La persistance en Java

Moussa Lo
UFR SAT, UGB
moussa.lo@ugb.edu.sn

La persistance des données

- Fait d'exister dans le temps pour un objet.
- Capacité d'un objet à rester en l'état lorsqu'il est sauvegardé, puis rechargé plus tard.
- Traite des aspects de stockage et de récupération des données.

La persistance en Java

- **JDBC**:
  - API Java permettant l'accès aux SGBDR.
  - Difficile à utiliser.

- **Sérialisation**.

- **Mapping objet-relationnel (ORM)**.

Sérialisation (1/2)

- Sauvegarde de l'état d'un objet en mémoire sur un flux de données (vers un fichier, par exemple).

- Ce concept permettra aussi de reconstruire, ultérieurement, l'objet en mémoire à l'identique de ce qu'il pouvait être initialement à partir de la sauvegarde: *désérialisation*.

Sérialisation (2/2)

- Méthode simple et transparente.

- Pour qu'un objet Java soit sérialisable, il faut qu'il implémente l'interface *java.io.Serializable* et posséder des attributs sérialisables.

- Mécanisme rarement utilisé car pas de langage de requêtes, ni d'infrastructure professionnelle.

JDBC

JDBC

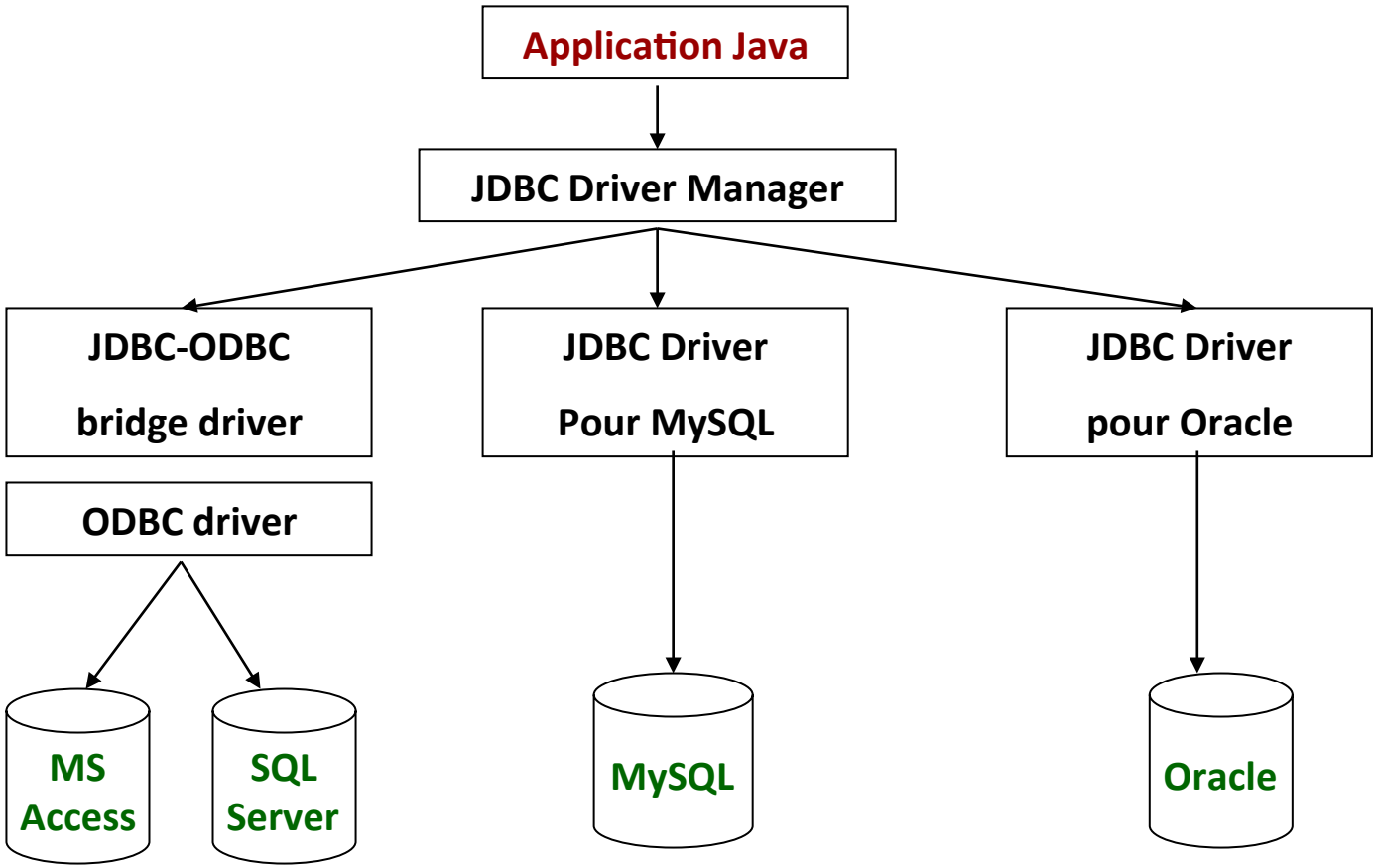
- *Java Database Connectivity*

- API Java permettant l'accès aux SGBDR (MS Access, SQL Server, MySQL, PostgreSQL, Oracle)

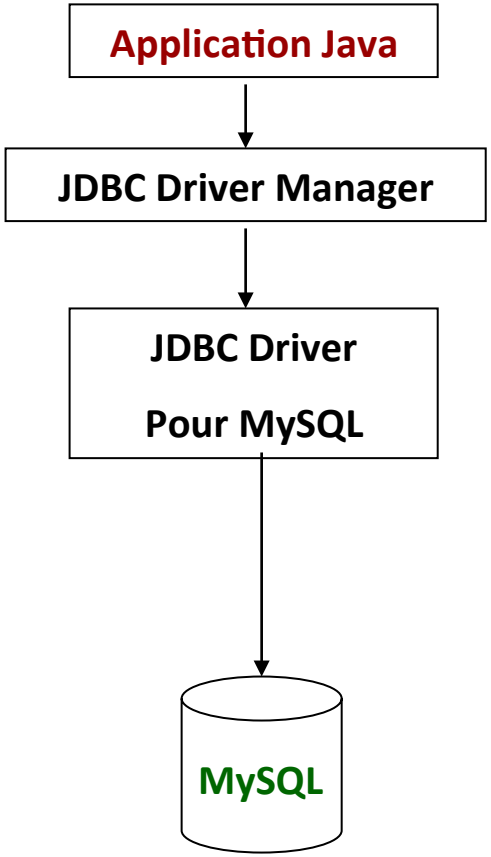
- Fonctionne en client/serveur (appl. Java / SGBD)

