Visons haut et loin
www.e-media.mg
UNIVERSITÉ NUMÉRIQUE
LICENCE
Siège : Lot II Y 33 P Bis Ampasanimalo, Bâtiment Bleu
101 Antananarivo, Madagascar
Tél : +261 34 11 777 37 | +261 34 56 777 07
Mail : emediaenligne2020@gmail.com

Support de cours

JAVA AVANCE

AVANT-PROPOS

Ce support de cours est destiné aux étudiants ayant une certaine maîtrise des bases de la programmation orientée objet et de son application en langage Java. Il est également obligatoire d'avoir acquise une certaine maîtrise de la modélisation d'une base de données avec la méthode Merise, et du langage SQL. Dans ce cours, nous allons d'abord voir les concepts orienté objet avancé de Java, puis nous allons coupler le langage Java avec le SGBD MySQL. Voici quelques conseils pour le bon déroulement de notre cours :

- Lisez régulièrement ce support de cours.
- N'hésitez pas à le relire dans le cas où quelques parties vous semblent floues.
- Observez les exemples dans ce cours. Aussitôt ces exemples assimilés, c'est à vous maintenant de répéter ces exemples.
- Traitez les exercices par vous-mêmes sans regarder le corrigé. Les exercices faits, comparez vos réponses avec le corrigé. Dans le cas où vos réponses étaient fausses, n'hésitez pas à refaire ces exercices. Notez que les exercices servent à assimiler le cours, c'est dans votre intérêt
- Puisqu'il s'agit de nouvelles technologies, des recherches personnelles sont très enrichissants pour votre avancement.

Bon, voilà, je vous souhaite bon courage.

Chapitre 1. CONCEPTS P.O.O. AVANCE

1. Les interfaces

Pour commencer, nous allons parler des classes abstraites.

Une classe abstraite est classe que l'on ne peut pas instancier. Elle sert à représenter un concept abstrait du monde à modéliser (Exemples : un animal, une personne…).

Voici des exemples de code :

Classe abstraite « Personne » :

```
public abstract class Personne {
 protected String nom;
 protected int age;
 public void marcher() {
  System.out.println("Je marche avec mes pieds");
 }
 public abstract void travailler();
 public String getNom() {
  return this.nom;
 }
 public int getAge() {
  return this.age;
 }
 public void setNom(String nom) {
  this.nom = nom;
 }
 public void setAge(int age) {
  this.age = age;
 }
}
```

Classe « Professeur » :

```
public class Professeur extends Personne {
 private String specialite;
 public Professeur(String nom, int age, String specialite)
{
  this.nom = nom;
  this.age = age;
```

```
  this.specialite = specialite;
 }
 public void travailler() {
  System.out.println("J'enseigne");
 }
 public String getSpecialite() {
  return this.specialite;
 }
 public void setSpecialite(String specialite) {
  this.specialite = specialite;
 }
}
```

Classe « Etudiant » :

```
public class Etudiant extends Personne {
 private String parcours;
 public Etudiant(String nom, int age, String parcours) {
  this.nom = nom;
  this.age = age;
  this.parcours = parcours;
 }
 public void travailler() {
  System.out.println("J'étudie");
 }
 public String getParcours() {
  return this.parcours;
 }
 public void setParcours(String parcours) {
  this.parcours = parcours;
 }
}
```

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  Etudiant etu = new Etudiant("Bema", 23, "NTIC");
```

```
      Professeur prof = new Professeur("Randria", 46,
"Télécommunications");
      System.out.println(prof.getNom());
      prof.travailler();
      etu.travailler();
      etu.marcher();
     }
    }
```

Comme nous voyons, on utilise le mot-clé « abstract » pour déclarer une classe ou une méthode abstraite. Une méthode abstraite est une méthode sans corps (sans accolades). Une classe héritant d'une classe abstraite doit « implémenter » les méthodes abstraites de sa classe mère, c'est-à-dire définir le contenu de leur corps. Mais attention, on n'utilise plus le mot-clé « abstract » dans l'implémentation de ces méthodes.

Voyons maintenant les interfaces.

Une interface est une classe totalement abstraite, c'est-à-dire qu'on ne peut pas l'instancier, et qu'elle contient des méthodes toutes abstraites, mais ne contient aucun attribut. Le principal avantage est la possibilité de réaliser l'héritage multiple avec des interfaces, qui était impossible avec des classes simples. Au niveau vocabulaire, lorsqu'il s'agit d'une classe on dit « héritage », mais lorsqu'il s'agit d'interfaces on dit « implémentation ».

Voici des exemples de code :

Interface « Actions »

```
public interface Actions {
 public void marcher();
 public void courir();
 public void rentrer();
}
```

Classe « Professeur » :

```
public class Professeur implements Actions {
 private String nom;
 private String specialite;
 public Professeur(String nom, String specialite) {
  this.nom = nom;
```

