Accès aux données

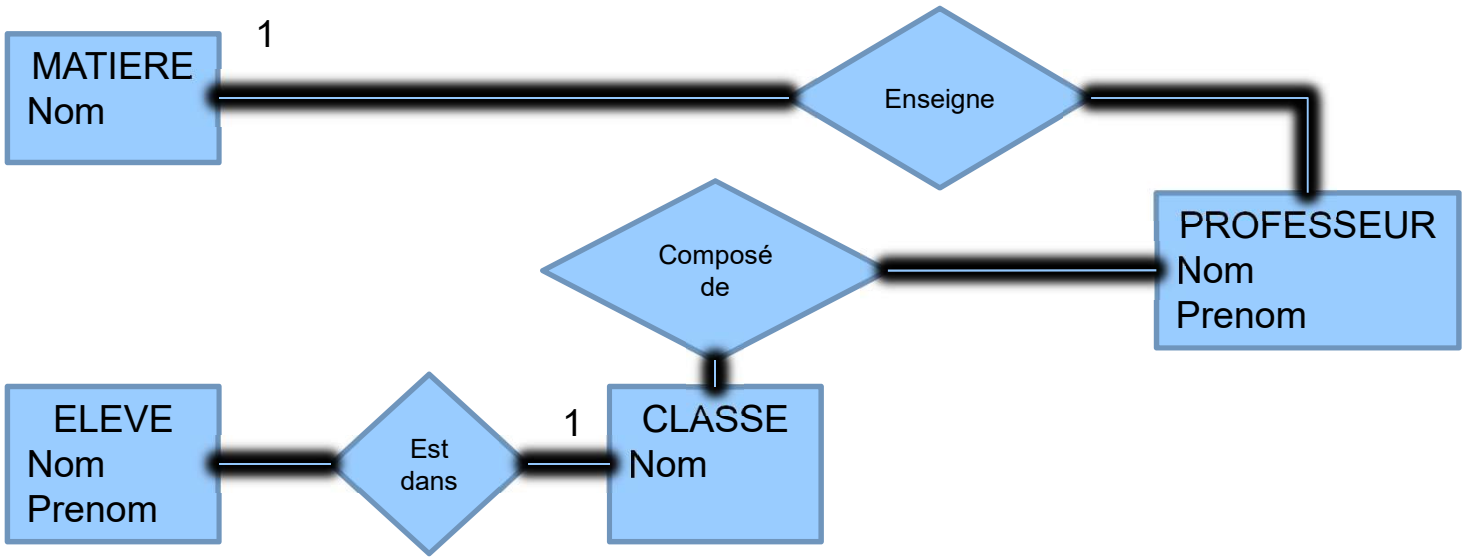
1. Pattern DAO

Plan du chapitre

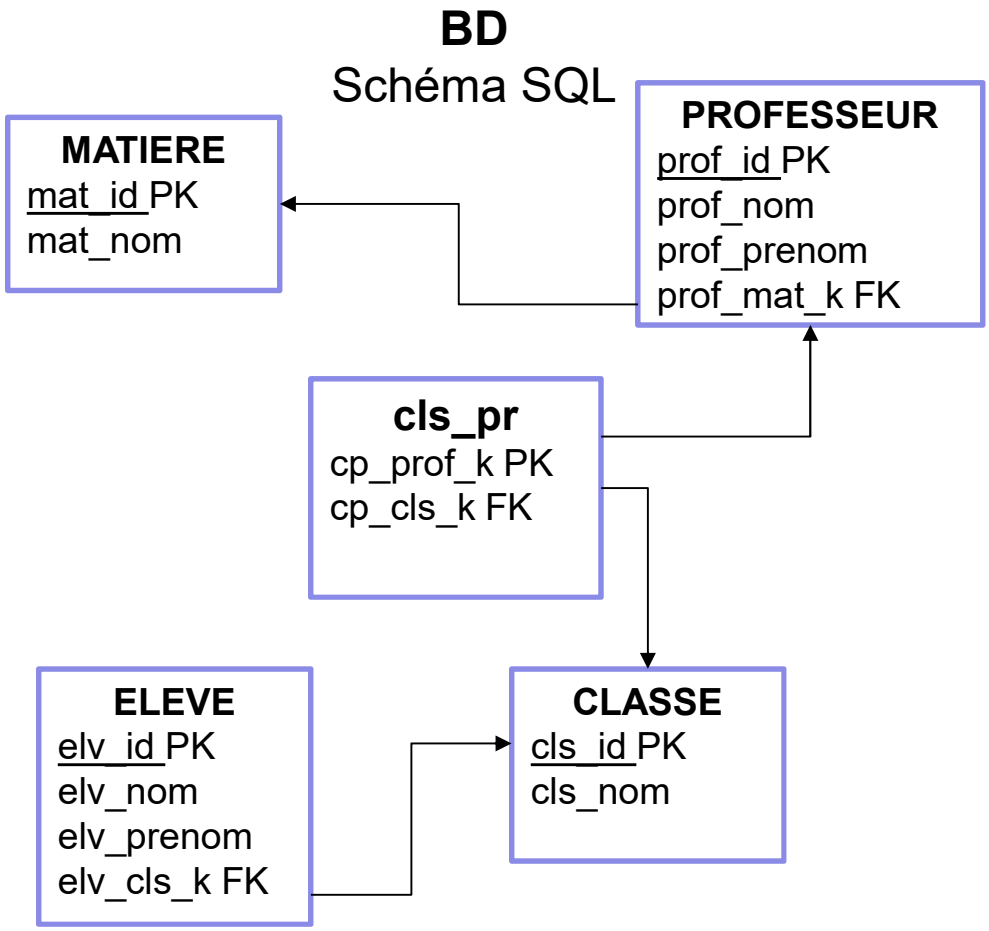
- Contexte et problématiques
  - Schéma de données fil rouge
  - Limitations & Objectifs
- Introduction Pattern DAO
  - Pattern DAO
  - Modèle POJO
- Implémentation de DAO
- Utilisation DAO
  - Instantiation brutale
  - Instantiattion via DataFactory

Contexte: Cas fil rouge

- Gestion simplifiée d'une école:
  - cette école est composée de classes ;
  - chaque classe est composée d'élèves ;
  - à chaque classe est attribué un ensemble de professeur
  - un professeur enseigne une et une seule matière mais peut exercer dans plusieurs classes.