- Les interfaces et classes de l'API JDBC se trouvent dans le package ***java.sql***

JDBC : architecture



JDBC : architecture



JDBC : architecture

- JDBC Driver Manager : Gestionnaire de drivers permettant à chaque application de charger le(s) driver(s) dont il a besoin.

- Driver JDBC : gère les détails de communication avec un type de SGBD.

  ✓ Un driver par SGBD (Oracle, MySQL, ...)

  ✓ JDBC-ODBC : driver générique pour toutes les sources accessibles via ODBC (*Open DataBase Connectivity*. interface Microsoft permettant la communication entre des clients bases de données fonctionnant sous Windows et les SGBD du marché).

JDBC : le driver JDBC

- Chaque éditeur de SGBDR fournit un driver JDBC sous la forme d'une archive jar.

- C'est un ensemble de classes qui implémentent les interfaces du package *java.sql*.

- Il faut ajouter l'archive au CLASSPATH afin de pouvoir y accéder dans vos programmes.

Fichiers d'archives JAR : définition

- Java Archive File
- Fichiers d'extension « **.jar** » qui regroupent un ensemble de ressources s'utilisant dans des programmes Java.
- Archives compressés au format « *rar* »
- Permettent de déployer et de distribuer très facilement des **bibliothèques** de classes java.
- Les archives java comportent essentiellement
  - ✓ des classes Java compilées « *.class* »,
  - ✓ et éventuellement toutes sortes de ressources (images, fichiers de configuration, etc.)