```
  this.specialite = specialite;
 }
 public void marcher() {
  System.out.println("Je marche");
 }
 public void courir() {
  System.out.println("Je cours");
 }
 public void rentrer() {
  System.out.println("Je rentre");
 }
 public void travailler() {
  System.out.println("J'enseigne");
 }
 public String getNom() {
  return this.nom;
 }
 public void setNom(String nom) {
  this.nom = nom;
 }
 public String getSpecialite() {
  return this.specialite;
 }
 public void setSpecialite(String specialite) {
  this.specialite = specialite;
 }
}
```

Classe « Etudiant » :

```
public class Etudiant implements Actions {
 private String nom;
 private String parcours;
 public Etudiant(String nom, String parcours) {
  this.nom = nom;
  this.parcours = parcours;
```

```
 }
 public void marcher() {
  System.out.println("Je marche");
 }
 public void courir() {
  System.out.println("Je cours");
 }
 public void rentrer() {
  System.out.println("Je rentre");
 }
 public void travailler() {
  System.out.println("J'étudie");
 }
 public String getNom() {
  return this.nom;
 }
 public void setNom(String nom) {
  this.nom = nom;
 }
 public String getParcours() {
  return this.parcours;
 }
 public void setParcours(String parcours) {
  this.parcours = parcours;
 }
}
```

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  Etudiant etu = new Etudiant("Bema", "NTIC");
  etu.travailler();
  etu.courir();
 }
}
```

Comme nous voyons, une interface se déclare à l'aide au mot-clé « interface » au lieu de « class ». On doit aussi impérativement redéfinir le corps des méthodes de l'interface implémentée. Bien évidemment, on a pu optimiser le code en ajoutant la classe « Personne », mais le dernier exemple de codes est plus pédagogique.

2. Les exceptions

Une exception est une erreur qui s'est produite durant l'exécution d'un programme, et qui produit l'arrêt immédiat de ce programme. Il est possible en Java de gérer les exceptions grâce au bloc « try… catch() ». Voici un exemple de code :

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  int a = sc.nextInt();
  int b = sc.nextInt();
  try {
   int c = a/b;
  } catch(ArithmeticException e) {
   System.out.println("Division par zéro interdite");
   e.printStackTrace();
  } finally {
   System.out.println("On continue…");
  }
 }
}
```

On met dans le bloc « try » (qui signifie « essayer ») tout portion de code qui est susceptible de produire une exception. En effet, dans notre exemple, l'utilisateur peut saisir le nombre `0` pour la valeur de `b`, et donc l'opération `c = a/b` peut produire une division par `0`, qui engendre une exception. Ensuite, nous voyons le bloc « catch » (qui signifie « attraper ») dont le paramètre est le type de l'exception à gérer (ici, on a géré l'exception de type « `ArithmeticException` », et on l'a mise dans un objet qu'on a nommé « `e` »), et le corps est le bloc d'instructions à effectuer en cas de cette exception. L'objet e contient une méthode `printStackTrace()` qui permet d'afficher le détail de l'exception qui s'est produite. Enfin, on met dans le bloc optionnel « finally » les instructions à effectuer quoi qu'il en soit, c'est-à-dire si une exception s'est produite ou non.

Il est aussi possible de créer nos propres exceptions. Il suffit de créer une classe qui hérite de la classe « Exception », par exemple :

Classe « AgeException » :

```
public class AgeException extends Exception {
 public AgeException(String message) {
  super(message);
 }
}
```

Classe « Prof » :

```
public class Prof {
 private String nom;
 private int age;
 public Prof(String nom, int age) throws AgeException {
  if(this.age >= 60) {
   throw new AgeException("Professeur retraité");
  }
  else {
   this.nom = nom;
   this.age = age;
  }
 }
 public String getNom() {
  return this.nom;
 }
 public int getAge() {
  return this.age;
 }
 public void setNom(String nom) {
  this.nom = nom;
 }
 public void setAge(int age) {
  this.age = age;
 }
}
```

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  try {
   Prof p = new Prof("Benja", 65);
  } catch(AgeException e) {
   System.out.println(e.getMessage());
  }
 }
}
```

Nous avons d'abord créé la classe « AgeException » qui hérite de la classe « Exception », et dont le constructeur n'est autre que celui de sa classe mère. Nous avons ensuite créé une classe « Prof », dont nous souhaitons que l'attribut « age » ne soit pas supérieur ou égal à 60. Ainsi, pour signaler qu'une méthode peut générer une exception, on utilise le mot-clé « throws » (Attention, avec un « s ») suivi du nom de la classe gérant l'exception (c'est le cas de notre constructeur). Ensuite, on utilise le mot-clé « throw » (Attention, sans « s ») suivi d'une instanciation de l'exception pour lever cette exception. En effet, dans notre exemple, si l'âge est supérieur ou égal à 60, on lève l'exception « AgeException ». Enfin, dans notre classe de test, le constructeur de la classe Prof est donc maintenant susceptible de produire une exception de type AgeException, c'est pourquoi on entoure l'instanciation de la classe Prof de blocs try… catch.

3. Les collections

La plateforme Java permet de stocker des collections d'objets (on parle aussi de structures de données). Il existe plusieurs types de collections en Java, mais nous allons ici n'en présenter que quelques exemples.

