
st.close();
```



# III.Accès aux données

## Lire dans la base : Statement et ResultSet

- ◆ On peut récupérer les infos de la ligne courante avec des instructions de type get.
- ◆ Si on connaît le nom et le type de données de chaque colonne on pourra utiliser un « get » approprié, comme « getString », ou « getInt », « getDouble... »
- ◆ Si par exemple on a une colonne "nom" et une colonne "prenom", pour afficher la table, on pourra écrire :

```
while(result.next()) {  
    System.out.print(result.getString("nom");  
    System.out.print(" | " );  
    System.out.print(result.getString("prenom");  
}
```

# III. Accès aux données

## Lire dans la base : ResultSetMetaData

- ◆ L'objet « Result » nous permet de lire la requête mais ne permet pas d'obtenir des informations globales, comme le nombre de lignes, le nombre des colonnes ou leur nom.
- ◆ Pour cela on passe par la classe « ResultSetMetaData ».
- ◆ **Exemple**

```
Statement st = conn.createStatement();
ResultSet result = st.executeQuery("SELECT * FROM mabase");
ResultSetMetaData resultMeta = result.getMetaData();
int n = resultMeta.getColumnCount();
for(int i=1 ; i<= n ;i++){
    String s= resultMeta.getColumnName(i);
    System.out.print(s +"\t");
}
```

# III.Accès aux données

## Lire dans la base : ResultSetMetaData - Exemple

```
try{
    String url = "jdbc:mysql://localhost/mabase";
    Connection conn = DriverManager.getConnection(url, "root","");

    Statement st = conn.createStatement();

    ResultSet result = st.executeQuery("SELECT * FROM mabase");
    ResultSetMetaData resultMeta = result.getMetaData();
    int n = resultMeta.getColumnCount();
    for(int i=1 ; i<= n ;i++){
        String s= resultMeta洗getColumnNane(i);
        System.out.print(s +"\t");
    }

    while(result.next()){
        System.out.print(result.getString("nom");
        System.out.print(" | " );
        System.out.print(result.getString("prenom");
    }
    result.close();
    st.close();
}
catch (Exception e) {
    e.printStackTrace();
}
```

# III.Accès aux données

## Écrire dans la base

- ◆ Les requêtes d'écriture se font via l'objet « Statement » déjà utilisé pour la lecture, en utilisant cette fois la méthode « executeUpdate(String s) » qui prend en argument une chaîne de caractères correspondant à l'instruction MySQL de type UPDATE, INSERT, DELETE, CREATE.
- ◆ Si notre table « table1 » avec les deux colonnes « nom » et « age » contient une ligne de nom pierre, alors l'instruction suivante change l'age de pierre :

```
Statement st = conn.createStatement();  
st.executeUpdate("UPDATE table1 SET age=20 WHERE nom LIKE 'pierre' ");
```

- ◆ Il existe aussi des méthodes de la classe « ResultSet » permettant de modifier la table :

```
ResultSet result = st.executeQuery("SELECT * from table1");  
result.next(); // on passe sur la premiere ligne  
result.setString("age",25); // on change la valeur de "age " pour cette ligne.
```

# III.Accès aux données

## Écrire dans la base - Requêtes Préparées

- ◆ La Classe « PreparedStatement » étend la classe Statement. Deux différences notables :
  - ◆ Les requêtes sont pré-compilées en SQL, ce qui implique un gain d'efficacité, surtout si la requête doit être appelée plusieurs fois
  - ◆ On peut faire des requêtes à trous
- ◆ **Exemple :**

```
String req = "SELECT * from matable WHERE nom = ? OR id =?" ;
PreparedStatement pst = conn.prepareStatement(req);

pst.setString(1, "Pierre");
pst.setInt(2, "32");
pst.executeUpdate();

pst.setString(1, "toto");
pst.executeUpdate();
```

# Questions ?