Création d'une archive JAR

- Exécuter la commande

jar *cvf* nom_archive liste_des_fichiers

- Les différents fichiers de la liste sont séparés par des espaces.
- L'option « **c** » crée une nouvelle archive.
- L'option « **f** » permet de spécifier le nom de l'archive.
- L'option « **v** », génère un affichage sur la sortie standard.
- *Exple* : jar cvf essai.jar f1.class f2.class

Extraction et Exécution d'une archive JAR

- Extraction du contenu d'une archive JAR :

jar xf nom_archive [liste_des_fichiers_à_extraire]

- Exécution d'une archive JAR :

java [options] –jar archive_java.jar [classe_principale]

JDBC : fonctionnement

JDBC fonctionne comme suit :

- Création d'une connexion à la BD
- Envoi de requêtes SQL (pour récupérer ou maj des données)
- Exploitation des résultats provenant de la base
- Fermeture de la connexion

JDBC : connexion à une BD (1/2)

1. **Charger la classe du driver JDBC**

Cette classe implémente l'interface java.sql.Driver et peut être chargée en appelant la méthode ***forName*** de ***java.lang.Class***

Exemple avec le driver de MySQL

```
Class.forName("org.gjt.mm.mysql.Driver");
```

JDBC : connexion à une BD (2/2)

2. **Appeler la méthode getConnection() de java.sql.DriverManager**

```
java.sql.Connection co;
co = DriverManager.getConnection("jdbc:mysql://
    localhost/MABD");

co = DriverManager.getConnection("jdbc:mysql://
    localhost/MABD,      "admin", "passer");
```

JDBC : exemple de connexion

```
import java.sql.*; import javax.swing.JOptionPane;

public static Connection initConnection() {
        Connection co = null;
        String url = "jdbc:mysql://localhost/MABD";
        try{
                Class.forName("org.gjt.mm.mysql.Driver");
                co = DriverManager.getConnection(url,"root",null);
                JOptionPane.showMessageDialog(null,"Connection OK");
                return co;
        }
        catch (ClassNotFoundException fe) {
           System.out.println("driver introuvable : " +fe.getMessage());}
        catch (SQLException se) {
           System.out.println("connexion impossible : " +se.getMessage());}
}
```

JDBC : requêtes SQL

- JDBC permet divers types de requêtes SQL : interrogation, maj, création de tables.

- Les objets suivants sont disponibles :

  ✓ **ResultSet** : contient des informations sur une table (noms des colonnes) ou le résultat d'une requête SQL.

```
Statement st = co.createStatement();
ResultSet rs = (ResultSet)st.executeQuery("Select …)";
```

  ✓ **ResultSetMetaData** : contient des informations sur le nom et le type des colonnes d'une table

```
ResultSetMetaData rsmd = rs.getMetaData();
int nbre_Colonnes = rsmd.getColumnCount();
```

  ✓ **DataBaseMetaData** : contient les informations sur la BD (noms des tables, index, etc.)

JDBC : l'interface java.sql.Connection (1/2)

- **createStatement** : retourne une instance de *java.sql.Statement* utilisée pour éxecuter une instruction SQL sur la base de données

- **prepareStatement** :précompile des instructions SQL paramétrées et retourne une instance de *java.sql.PreparedStatement*

- **prepareCall** : prépare l'appel à une procédure stockée et renvoie une instance de *java.sql.CallableStatement*

JDBC : l'interface java.sql.Connection (2/2)

- **setAutoCommit, commit, rollback** : gèrent les transactions

- **getMetaData** : renvoie une instance de *java.sql.DatabaseMetaData* pour obtenir des informations sur la base de données

- **close, isClosed** : gèrent la fermeture de la connection

JDBC : java.sql.Statement

- La méthode *createStatement* d'une connection retourne une instance de *java.sql.Statement* dont les méthodes les plus utilisées sont :

✓ **executeUpdate** : permet de mettre à jour les données d'une base en exécutant des instructions SQL de maj

✓ **executeQuery** : permet d'exécuter des requêtes sélection; renvoie une instance de *java.sql.ResultSet*

JDBC : java.sql.ResultSet (1/2)

- Permet de récupérer et d'exploiter les résultats d'une requête Sélection

- Des méthodes **next**, **first**, **last** permettent de parcourir la liste des enregistrements retournés par la sélection SQL

```
java.sql.ResultSet rs = st.executeQuery("Select …");
while (rs.next()) {
        // interrogation des infos de l'enregistrement courant
}
```

JDBC : java.sql.ResultSet (2/2)

- Des méthodes ***getXXX()*** renvoient la valeur d'un des champs de l'enregistrement :
  - ✓ getString()
  - ✓ getInt()
  - ✓ getDate()
  - ✓ getObject()
  - ✓ etc.