La première catégorie que nous allons voir est ce qu'on appelle les « listes ». Il existe plusieurs types de listes, mais la plus utilisée est la structure gérée par la classe « ArrayList ». Une liste se comporte comme un tableau, mais extensible à volonté (théoriquement sans limite de taille). Voici un exemple de code :

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  ArrayList liste = new ArrayList();
  liste.add("Bonjour");
```

```
    liste.add(15);
    liste.add(3.14);
    for(int i = 0 ; i < liste.size() ; i++) {
     System.out.println(liste.get(i));
    }
   }
}
```

Comme nous voyons dans cet exemple, il suffit d'instancier la classe « ArrayList » pour créer ce type de liste. Ensuite, on utilise la méthode add(element) pour ajouter un élément de n'importe quel type dans la liste, dont l'indice est automatiquement incrémenté. Enfin, la méthode get(indice) permet de récupérer l'élément à l'indice spécifié (le premier élément de la liste a l'indice `0`).

La seconde catégorie que nous allons voir est ce qu'on appelle « Map ». Ce type de collection permet d'associer une clé (de n'importe quel type, on parle de clés au lieu d'indices) avec une valeur (qui est aussi de n'importe quel type). La plus utilisée est la classe « HashMap » qui s'utilise comme suit :

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  HashMap map = new HashMap();
  map.put("nom", "Rakoto");
  map.put("age", 45);
  map.put(1, 3.14);
  Enumeration e = map.elements();
  while(e.hasMoreElements()) {
   System.out.println(e.nextElement());
  }
 }
}
```

Comme pour l'exemple précédent, il suffit d'instancier la classe « HashMap » pour créer ce type de map. Ensuite, pour ajouter un élément correspondant à une clé, on utilise la méthode put(clé, valeur). Enfin, pour parcourir tous les éléments, on utilise la méthode elements() pour obtenir une énumération (ou liste) des éléments, puis on appelle la méthode nextElement() de cette énumération pour obtenir la valeur suivante.

Enfin, la dernière catégorie que nous allons voir est ce que l'on appelle « Set ». Ce type de collection se comporte comme une liste, à la seule différence qu'elle n'accepte pas les doublons d'éléments. Voici un exemple d'utilisation des « HashSet » :

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  HashSet set = new HashSet();
  set.add("Rakoto");
  set.add(15);
  set.add(2.23);
  Object[] tableau = set.toArray();
  for(element : tableau) {
   System.out.println(element);
  }
 }
}
```

Nous voyons que pour créer ce type de collection, il suffit d'instancier la classe « HashSet ». Ensuite, la méthode add(element) permet d'ajouter un élément de n'importe quel type. Enfin, on peut transformer un HashSet en un tableau d'objets grâce à sa méthode toArray(), et dans l'exemple nous avons parcouru ce tableau grâce à la syntaxe de la boucle for spécialisée pour la parcours des tableaux.

4. Les classes génériques

La généricité est un concept très puissant en Java. Elle permet de standardiser le type d'un objet en Java. Expliquons ce concept à l'aide d'un exemple :

Classe « ClasseUn » :

```
public class ClasseUn<T> {
 private T x;
 public ClasseUn(T x) {
  this.x = x;
 }
 public T getX() {
  return this.x;
 }
 public void setX(T x) {
```

```
  this.x = x;
 }
}
```

Classe « ClasseDeux » :

```
public class ClasseDeux<T, S> {
 private T x;
 private S y;
 public ClasseDeux(T x, S y) {
  this.x = x;
  this.y = y;
 }
 public T getX() {
  return this.x;
 }
 public S getY() {
  return this.y;
 }
 public void setX(T x) {
  this.x = x;
 }
 public void setY(S y) {
  this.y = y;
 }
}
```

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  ClasseUn c11 = new ClasseUn<Integer>(5);
  ClasseUn c12 = new ClasseUn<String>("Salut");
  ClasseDeux c2 = new ClasseDeux<Integer, String>(26, "AA");
 }
}
```

Dans la classe « ClasseUn », on a déclaré un attribut « x » de type « T ». Ce type « T » (on aurait pu utiliser un autre nom de type de votre choix) est un type qui n'est pas encore défini

dans cette classe. Le principe est le même pour la classe « ClasseDeux », dont on a utilisé deux types non définis que l'on appelés « T » et « S » respectivement pour les attributs « x » et « y ». Ensuite, c'est au niveau de l'instanciation de ces classes qu'on définit réellement les valeurs de « T » et « S ». En effet, pour l'objet « c11 » T vaut Integer, pour « c12 » T vaut String, et pour l'objet « c2 » T vaut Integer et S vaut String. Ainsi tous les « T » dans l'objet c2 seront remplacés par Integer, et tous les « S » dans cet objet seront remplacés par String.

Il est plus pratique de coupler les classes génériques avec les collections. Nous avons vu précédemment que les éléments des listes sont de n'importe quel type, mais on peut restreindre le type des éléments grâce au concept de généricité.