Contexte: Schémas

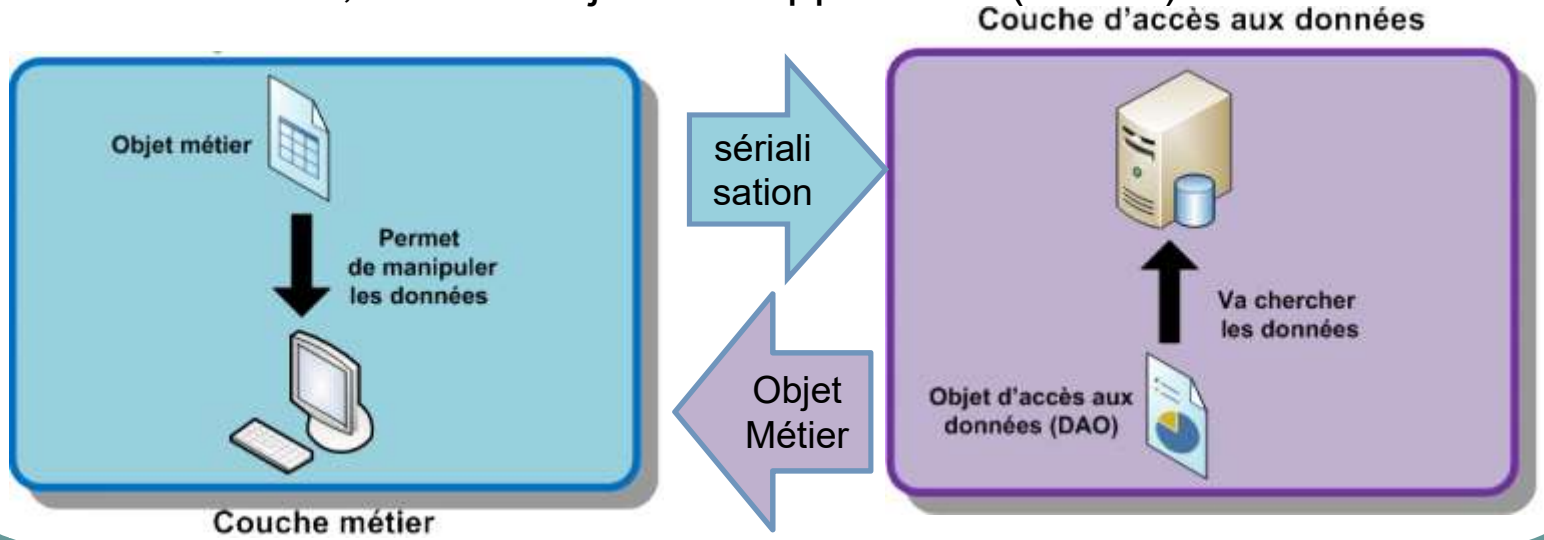


Problématique

- Utiliser les données dans des objets sérialisés dans une base relationnelle
  - une classe est composée de plusieurs élèves et de plusieurs professeurs,
  - un professeur enseigne une matière
  - Les tables de jointures de la base sont symbolisées par la composition dans le diagramme.
- Vous ne savez pas si vos données vont :
  - rester où elles sont ;
  - migrer sur une autre base de données ;
  - être stockées dans des fichiers XML ;
  - Ou même stockées dans différentes sources (SGBDR, XML, etc.
- DAO assure la transparence d'accès aux données

Introduction: patron DAO

- Faire le lien entre la couche d'accès aux données et la couche métier
  - Faire en sorte qu'un type d'objet (DAO) se charge de récupérer les données dans la BD et qu'un autre type métier (Contrôleur) soit utilisé pour manipuler ces données,
  - Les objets DAO doivent fournir des opérations de recherche, d'insertion, de mise à jour et suppression (CRUD)



Introduction: Objet métier POJO

- Coder les objets métiers avec assesseurs et mutateurs :
  - getters et setters les attributs des classes ;
  - Méthode ajout et suppression pour les objets constitués de listes d'objets (collections)
- On obtient des classes d'objets POJO: Plain Old Java Object

```
public class Eleve {
private int id = 0; // Id
private String nom = ""; //Nom de l'élève
private String prenom = ""; //Prénom de l'élève
public Eleve(int id, String nom, String prenom) {
  this.id = id;
  this.nom = nom;
  this.prenom = prenom; }
public Eleve(){};
public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getNom() { return nom; }
public void setNom(String nom) { this.nom = nom; }
public String getPrenom() { return prenom; }
public void setPrenom(String prenom) { this.prenom = prenom; }
}
```

Class Eleve

Introduction: Exemples POJO

Class Matiere

```
public class Matiere {
  private int id = 0;   //ID
  private String nom = "";   //Nom matière

  public Matiere(int id, String nom) {
    this.id = id;
    this.nom = nom;  }

  public Matiere(){}

  public int getId() { return id; }

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

  public String getNom() { return nom; }

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

Introduction: Exemples POJO

Class Matiere

```
public class Matie
  private int id = 0;
  private String no
professeur

  public Matiere(in
    this.id = id;
    this.nom = nom

  public Matiere(){}

  public int getId()

  public void setId(

  public String getN

  public void setNo
this.nom = nom; }
}
```

Class Professeur

```
import java.util.HashSet;
import java.util.Set;
public class Professeur {
  private int id = 0; //ID
  private String nom = ""; //Nom du prof
  private String prenom = ""; //Prénom
  private Matiere saMatiere = null; //matière
enseignée

  public Professeur(int id, String nom,
String prenom, Matiere saMatiere) {
    this.id = id;
    this.nom = nom;
    this.prenom = prenom;
    this.saMatiere= saMatiere }