- *Exemple* : System.out.println(rs.getString(1), rs.getString("prenom"), rs.getDouble(3));

JDBC : exemple de requête SQL

```
public test_jdbc {
        public static void main (String[] args) {
                Connection maCo = initConnection();
                if (maCo == null) return;
                String req = "Select nom, prenom, age from Personne";
                try{
                    Statement st = maCo.createStatement();
                    ResultSet rs = st.executeQuery(req);
                    while (rs.next()) {
                        System.out.print("nom :"+rs.getString("nom"));
                        System.out.print("prenom:"+rs.getString(2));
                        System.out.println("age :"+rs.getDouble(3));
                    }
                    rs.close(); st.close(); maCo.close();
                }
                catch (SQLException se) {
                        System.out.println("connexion impossible");
                }
        }
}
```

JPA

Exemple

```
public class Personne {

    private int id;
    private String nom;
    private String prenom;

    // methodes
}
```

```
CREATE TABLE PERSONNE (

ID INT NOT NULL,
NOM VARCHAR(255),
PRENOM VARCHAR(255),

PRIMARY KEY(ID))
```

Mapping objet-relationnel

- L'accès aux données est délégué à un outil externe :
  - ☞ Frameworks : *Hibernate*, *Toplink.*
  - ☞ APIs: *JDO, **JPA**.*

- *Avantage* : propose une vue OO d'une structure de données relationnelles.

- Les outils de mapping O-R mettent en correspondance les objets et les données de la base.

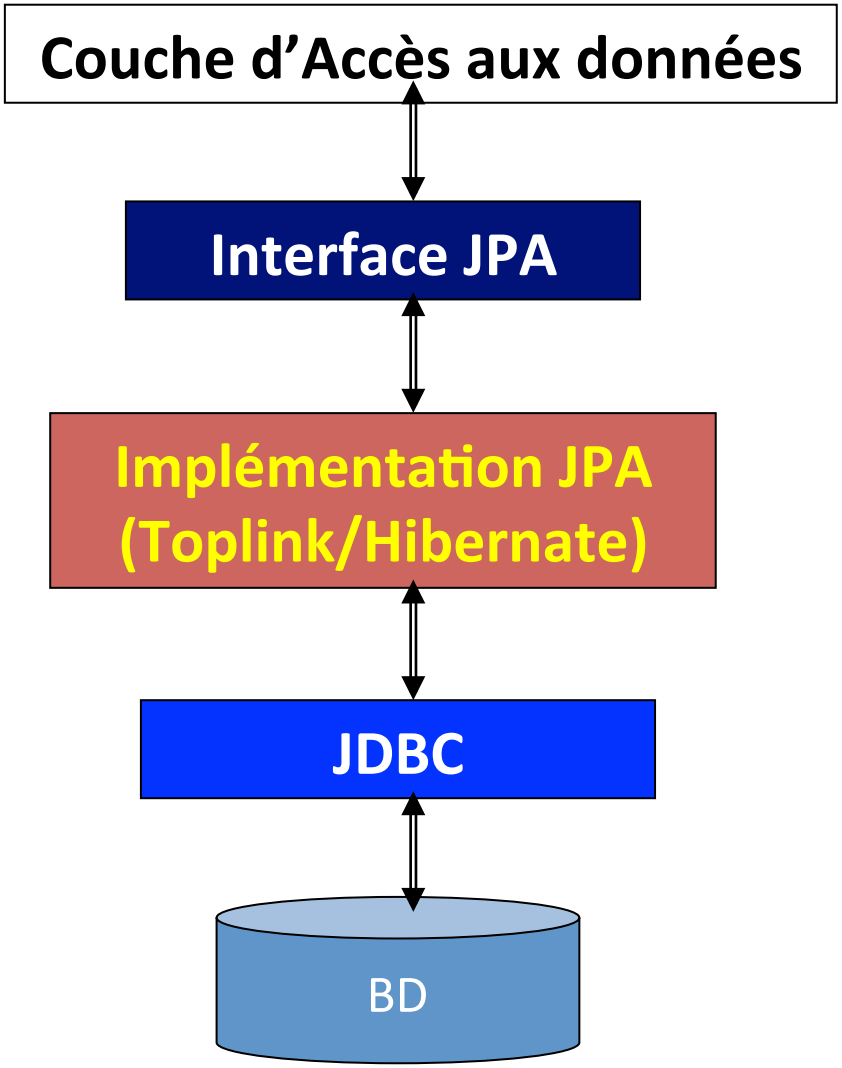
JPA (Java Persistence API)