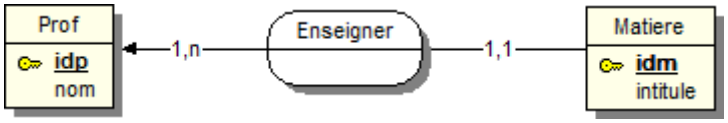
Par exemple, si on veut qu'un ArrayList ne contient que des entiers, on l'instancie comme suit : `ArrayList<Integer> liste = new ArrayList<Integer>()`. Un autre exemple, si on souhaite qu'un HashMap contient des String comme clés et des String comme élément, on l'instancie comme suit : `HashMap<String, String> = new HashMap<String, String>()`.

Chapitre 2. ASSOCIATION AVEC UN S.G.B.D.

1. Le modèle logique objet

Nous allons voir ici comment traduire un modèle conceptuel de données (M.C.D.) en classes Java. Il existe plusieurs manières de traduction, qui dépend aussi du type d'association.

Pour les associations de type un à plusieurs, on peut procéder de plusieurs manières, mais nous préférons utiliser deux classes. Voyons un exemple :



Pour rappel, ce modèle se lit : « Un prof enseigne une ou plusieurs matière, et une matière est enseigné uniquement par un seul prof ».

La manière de faire consiste à transformer chaque entité en sa classe correspondante, puis d'ajouter une liste de l'entité fils (l'entité fils est celle à proximité de la cardinalité 1,1) dans l'entité père (l'entité père est celle à proximité de la cardinalité 1,n), et enfin d'ajouter l'identifiant de l'entité père en tant qu'attribut de l'entité fils. Ainsi, en appliquant cette méthode à notre exemple, on obtient les classes : Prof[idp, nom, matieres{@Matiere}], Matiere[idm, intitule, idp]. Le symbole {} permet de préciser que « matieres » est de type liste, et le symbole @ d'exprimer que chaque élément de la liste est de « Matiere ».

Cette structure de classes s'implémente en Java comme suit :

Classe « Prof » :

```
public class Prof {
 private int idp;
 private String nom;
 private ArrayList<Matiere> matieres;
 public Prof() {
  this.idp = 0;
  this.nom = "";
  this.matieres = null;
 }
 public Prof(int idp, String nom, ArrayList<Matiere>
matieres) {
  this.idp = idp;
  this.nom = nom;
```

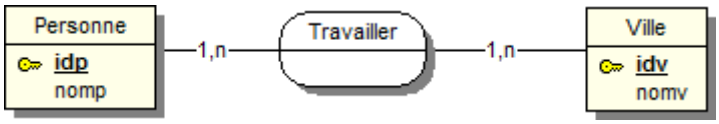
```
  this.matieres = matieres;
 }
 public int getIdp() {
  return this.idp;
 }
 public String getNom() {
  return this.nom;
 }
 public ArrayList<Matiere> getMatiere() {
  return this.matieres;
 }
 public void setIdp(int idp) {
  this.idp = idp;
 }
 public void setNom(String nom) {
  this.nom = nom;
 }
 public void setMatieres(ArrayList<Matiere> matieres) {
  this.matieres = matieres;
 }
}
```

Classe « Matiere » :

```
public class Matiere {
 private int idm;
 private String intitule;
 private int idp;
 public Matiere() {
  this.idm = 0;
  this.intitule = "";
  this.idp = 0;
 }
 public Matiere(int idm, String intitule, int idp) {
  this.idm = idm;
  this.intitule = intitule;
```

```
    this.idp = idp;
  }
  public int getIdm() {
    return this.idm;
  }
  public String getIntitule() {
    return this.intitule;
  }
  public int getIdp() {
    return this.idp;
  }
  public void setIdm(int idm) {
    this.idm = idm;
  }
  public void setIntitule(String intitule) {
    this.intitule = intitule;
  }
  public void setIdp(int idp) {
    this.idp = idp;
  }
}
```

Pour les associations plusieurs à plusieurs et N-aires, il existe aussi plusieurs façons de procéder, mais on préfère utiliser trois classes. Voici un exemple :



La méthode consiste à transformer chaque entité en une classe, et de transformer également l'association en une classe dont les attributs sont les variables dont le type est la classe issue de chaque entité connectée. Si l'association possède des attributs, ces derniers deviennent également attributs de la classe correspondante. Avec notre exemple, on obtient les classes : Personne[idp, nomp], Ville[id, nomv] et Travailler[@Personne, @Ville].

