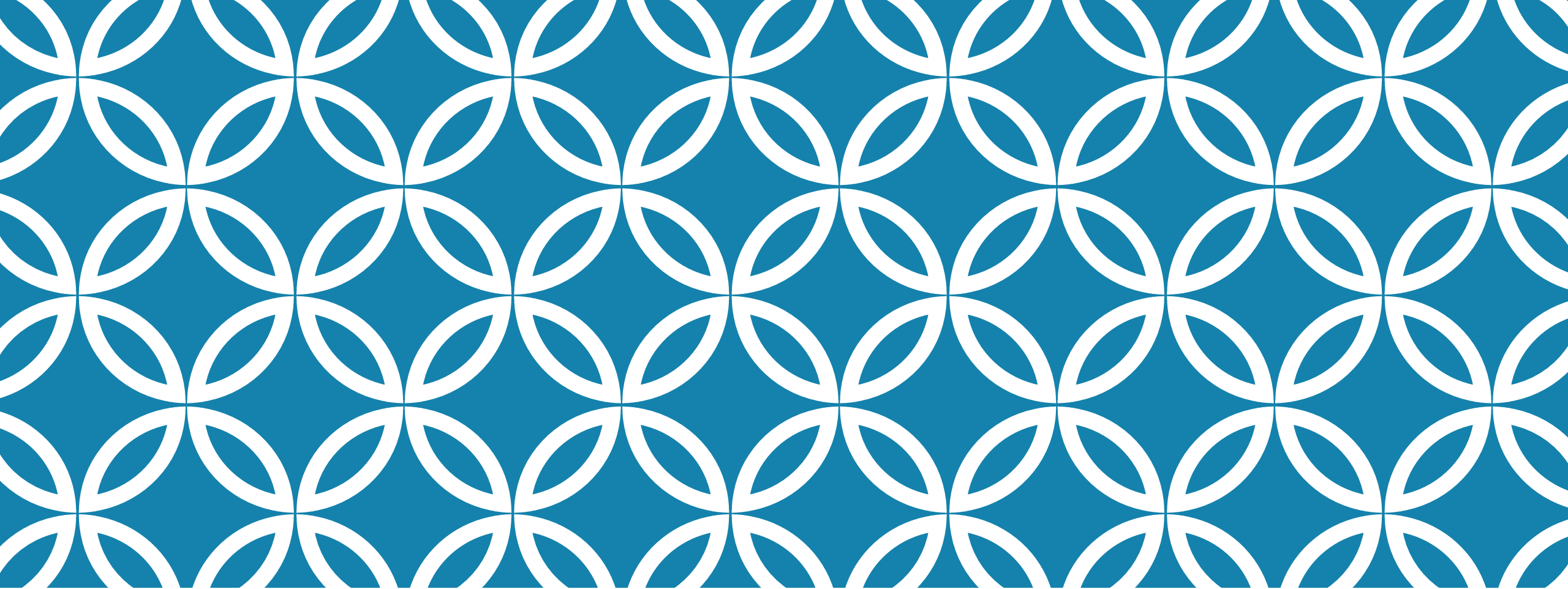


# JPA / HIBERNATE

Jordan ABID



**BASES DE DONNÉES**

# RAPPELS SGBDR

Systèmes de Gestion de Bases de Données Relationnelles

Outil pour

- Structurer
- Stocker
- Interroger
- Garantir l'**intégrité** des données

Processus actif

Accessible via un port de communication spécifique

Langage DDL

- Définir la structure des données

Langage DRL / DML

- Interroger, créer, supprimer et manipuler les données

# RAPPELS SGBDR

## Transactions

### Suite d'instructions

### ACID

- Atomique                      Indivisible, tout ou rien
- Cohérente                    Le contenu final (dans la base de données) doit être cohérent
- Isolée                         Une transaction ne doit pas interférer avec une autre
- Durable                      Le résultat final est conservé indéfiniment (persistance de la donnée)

- Une transaction démarre (begin)
- Une transaction va jusqu'au bout et se termine bien (commit)
- Une transaction ne va pas jusqu'au bout (rollback)
- Toutes les instructions de la transaction sont annulés !