- Spécification sortie en mai 2006.
- Abstraction au-dessus de JDBC.
- Permet de s'affranchir de SQL.
- Package *javax.persistence*
- Gère la persistance des objets de l'application sur des BD relationnelles par l'intermédiaire d'un ORM.
- Permet de mapper les objets Java (*POJO*) avec les tables de la base de données.

Services de JPA

- Mécanisme permettant de **définir le mapping O/R** de façon déclarative avec :
  - ✓ des ***annotations***
  - ✓ des descripteurs XML
- **API** permettant d'effectuer les opérations de base pour la persistance des données (CRUD) en manipulant simplement des objets Java.
- **Langage de requêtes** standard pour la récupération des objets : JPQL.

Architecture de JPA



Fournisseur de persistance

- JPA nécessite un fournisseur de persistance qui implémente l'API.

- JPA est implémentée par deux produits de références :

  ✓ *TopLink* de Oracle, produit commercial devenu libre.

  ✓ *Hibernate*, projet open-source.

Entités

- Une classe dont les instances peuvent être persistantes est appelée **Entité** (***Entity***).

- Une entité réprésente généralement une table dans une BD relationnelle.

- Chaque instance d'une entité représente une ligne dans la table asociée à l'entité.

Classe Entité (1/4)

- On indique qu'une classe est une entité en lui associant l'**annotation @Entity**

- ***Exemple :***

```
import javax.Persistence.Entity;

@Entity
public class  Personne {
      …
}
```

Annotations

- ✓ Métadonnées permettant :
  - d'ajouter des données sémantiques au code.
  - de préciser la façon dont ces données doivent être traitées.
  - à certains outils de générer des constructions additionnelles à la compilation ou à l'exécution ou de renforcer un comportement voulu au moment de l'exécution.

Annotations

✓ Contrairement aux commentaires JavaDoc, les annotations ne disparaissent pas lors de la compilation.

✓ Sont reconnues et traitées par le compilateur et sont généralement conservées avec les classes produites.

Classe Entité (2/4)

Une classe entité doit respecter certaines règles :

- Les propriétés et les méthodes ne doivent pas être finales.

- Les variables d'instance de persistance doivent être déclarées *private* ou *protected*.

Classe Entité (3/4)

- Si une instance de l'entité peut être envoyée à un client distant, la classe doit implémenter ***java.io.Serializable*** (RMI utilise la sérialisation pour passer les arguments entre le client et le serveur).

- Doit posséder un constructeur sans argument mais peut posséder aussi d'autres constructeurs spécifiques.

Classe entité (4/4)

- Tout champ non statique est automatiquement considéré comme persistant par le conteneur.

- Les annotations peuvent se placer :
  - soit sur les propriétés
  - soit sur les accesseurs

Exemples d'entité (1/3)

```
@Entity
public class Personne {
    @Id
    @GeneratedValue
    private int id;

    private String nom;
    private String prenom;
        private int getId(){ return id; }
        …
}
```

Exemples d'entité (2/3)

```
@Entity
public class Personne implements Serializable {
    private int id;
    private String nom;
    private String prenom;
    @Id
    @GeneratedValue
    private int getId(){ return id;}
        …
}
```

Exemples d'entité (3/3)

```
@Entity
public class Personne implements Serializable {
    ...
    public Personne() {}
    public Personne(String prenom, String nom) {
        this.prenom = prenom;
        this.nom = nom;
    }
}
```

Entité - Table

- Généralement, une table correspond à une classe :
  - ✓ le **nom de la table** est le **nom de la classe.**
  - ✓ les **noms des colonnes** correspondent aux **noms des attributs persistants**.