Sous forme de code Java, on obtient :

Classe « Personne » :

```
public class Personne {
```

```
 private int idp;
 private String nomp;
 public Personne() {
  this.idp = 0;
  this.nomp = "";
 }
 public Personne(int idp, String nomv) {
  this.idp = idp;
  this.nomv = nomv;
 }
 public int getIdp() {
  return this.idp;
 }
 public String getNomp() {
  return this.nomp;
 }
 public void setIdp(int idp) {
  this.idp = idp;
 }
 public void setNomp(String nomp) {
  this.nomp = nomp;
 }
}
```

Classe « Ville » :

```
public class Ville {
 private int idv;
 private String nomv;
 public Ville() {
  this.idv = 0;
  this.nomv = "";
 }
 public Ville(int idv, String nomv) {
  this.idv = idv;
  this.nomv = nomv;
```

```
 }
 public int getIdv() {
  return this.idv;
 }
 public String getNomv() {
  return this.nomv;
 }
 public void setIdv(int idv) {
  this.idv = idv;
 }
 public void setNomv(String nomv) {
  this.nomv = nomv;
 }
}
```

Classe « Travailler » :

```
public class Travailler {
 private Personne personne;
 private Ville ville;
 public Travailler() {
  this.personne = null;
  this.ville = null;
 }
 public Travailler(Personne personne, Ville ville) {
  this.personne = personne;
  this.ville = ville;
 }
 public Personne getPersonne() {
  return this.personne;
 }
 public Ville getVille() {
  return this.ville;
 }
 public void setPersonne(Personne personne) {
  this.personne = personne;
```

```
 }
 public void setVille(Ville ville) {
  this.ville = ville;
 }
}
```

2. Interactions avec le S.G.B.D.

Pour pouvoir lier Java avec un SGBD, on aura besoin d'un pilote nommé JDBC (Java DataBase Connector). Dans notre cas, pour lier Java avec MySQL, nous aurons besoin d'un pilote, qui se présenté sous forme d'un fichier jar nommé « mysql connector ». Ceci fait, on peut réaliser n'importe quelle requête SQL en Java.

On travaillera sur la structure suivante :

Table « Prof » :

idp	nom
1	Rakoto
2	Randria
3	Benja

Table « Matiere » :

idm	intitule	idp
1	Java	2
2	C++	1
3	Base de données	3
4	POO en PHP	2

Pour commencer, voyons comment afficher sous Java les données provenant de la base de données :

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  try {
   Class.forName("org.postgresql.Driver");
   String url = "jdbc:mysql://127.0.0.1/ohatra";
   String user = "postgres";
   String passwd = "postgres";
```

```
        Connection cx = DriverManager.getConnection(url, user,
passwd);
        String query = "SELECT * FROM professeur NATURAL JOIN
matiere";
        PreparedStatement prepare = cx.prepareStatement(query);
        ResultSet result = prepare.executeQuery();
        while(result.next()) {
         System.out.println(result.getInt("IDM") + " - " +
result.getString("INTITULE") + " - " +
result.getString("NOM"));
        }
       } catch (Exception e) {
        e.printStackTrace();
       }
      }
     }
```

Tout d'abord, on doit se connecter à la base de données grâce à la méthode DriverManager.getConnection(url, user, passwd) qui contient trois chaînes de caractères comme paramètres, à savoir l'URL du serveur de bases de données (sous la forme "jdbc:nomSGBD://adresseIP:port/nomBaseDeDonnees"), le nom d'utilisateur (ici "root") et le mot de passe (ici il n'y a pas de mot de passe). Ensuite, on formule la requête dans une chaîne nommée « query » dans notre exemple. Ensuite, on crée une requête préparée grâce à la méthode prepareStatement(chaîne) de l'objet « cx » de type Connection. On exécute ensuite la requête préparée grâce à la méthode executeQuery() de l'objet représentant la requête préparée. Enfin, on parcourt les données ainsi collectées, dans l'objet « result », grâce à la méthode next() de cet objet, utilisée pour arrêter la boucle lorsque tous les enregistrements sont parcourus (cette méthode retourne vrai si il y a encore un prochain enregistrement, et retourne faux sinon). À l'intérieur de la boucle, on appelle la méthode getType(chaîne), avec « Type » ici est remplacé par un type Java, qui retourne la valeur de la donnée dans la colonne spécifiée en paramètre. Ainsi, si le champ s'appelle "IDM" dans la base de données, et qu'il est de type entier, on écrit « result.getInt("IDM") »…

Pour les requêtes de type LMD, on procède de la même façon :

Classe « Test » :