# RAPPELS SGBDR

Un SGBD peut gérer plusieurs bases de données

Une base de données peut contenir plusieurs tables

Une table possède plusieurs colonnes

Chaque enregistrement est identifié grâce à une clé primaire

On peut créer un lien entre enregistrements grâce à la clé étrangère

# RAPPELS SGBDR

Quelques serveurs

- MySQL
- MariaDB
- Oracle
- PostgreSQL
- Microsoft SQL Server
- SQLite
- ...

# ACCÈS EN JAVA

L'accès le plus bas niveau avec Java  
Driver JDBC adapté au serveur SQL manipulé

On doit charger ce driver adapté  
La classe doit être présente dans le *classpath*

Pour s'y connecter  
Il faut connaître l'URL de connexion

```
try { Class.forName("com.mysql.jdbc.Driver");  
}  
catch(ClassNotFoundException e) {  
    //...  
}
```

```
try {  
    Connection myConnection =  
    DriverManager.getConnection("jdbc:mysql://localhost:3306/eshop", "username", "password"); }  
catch(SQLException e) { //...  
}
```

# ACCÈS EN JAVA

Exécuter des requêtes

Création d'un Statement

Récupération du résultat dans un ResultSet avec la méthode `executeQuery()`

*Il existe aussi `execute()` et `executeUpdate()`*

```
Statement myStatement = myConnection.createStatement();
ResultSet myResult = myStatement.executeQuery("SELECT PRO_ID, PRO_LIBELLE,
PRO_PRIX FROM produit");

while(myResult.next()) { System.out.println(myResult.getString("PRO_LIBELLE"));
//...
}
```