- Par exemple, les données de la *classe* **Personne** sont enregistrées dans la *table* **Personne** dont les colonnes seront *id*, *nom*, *prenom*.

Entité - Table

```
@Entity
public class Personne {
    @Id
    private int id;
    private String nom;
    private String prenom;
    // methodes get et set
}
```

```
CREATE TABLE PERSONNE (

ID INT NOT NULL,
NOM VARCHAR(255),
PRENOM VARCHAR(255),

PRIMARY KEY(ID))
```

Annotations élémentaires du mapping OR (1/2)

@Table :

- ✓ **pour spécifier le nom de la table**
- ✓ **par défaut c'est le nom de la classe**

```
@Entity
@Table(name="t_Personne", schema="MaBD")
public class Personne { … }
```

Annotations élémentaires du mapping OR (2/2)

@Column :

- ✓ **pour définir les propriétés d'un champ**

```
@Entity
public class Personne {
    @Column(name="t_nom", nullable=false, length=32)
    private String nom;
}
```

Associations

Les 4 relations possibles entre entités sont signalées par des annotations :

- **Un à Un :**

  ***Annotation* : @OneToOne**
- **Un à plusieurs :**

  ***Annotation* : @OneToMany**
- **Plusieurs à Un**

  ***Annotation* : @ManyToOne**
- **Plusieurs à Plusieurs**

  ***Annotation* : @ManyToMany**

Exemple d'association

```
@OneToMany(mappedBy="section")
public Collection<Enseignant> getEnseignants() {
        return enseignants;
}

public void setEnseignants(Collection<Enseignant>
ens) {
        this.enseignants = ens;
    }
```

Entity Manager (1/2)

- Service qui centralise toutes les actions de persistance.
- Pour rendre les entités persistantes, on le précise via l'interface ***javax.persistence.EntityManager***.
- Correspond à l'état d'une connexion avec la BD.
- Offre des méthodes d'ajout, modification, suppression, recherche et un accès au langage de requêtes.

Entity Manager (2/2)

- Lorsqu'une entité est prise en compte par l'EM, on dit qu'il est attaché ou managé (*managed bean*).
- Dès qu'un objet est managé, on peut alors effectuer des opérations de persistance via l'EM.

- L'ensemble des entités managées par l'EM est appelé contexte de persistance.
- Le **contexte de persistance** permet de préciser :
  - le type de base de données à utiliser
  - les paramètres de connexion.

Création d'une Entity Manager

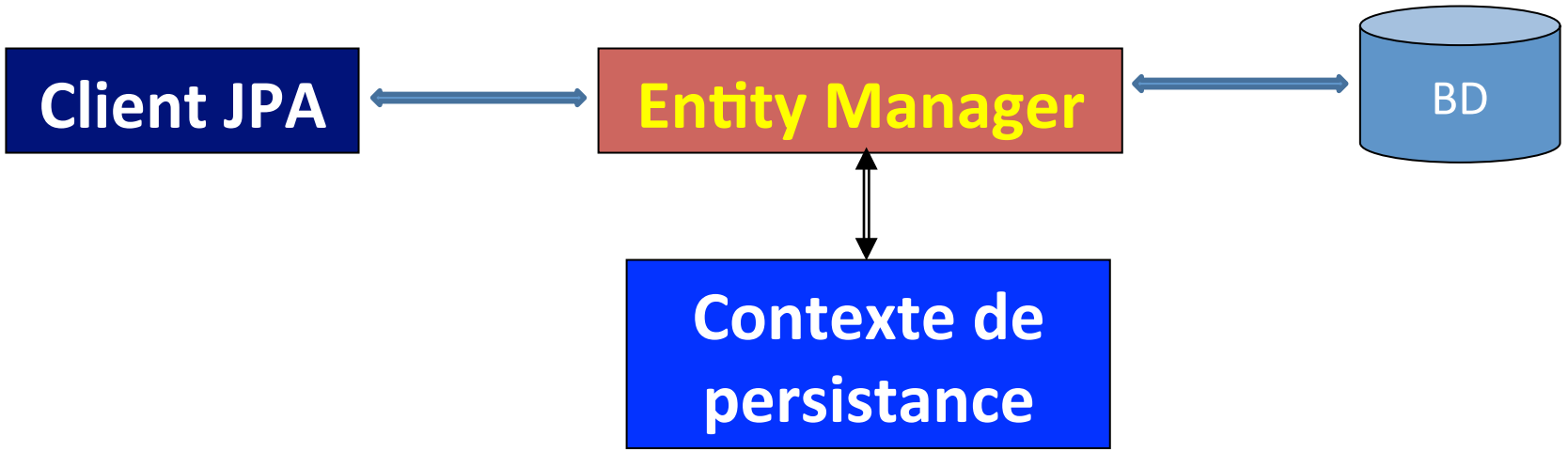
Une EM est obtenue à partir de l'interface