  public Professeur(){}

  public int getId() { return id; }

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

  public String getNom() { return nom; }

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

  public String getPrenom() { return prenom; }

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

  public String getMatiere() { return saMatiere; }

  public void setMatiere(String saMatiere) {
this.saMatiere = saMatiere; }
}
```

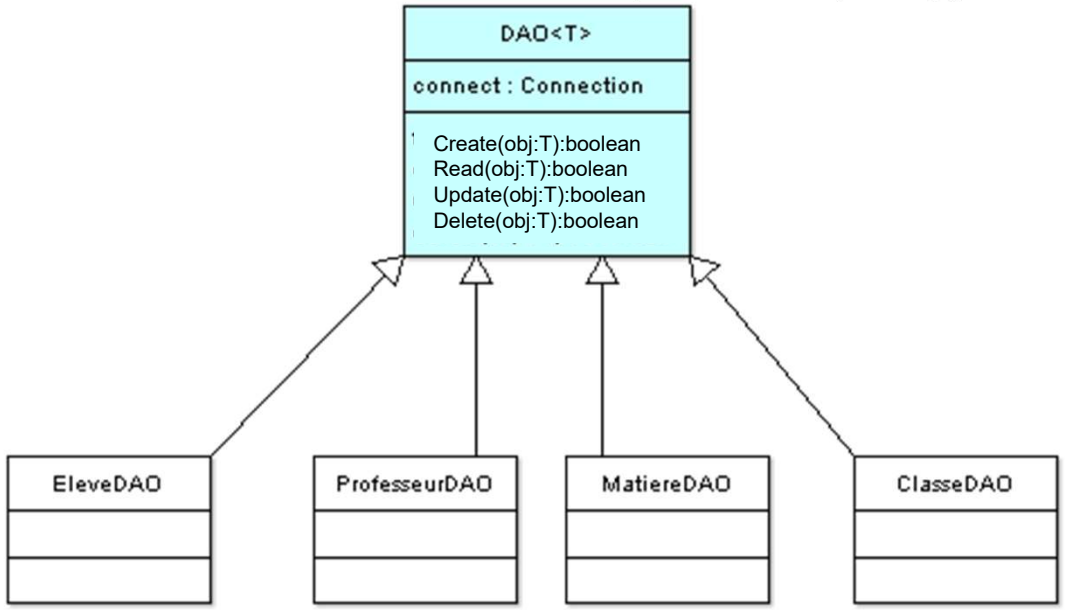
Introduction: Exemples POJO

Class Classe

```
import java.util.HashSet;
import java.util.Set;
public class Classe {
  private int id = 0;
  private String nom = "";
  private Set<Professeur> listProfesseur =
 new HashSet<Professeur>();
  private Set<Eleve> listEleve =
 new HashSet<Eleve>();

  public Classe(int id, String nom) {
    this.id = id;
    this.nom = nom; }

  public Classe(){}

  public int getId() { return id; }

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

  public String getNom() { return nom; }

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

  public Set<Professeur> getListProfesseur() {
    return listProfesseur; }

  public void setListProfesseur(Set<Professeur>
listProfesseur) {
    this.listProfesseur = listProfesseur; }

  public void addProfesseur(Professeur prof) {
    if(!listProfesseur.contains(prof))
      listProfesseur.add(prof);  }

  public void removeProfesseur(Professeur prof ) {
    this.listProfesseur.remove(prof); }

  public Set<Eleve> getListEleve() { return listEleve; }

  public void setListEleve(Set<Eleve> listEleve) {
    this.listEleve = listEleve; }

  //Ajoute un élève à la classe
  public void addEleve(Eleve eleve){
    if(!this.listEleve.contains(eleve))
      this.listEleve.add(eleve);  }

  //Retire un élève de la classe
  public void removeEleve(Eleve eleve){
    this.listEleve.remove(eleve); }

  public boolean equals(Classe cls){
    return this.getId() == cls.getId(); }
}
```

Introduction: super classe DAO

- classe abstraite (ou interface) mettant en œuvre toutes les méthodes CRUD :
  - Polymorphisme
  - À l'aide d'une classe générique
  - Spécialisée avec les classes DAO pour chaque type d'objets



Introduction: super classe DAO

```
import java.sql.Connection;

public abstract class DAO<T> {
protected Connection connect = null;
public DAO(Connection conn){ this.connect = conn; }

public abstract boolean create(T obj);

public abstract T read(int id);

public abstract boolean update(T obj);

public abstract boolean delete(T obj);
}
```

On se laisse la possibilité d'une connection par DAO ! (Donc Singleton ou pas)

CREATE

READ

UPDATE

DELETE

Implémentation DAO: Eleve