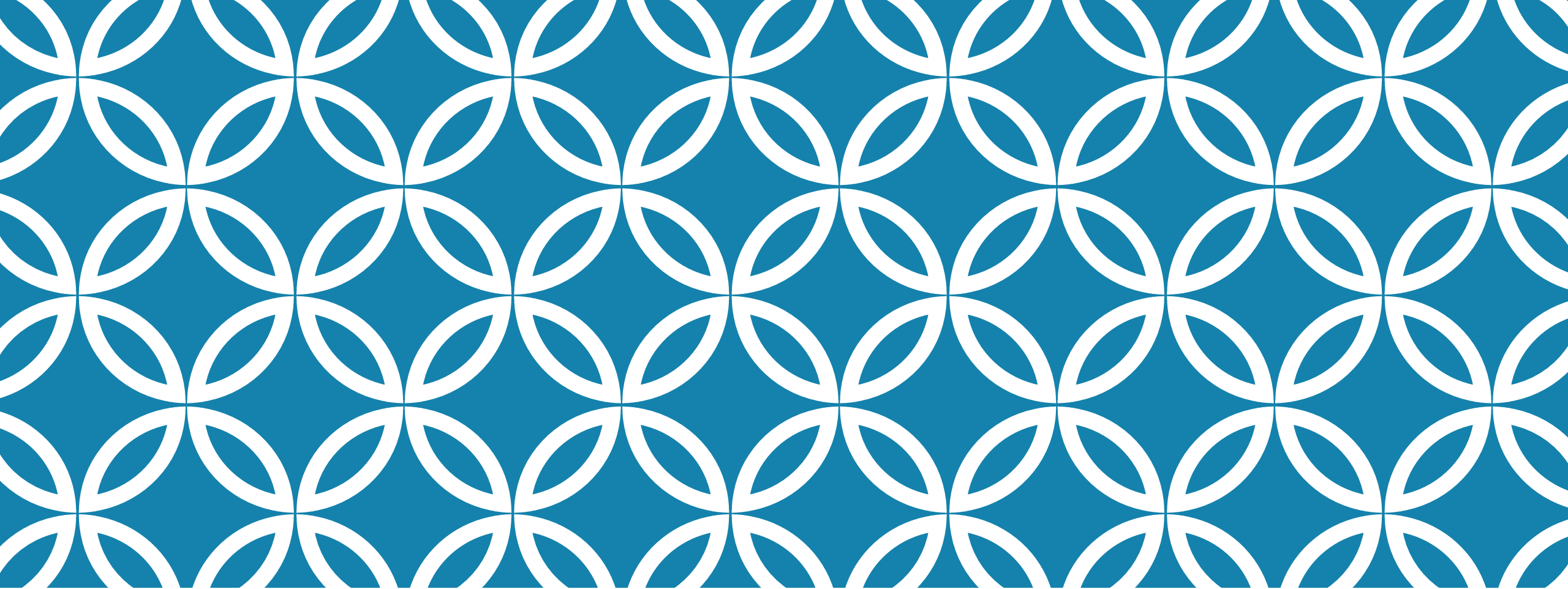


# ACCÈS EN JAVA

## Contrôler les transactions

- *Les instructions de transaction sont accessibles en Java*

```
//On désactive l'auto-commit myConnection.setAutoCommit(false);  
  
//On joue la transaction  
Statement myStatement = myConnection.createStatement();  
//...  
  
//On valide la transaction  
myConnection.commit();  
  
//On aurait pu l'annuler avec 'myConnection.rollback()'
```



**JPA**

# PRÉSENTATION ORM

Permet d'éviter d'écrire du code DML et DDL fastidieux et répétitif  
Surcouche qui permet un accès au SGBD

- Plus conforme à la vision objet

Prend en charge la communication avec le SGBD

Fourni à l'application une *API* de plus haut niveau

# PRÉSENTATION JPA

Java Persistence API

- Repose sur le principe POJO
  - EJB Entité

S'affranchir de la gestion des données SGBD (repose sur un ORM)

EntityManager et EJB Entités

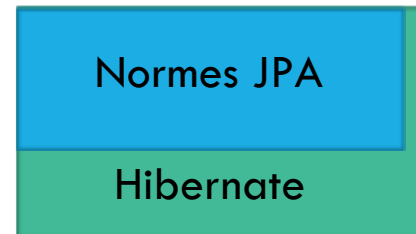
Langage JPA-QL (ou JPQL)