EntityManagerFactory

EntityManagerFactory emf =

Persistence.createEntityManagerFactory(unite_de_persistance);

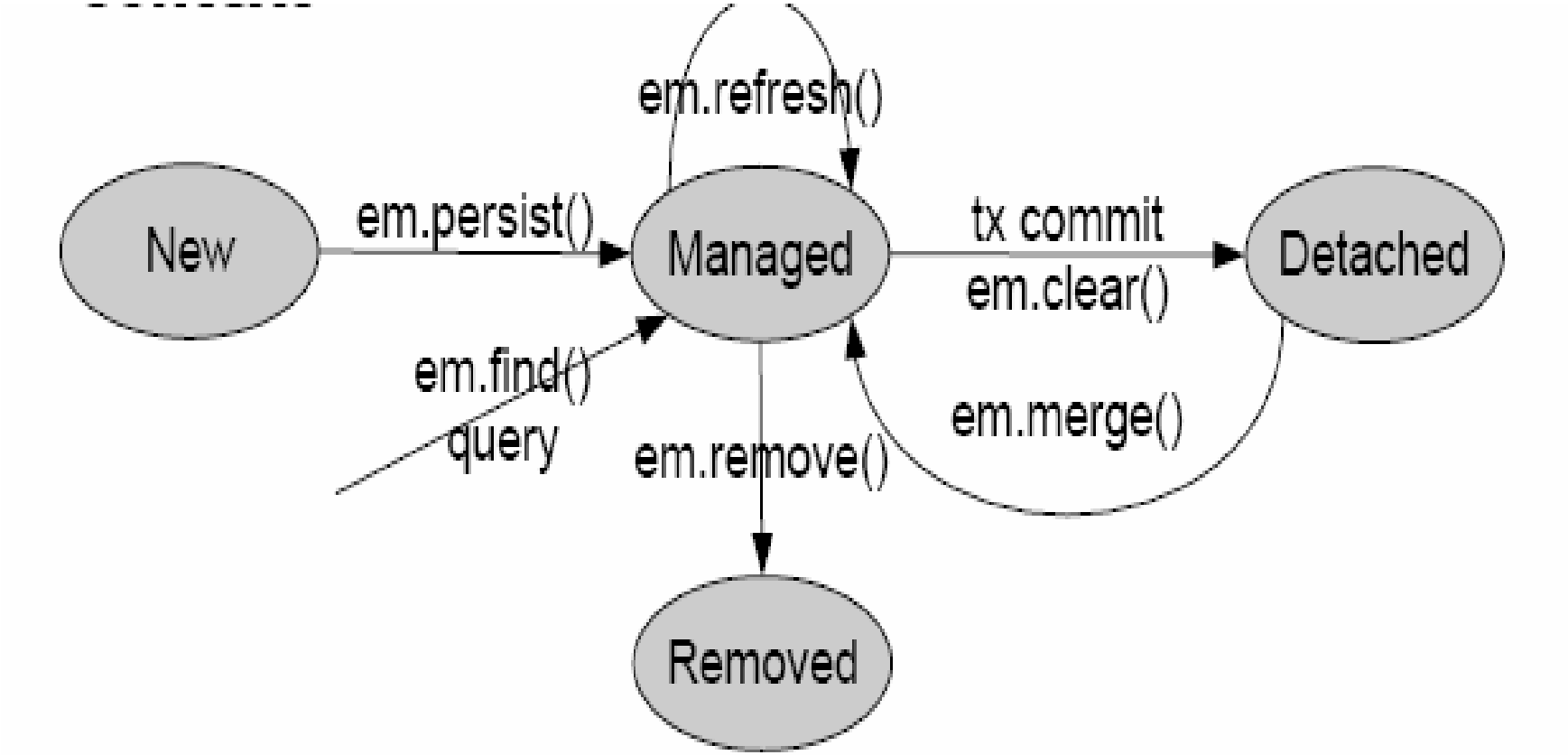
EntityManager em = emf.createEntityManager();