```
public class Test {
```

```
    public static void main(String[] args) {
     try {
      Class.forName("org.postgresql.Driver");
      String url = "jdbc:mysql://127.0.0.1/ohatra";
      String user = "postgres";
      String passwd = "postgres";
      Connection cx = DriverManager.getConnection(url, user,
passwd);
      Scanner sc = new Scanner(System.in);
      System.out.println("Saisir le nom du nouveau prof");
      String nom = sc.nextLine();
      String query = "INSERT INTO professeur VALUES(NULL, ?)";
      PreparedStatement prepare = cx.prepareStatement(query);
      prepare.setString(1, nom);
      ResultSet result = prepare.executeUpdate();
      System.out.println("Saisir l'intitulé de la nouvelle
matiere");
      String intitule = sc.nextLine();
      System.out.println("Saisir le numéro du prof
correspondant");
      int id = sc.nextInt();
      query = "INSERT INTO matiere VALUES(NULL, ?, ?)";
      prepare = cx.prepareStatement(query);
      prepare.setString(1, intitule);
      prepare.setInt(2, id);
      prepare.executeUpdate();
     } catch (Exception e) {
      e.printStackTrace();
     }
    }
   }
```

Comme nous pouvons l'observer, la structure est identique à la précédente. On constate quand même quelques nouveautés, comme l'utilisation des points d'interrogation dans les requêtes préparées, qui représentent des « trous ». Pour remplir ces trous, on utilise les

méthodes setType(numéro, valeur) de l'objet représentant la requête préparée. Par exemple, query.setString(2, id) remplit le point d'interrogation numéro 2 de la requête préparée par la valeur de la variable « id ». Enfin, pour réaliser la requête, au lieu d'utiliser la méthode executeQuery(), on utilise la méthode executeUpdate().

Par exemple, pour la mise à jour :

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  try {
   Class.forName("org.postgresql.Driver");
   String url = "jdbc:mysql://127.0.0.1/ohatra";
   String user = "postgres";
   String passwd = "postgres";
   Connection cx = DriverManager.getConnection(url, user,
passwd);
   Scanner sc = new Scanner(System.in);
   System.out.println("Saisir le nouvel intitulé :");
   String intitule = sc.nextLine();
   System.out.println("Saisir le nouveau numéro du
prof :");
   int idp = sc.nextInt();
   System.out.println("Saisir le numéro de la matière à
modifier :");
   int idm = sc.nextInt();
   PreparedeStatement query = "UPDATE matiere SET intitule
= ?, idp = ? WHERE idm = ?";
   prepare = cx.prepareStatement(query);
   prepare.setString(1, intitule);
   prepare.setInt(2, idp);
   prepare.setInt(3, idm);
   prepare.executeUpdate();
  } catch (Exception e) {
   e.printStackTrace();
  }
```

```
    }
   }
```

Et enfin, un exemple de suppression d'enregistrements :

Classe « Test » :

```
public class Test {
 public static void main(String[] args) {
  try {
   Class.forName("org.postgresql.Driver");
   String url = "jdbc:mysql://127.0.0.1/ohatra";
   String user = "postgres";
   String passwd = "postgres";
   Connection cx = DriverManager.getConnection(url, user,
passwd);
   Scanner sc = new Scanner(System.in);
   System.out.println("Saisir le numéro de la matière à
supprimer :");
   int idm = sc.nextInt();
   PreparedeStatement query = "DELETE FROM matiere WHERE
idm = ?";
   prepare = cx.prepareStatement(query);
   prepare.setInt(1, idm);
   prepare.executeUpdate();
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

3. Optimisation : le pattern D.A.O.

Un pattern (ou patron de conception) permet de mieux organiser un programme, en vue de faciliter sa lecture et sa maintenance. Le pattern DAO (Data Access Object) permet de mieux séparer la couche de traitement de données des autres couches, et de mettre efficacement en œuvre le modèle MVC. Dans ce cours, nous n'allons nous occuper que du « modèle ».

Nous aurons besoin d'une classe dans laquelle nous allons stocker la connexion à la base de données :

Classe « Connexion » :

```
public class Connexion {
 private String url = "jdbc:mysql://127.0.0.1/ohatra";
 private String utilisateur = "root";
 private String passe = "";
 private static Connection cx;
 private Connexion() {
  try {
   this.cx = DriverManager.getConnection(url, utilisateur,
passe);
  } catch(SQLException e) {
   e.printStackTrace();
  }
 }
 public static Connection getInstance() {
  if(cx == null) {
   new Connexion();
  }
  return cx;
 }
}
```

Ici, nous avons utilisé le pattern « Singleton », qui a pour objectif de faire à ce qu'on ne puisse avoir qu'une seule instance d'une classe. Dans notre cas, c'est très utile pour que le programme ne se connecte à la base de données qu'une seule fois. En effet, le constructeur est déclaré privé, donc non accessible à l'extérieur de cette classe. Pour avoir une instance de cette classe, il est impératif de passer par la méthode getInstance(), qui permet de réaliser que si l'objet de connexion (ici « cx ») n'existe pas encore (donc nul), on l'instancie ; sinon, on retourne l'objet de connexion déjà existant.

Nous aurons aussi besoin d'une classe abstraite et générique, nommée DAO, dans laquelle nous allons citer (donc aucun corps) les méthodes CRUD (Create = insertion, Read = sélection, Update = mise à jour, Delete = suppression) :

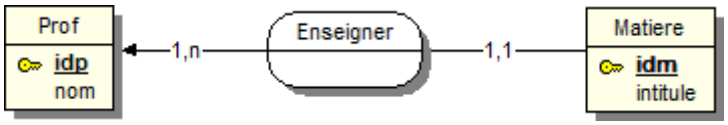
Classe « DAO » :