```
public class EleveDAO extends DAO<Eleve> {

  public EleveDAO(Connection conn) { super(conn); }
  public boolean create(Eleve obj) {
    try {
      this.connect.createStatement(). executeUpdate("insert into
eleve values(?,?,?,?)");
        …
        } catch (SQLException e) { e.printStackTrace(); }
  };
  ...
  public Eleve read(int id) {
    Eleve eleve;
    try {
      ResultSet result = this.connect.createStatement().
executeQuery("SELECT * FROM eleve WHERE elv_id = " + id);
      if(result.first())
        eleve = new Eleve(
          id,
          result.getString("elv_nom"),
          result.getString("elv_prenom"
        ));
    } catch (SQLException e) { e.printStackTrace(); }
    return eleve;  }
}
```

Connection à l'aide du pattern Singleton ou pas

Implémenter chaque méthode CRUD
Attention aux exceptions (Pk & FK incohérents par exemple)

Focus sur READ
- **Créer Statement**
- **Exécuter requête d'accès**
- **Récupération des données**
- **Peuplement de l'objet métier**

(idem pour MATIERE)

Implémentation DAO: Professeur

```
public class ProfesseurDAO extends DAO<Professeur> {

  public ProfesseurDAO(Connection conn) { super(conn); }
  public boolean create(Professeur obj) { return false; }
  ...
  public Professeur read(int id) {
    Professeur professeur;
    try {
    ResultSet result = this.connect.createStatement().executeQuery(
      "SELECT * FROM professeur +
      "WHERE prof_id = "+ id );
    if(result.first()){
      professeur = new Professeur(id, result.getString("prof_nom"),
        result.getString("prof_prenom"));
      result.beforeFirst();
      MatiereDAO matDao = new MatiereDAO(this.connect);
      while(result.next())
          professeur.saMatiere=matDao.read(result.getInt("prof_mat_k")); }
    } catch (SQLException e) { e.printStackTrace(); }
   return professeur; }
}
```

Connection à l'aide du pattern Singleton

Implémenter chaque méthode CRUD

Focus sur READ

- **Créer Statement**
- **Exécuter requête d'accès (le professeur avec sa matière)**
- **Récupération des données**
- **Peuplement partiel de l'objet métier**

- **Récupération de la matière enseignée**
- **MàJ de l'objet métier**

Implémentation DAO: Classe 1/2

```
public class CLasseDAO extends DAO<Classe> {

  public ClasseDAO(Connection conn) { super(conn); }
  public boolean create(Classe obj) { return false; }
  ...
   public Classe read(int id) {
    Classe classe ;
    try {
      ResultSet result = this.connect.createStatement().executeQuery(
"SELECT * FROM classe WHERE cls_id = " + id);
      if(result.first()){
        classe = new Classe(id, result.getString("cls_nom"));

        result = this.connect.createStatement().executeQuery(
          "SELECTprof_k from cls_pr
           WHERE cls_k = " + id );

        ProfesseurDAO profDao = new ProfesseurDAO(this.connect);
        while(result.next())
         classe.addProfesseur(profDao.read(result.getInt("prof_k")));
```

Connection à l'aide du pattern Singleton

Implémenter chaque méthode CRUD

Focus sur READ

- **Créer Statement**
- **Exécuter requête d'accès**
- **Récupération des données**
- **Peuplement partiel de l'objet métier**

- **Récupération des profs de la classe**
- **Peuplement partiel de la liste des profs de l'objet métier**

Implémentation DAO: Classe 2/2

```
public class CLasseDAO extends DAO<Classe> {
  ...
  public Classe read(int id) {
      …
      EleveDAO eleveDao = new EleveDAO(this.connect);
      result = this.connect.createStatement().executeQuery(
        "SELECT elv_cls_k FROM eleve WHERE elv_cls_k = " + id
      );