Exemple de contexte de persistence

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
    java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name=“PersonUnit">
        <class>Personne</class>
        <properties>
            <property name="toplink.jdbc.url" value="jdbc:mysql://localhost:
    3306/BD"/>
            <property name="toplink.jdbc.user" value="root"/>
            <property name="toplink.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="toplink.jdbc.password" value=""/>
            <property name="toplink.ddl-generation" value="drop-and-create-
    tables"/>
        </properties>
    </persistence-unit>
</persistence>
```

Cycle de vie d'une entité



Opérations principales sur les entités

- ✓ Persistance d'une entité
- ✓ Recherche d'entités
- ✓ Suppression d'une entité
- ✓ Mise à jour d'une entité
- ✓ Gestion des transactions

Persistance d'une entité

- Insérer l'entité dans la BD.
- Méthode ***void persist(Object o)***
- ***Exemple :***

```
Personne p = new Personne();
p.setNom("Lo");
p.setPrenom("Moussa");
…
em.persist(p);
```

Recherche d'une entité

- Retrouver une entité stockée dans la BD à partir du nom de la classe et de la clé primaire.
- Méthode

  <T> T find(Class<T> entityClass, Object primaryKey)

- ***Exemple :***

```
public Personne trouverPersonne(int id){
    return em.find(Personne.class, id);
}
```

Suppression d'une entité

- Supprimer une entité stockée dans la BD à partir du nom de la classe et de la clé primaire.

- Méthode ***void remove(Object entity)***

- ***Exemple :***

```
public void supprimerPersonne(int id) {
    Personne p = em.find(Personne.class, id);
    if (p != null)
        em.remove(p);
}
```

Mise à jour d'une entité

- Mettre à jour une entité stockée dans la BD.

- **Exemple :**

```
public  Personne modifierPassword (int id, String newPwd){
      Personne p = em.find(Personne.class, id);
      if (p != null)
                  p.setPassword(newPwd);
      return p;
}
```

Le langage de requêtes JPQL

- Java Persistence Query Language.
- Déclaratif et inspiré de la syntaxe de SQL.
- Manipule des objets dans sa syntaxe de requêtes et retourne des objets en résultat.

- *Exemple :*

```
String req = "SELECT p FROM Personne p WHERE
     p.Nom = 'Lo'";
Query query = em.createQuery(req);
List resultat = (List)query.getResultList();
```

Exemples de Requêtes JPQL (1/2)

```
String requete = “SELECT e FROM Personne e";

    Query query = em.createQuery(requete);

    Collection<Personne> pers = query.getResultList();

    for (Personne p : pers)
        System.out.println(" Personne trouvee : " + p);
```

Exemples de Requêtes JPQL (2/2)

```
String requete = “SELECT e FROM Personne e “
+ “ WHERE e.salaire >= :salaire”;

    Query query = em.createQuery(requete);
Query.setParameter(“salaire”, 400000);
List<Personne> pers = query.getResultList();

for (Personne p : pers)
        System.out.println(" Personne : " + p.getNom());
```

Transactions

- L'Entity Manager offre des méthodes pour la gestion des transactions.
- ***Exemple :***

```
// creation et persistance d'une entité personne
em.getTransaction().begin();
Personne p = new Personne();
p.setNom("Lo");
p.setPrenom("Moussa");
 …
em.persist(p);
em.getTransaction().commit();
```

Biblio & Webo-graphie

- **EJB 3**, Laboratoire SUPINFO des technologies Sun, *Dunod*, 2006.
- **Développement JEE 5 avec Eclipse Europa,** K. Djaafar, *Eyrolles*, 2007.
- **Les cahiers du programmeur Java EE 5**, Antonio Goncalves, *Eyrolles*, 2007.
- **Développons en Java**, J. M. Doudoux, Tutorial en ligne, http://perso.wanadoo.fr/jm.doudoux/java/tutorial/
- **Site officiel de Java**