```
public abstract class DAO<T> {
 protected Connection cx = null;
```

```
    public DAO(Connection cx) {
     this.cx = cx;
    }
    public abstract boolean insertion(T table);
    public abstract boolean miseAJour(T table);
    public abstract boolean suppression(T table);
    public abstract ArrayList<T> selection();
   }
```

L'étape suivante consiste à transformer le MCD en son modèle logique objet, puis pour chaque classe obtenue, on crée les classes DAO correspondantes héritant chacune de la classe DAO, et dont la valeur du type générique est celle de la classe du modèle objet obtenu précédemment. Chaque classe DAO contient l'implémentation des méthodes abstraites dont elle doit redéfinir.

Exemple 1



Nous avons déjà vu le modèle logique objet correspondant.

Créons les classes DAO :

Classe « ProfDAO »

```
public class ProfDAO extends DAO<Prof> {
 public ProfDAO(Connection cx) {
  super(cx);
 }
 public boolean insertion(Prof table) {
  try {
   PreparedStatement ps = this.cx.prepareStatement("INSERT
INTO prof VALUES(null, ?)");
   ps.setString(1, table.getNom());
   ps.executeUpdate();
   return true;
  } catch(SQLException e) {
   e.printStackTrace();
   return false;
```

```
     }
    }
    public boolean miseAJour(Prof table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("UPDATE
prof SET nom = ? WHERE idp = ?");
      ps.setString(1, table.getNom());
      ps.setInt(2, table.getIdp());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public boolean suppression(Prof table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("DELETE
FROM prof WHERE idp = ?");
      ps.setInt(1, table.getIdp());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public ArrayList<Prof> selection() {
     try {
      ArrayList<Prof> liste = new ArrayList<Prof>();
      PreparedStatement ps = this.cx.prepareStatement("SELECT
* FROM prof");
      ResultSet rs = ps.executeQuery();
      while(rs.next()) {
```

```
       Prof p = new Prof(rs.getInt("IDP"),
rs.getString("NOM"), new ArrayList<Matiere>());
       liste.add(p);
      }
      ps = this.cx.prepareStatement("SELECT * FROM matiere");
      rs = ps.executeQuery();
      while(rs.next()) {
       for(int i = 0 ; i < liste.size() ; i++) {
        if(liste.get(i).getIdp() == rs.getInt("IDP")) {
         Matiere m = new Matiere(rs.getInt("IDM"),
rs.getString("INTITULE"), rs.getInt("IDP"));
         liste.get(i).getMatieres().add(m);
        }
       }
      }
      return liste;
     } catch(SQLException e) {
      e.printStackTrace();
      return null;
     }
    }
   }
```

Classe « MatiereDAO »

```
   public class MatiereDAO extends DAO<Matiere> {
    public MatiereDAO(Connection cx) {
     super(cx);
    }
    public boolean insertion(Matiere table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("INSERT
INTO matiere VALUES(null, ?, ?)");
      ps.setString(1, table.getIntitule());
      ps.setInt(2, table.getIdp());
      ps.executeUpdate();
```

```
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public boolean miseAJour(Matiere table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("UPDATE
matiere SET intitule = ?, idp = ? WHERE idm = ?");
      ps.setString(1, table.getIntitule());
      ps.setInt(2, table.getIdp());
      ps.setInt(3, table.getIdm());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public boolean suppression(Matiere table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("DELETE
FROM matiere WHERE idm = ?");
      ps.setInt(1, table.getIdm());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public ArrayList<Matiere> selection() {
     try {
```

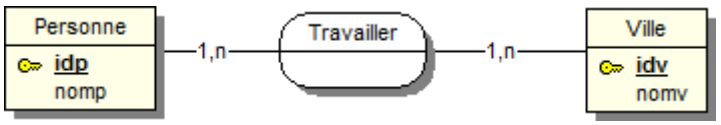
```
      ArrayList<Matiere> liste = new ArrayList<Matiere>();
      PreparedStatement ps = this.cx.prepareStatement("SELECT
* FROM matiere");
      ResultSet rs = ps.executeQuery();
      while(rs.next()) {
       Matiere m = new Matiere(rs.getInt("IDM"),
rs.getString("INTITULE"), rs.getInt("IDP"));
       liste.add(m);
      }
      return liste;
     } catch(SQLException e) {
      e.printStackTrace();
      return null;
     }
    }
   }
```

Classe « Test » :

```
   public class Test {
    public static void main(String[] args) {
     DAO<Prof> profdao = new ProfDAO(Connexion.getInstance());
     DAO<Matiere> matieredao = new
MatiereDAO(Connexion.getInstance());
     Prof p = new Prof(0, "Rasoa");
     Matiere m = new Matiere(0, "Big data", 4);
     profdao.insertion(p);
     matieredao.insertion(m);
     ArrayList<Prof> liste = profdao.selection();
     for(int i = 0 ; i < liste.size() ; i++) {
      System.out.println(liste.get(i).getIdp() + " - " +
liste.get(i).getNom() + " : ");
      for(int j = 0 ; j < liste.get(i).getMatieres().size() ;
j++) {
```

```
System.out.println(liste.get(i).getMatieres().get(j).getIntitu
le());
      }
     }
    }
   }
```

Exemple 2



Classe « PersonneDAO »