      while(result.next())
        classe.addEleve(eleveDao.read(result.getInt("elv_cls_k")));
    }
  } catch (SQLException e) {
    e.printStackTrace();
  }
  return classe;
}}
```

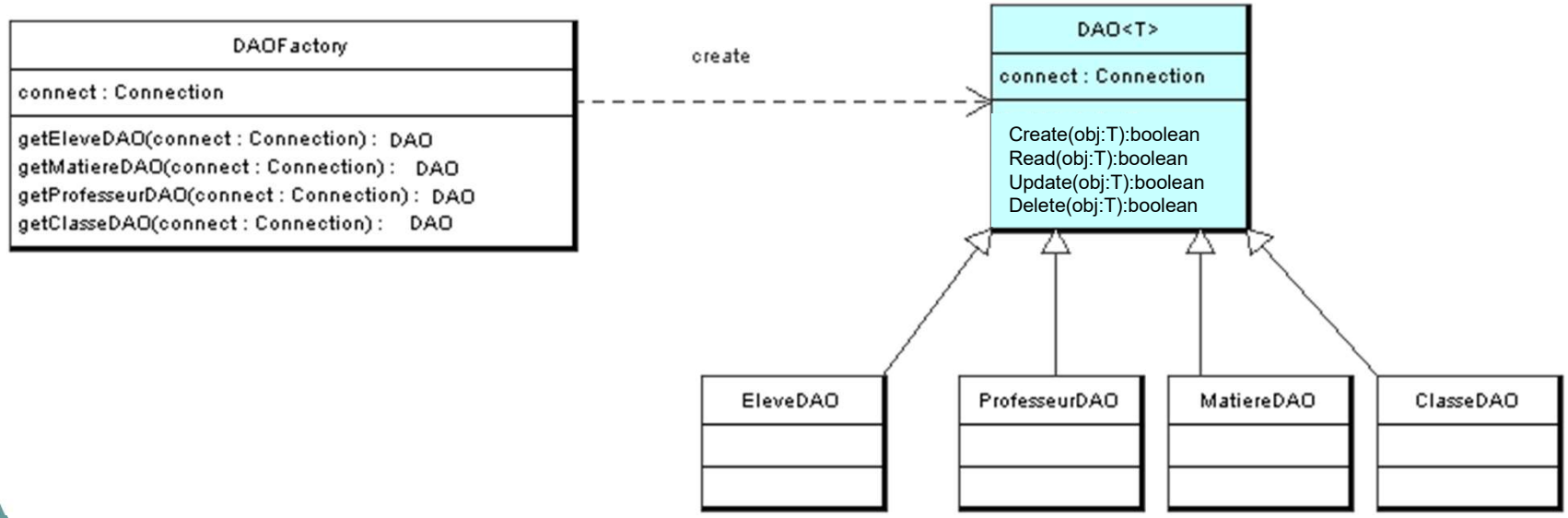
- **Récupération des élèves de la classe**
- **Peuplement de la liste des élèves de l'objet métier**

Utilisation DAO: version basique

```
public class FirstTest {
  public static void main(String[] args) {
    //Testons des élèves
   DAO<Eleve> eleveDao = new  EleveDAO(TheConnection.getInstance());
   for(int i = 1; i < 5; i++){
     Eleve eleve = eleveDao.read(i);
     System.out.println("Elève N°" + eleve.getId() + " - " + eleve.getNom() + " " +
eleve.getPrenom());
   }

   //Testons les professeurs
   DAO<Professeur> profDao = new ProfesseurDAO(TheConnection.getInstance());
   for(int i = 4; i < 8; i++){
     Professeur prof = profDao.read(i);
     System.out.println(prof.getNom() + " " + prof.getPrenom() + " enseigne :
"+prof.getMatiere().getNom());}
…
}
```

Utilisation DAO: version Factory

- Déléguer la création d'objet: pattern Factory
  - Les instances concrètes se font à un seul endroit
  - Les multi fabriques s'implémentent simplement
- Une classe devient la fabrique de DAO



Utilisation DAO: version Factory

- Code d'une fabrique de DAO relationnel

```
public class DAOFactory {
  protected static final Connection conn = TheConnection.getInstance();

  public static DAO getClasseDAO(){ return new ClasseDAO(conn); }

  public static DAO getProfesseurDAO(){  return new ProfesseurDAO(conn); }

  public static DAO getEleveDAO(){ return new EleveDAO(conn); }

  public static DAO getMatiereDAO(){ return new MatiereDAO(conn); }
}
```

Utilisation DAO: version Factory

- Et le code pour utiliser la fabrique

```
public class TestDAO {

  public static void main(String[] args) {

    DAO<Eleve> eleveDao = DAOFactory.getEleveDAO();

    for(int i = 1; i < 5; i++){
      Eleve eleve = eleveDao.read(i);
      System.out.println("\tELEVE N°" + eleve.getId() + " - NOM : " + eleve.getNom() + " -
PRENOM : " + eleve.getPrenom());  }

    DAO<Classe> classeDao = DAOFactory.getClasseDAO();

    Classe classe = classeDao.read(10);
    System.out.println("\tCLASSE DE " + classe.getNom());

    …
}
```