```
public class PersonneDAO extends DAO<Personne> {
 public PersonneDAO(Connection cx) {
  super(cx);
 }
 public boolean insertion(Personne table) {
  try {
   PreparedStatement ps = this.cx.prepareStatement("INSERT
INTO personne VALUES(null, ?)");
   ps.setString(1, table.getNomp());
   ps.executeUpdate();
   return true;
  } catch(SQLException e) {
   e.printStackTrace();
   return false;
  }
 }
 public boolean miseAJour(Personne table) {
  try {
   PreparedStatement ps = this.cx.prepareStatement("UPDATE
personne SET nomp = ? WHERE idp = ?");
   ps.setString(1, table.getNomp());
   ps.setInt(2, table.getIdp());
   ps.executeUpdate();
```

```
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public boolean suppression(Personne table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("DELETE
FROM personne WHERE idp = ?");
      ps.setInt(1, table.getIdp());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public ArrayList<Personne> selection() {
     try {
      ArrayList<Personne> liste = new ArrayList<Personne>();
      PreparedStatement ps = this.cx.prepareStatement("SELECT
* FROM personne");
      ResultSet rs = ps.executeQuery();
      while(rs.next()) {
       Personne p = new Personne(rs.getInt("IDP"),
rs.getString("NOMP"));
       liste.add(p);
      }
      return liste;
     } catch(SQLException e) {
      e.printStackTrace();
      return null;
     }
```

```
    }
   }
```

Classe « VilleDAO »

```
   public class VilleDAO extends DAO<Ville> {
    public VilleDAO(Connection cx) {
     super(cx);
    }
    public boolean insertion(Ville table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("INSERT
INTO ville VALUES(null, ?)");
      ps.setString(1, table.getNomv());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public boolean miseAJour(Ville table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("UPDATE
ville SET nomv = ? WHERE idv = ?");
      ps.setString(1, table.getNomv());
      ps.setInt(2, table.getIdv());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public boolean suppression(Ville table) {
     try {
```

```
      PreparedStatement ps = this.cx.prepareStatement("DELETE
FROM ville WHERE idv = ?");
      ps.setInt(1, table.getIdv());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public ArrayList<Ville> selection() {
     try {
      ArrayList<Ville> liste = new ArrayList<Ville>();
      PreparedStatement ps = this.cx.prepareStatement("SELECT
* FROM ville");
      ResultSet rs = ps.executeQuery();
      while(rs.next()) {
       Ville v = new Ville(rs.getInt("IDV"),
rs.getString("NOMV"));
       liste.add(v);
      }
      return liste;
     } catch(SQLException e) {
      e.printStackTrace();
      return null;
     }
    }
   }
```

Classe « TravaillerDAO »

```
   public class TravaillerDAO extends DAO<Travailler> {
    public TravaillerDAO(Connection cx) {
     super(cx);
    }
    public boolean insertion(Travailler table) {
```

```
     try {
      PreparedStatement ps = this.cx.prepareStatement("INSERT
INTO travailler VALUES(?, ?)");
      ps.setInt(1, table.getPersonne().getIdp());
      ps.setInt(2, table.getVille().getIdv());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public boolean miseAJour(Travailler table) {
     return true;
    }
    public boolean suppression(Travailler table) {
     try {
      PreparedStatement ps = this.cx.prepareStatement("DELETE
FROM travailler WHERE idp = ? AND idv = ?");
      ps.setInt(1, table.getPersonne().getIdp());
      ps.setInt(2, table.getVille().getIdv());
      ps.executeUpdate();
      return true;
     } catch(SQLException e) {
      e.printStackTrace();
      return false;
     }
    }
    public ArrayList<Travailler> selection() {
     try {
      ArrayList<Travailler> liste = new
ArrayList<Travailler>();
      PreparedStatement ps = this.cx.prepareStatement("SELECT
* FROM travailler NATURAL JOIN personne NATURAL JOIN ville");
```

```
      ResultSet rs = ps.executeQuery();
      while(rs.next()) {
       Personne p = new Personne(rs.getInt("TRAVAILLER.IDP"),
rs.getString("NOMP"))
       Ville v = new Ville(rs.getInt("TRAVAILLER.IDV"),
rs.getString("NOMV"));
       Travailler t = new Travailler(p, v);
       liste.add(t);
      }
      return liste;
     } catch(SQLException e) {
      e.printStackTrace();
      return null;
     }
    }
   }
```

Classe « Test » :

```
   public class Test {
    public static void main(String[] args) {
     DAO<Travailler> travaillerdao = new
TravaillerDAO(Connexion.getInstance());
     Personne p = new Personne(2, "");
     Ville v = new Ville(3, "");
     Travailler t = new Travailler(p, v);
     travaillerdao.insertion(t);
     ArrayList<Travailler> liste = travaillerdao.selection();
     for(int i = 0 ; i < liste.size() ; i++) {
      System.out.println(liste.get(i).getPersonne().getNomp()
+ " travaille à " + liste.get(i).getVille().getNomv());
     }
    }
   }
```

Nous vous suggérons d'analyser ce code pour le comprendre par vous-même.

Table des matières