Disponible dans le package *javax.persistence*

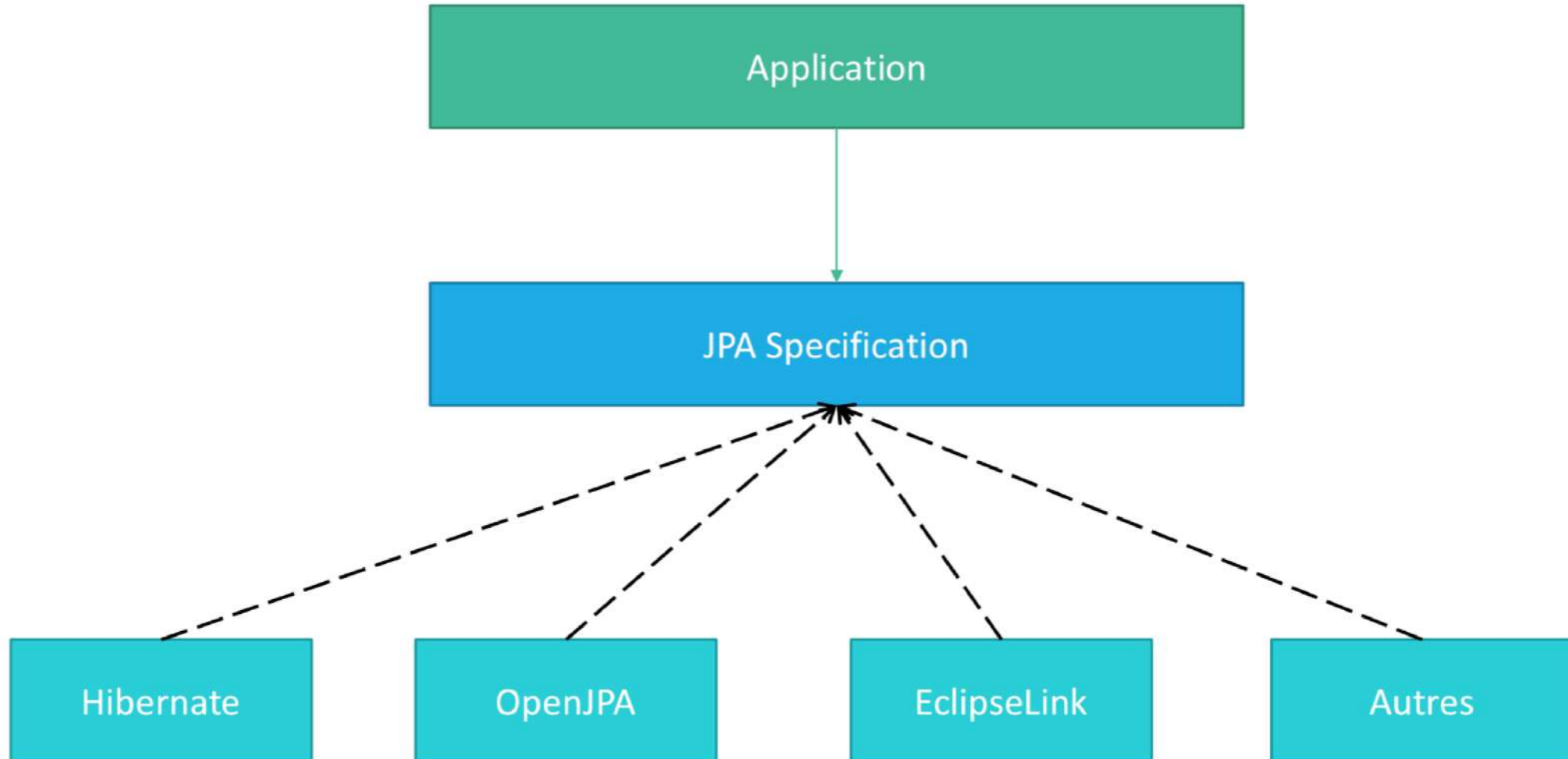
# PRÉSENTATION JPA

Plusieurs implémentations possibles

- Hibernate
- OpenJPA
- Toplink
- DataNucleus
- EclipseLink
- ...



# PRÉSENTATION JPA



# PRÉSENTATION JPA

## Problématique

Modèle objet != Modèle relationnel

Modèle objet	Modèle relationnel
Graphe d'objets	Base de données relationnelle
Instances de classes	Enregistrements dans une table
Références	Relations (FK→PK)
« Clé primaire » optionnelle	
Héritage	

# PRÉSENTATION JPA

Masquer la « plomberie » relationnelle

Les connexions à la base de données ne sont pas visibles Plus d'utilisation du SQL

Mapping par annotations



# PERSISTENCE UNIT

Unité de persistance

- Fichier *main/resources/META-INF/persistence.xml*
- On lui précise le DataSource, le provider et des options
- On lui précise le type de transaction (JTA / RESOURCE\_LOCAL)
- **Attention, chaque implémentation se configure différemment !**

# PERSISTENCE UNIT

## Unité de persistance

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  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_2_0.xsd">
  <persistence-unit name="NomPersistenceUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <class>fr.formation.model.Personne</class>
    <class>fr.formation.model.Fournisseur</class>
    <class>fr.formation.model.Client</class>
    <class>fr.formation.model.Produit</class>

    <properties>
      <!-- Listes des propriétés liées à l'implémentation (Hibernate, OpenJPA, ...) -->
    </properties>
  </persistence-unit>
</persistence>
```

# PERSISTENCE UNIT

```
<persistence-unit name="NomPU" transaction-type="RESOURCE_LOCAL">
<!-- ... -->
  <properties>
    <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/eshop" />
    <property name="hibernate.connection.driver" value="com.mysql.jdbc.Driver" />
    <property name="hibernate.connection.user" value="root" />
    <property name="hibernate.connection.password" value="" />

    <!-- Permet d'exécuter les requêtes DDL pour la génération de la base de données -->
    <!-- Valeurs possibles : validate, update, create, create-drop -->
    <property name="hibernate.hbm2ddl.auto" value="update" />

    <!-- On utilise le moteur innoDB -->
    <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />

    <!-- On imprime les requêtes SQL générées par Hibernate dans la console -->
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.format_sql" value="true" />
  </properties>
</persistence-unit>
```

# PERSISTENCE UNIT

## Type de transaction

- RESOURCE\_LOCAL
  - Utilisation d'un EntityManagerFactory
    - (possible d'injecter un EntityManagerFactory via @PersistenceUnit)
  - Création manuelle des EntityManager (veiller à n'en avoir qu'un seul actif à la fois)
  - Gestion manuelle des transactions
- JTA
  - Non-utilisation d'un EntityManagerFactory
  - Injection d'un EntityManager via @PersistenceContext
  - Transaction gérée par le conteneur

# MAPPING JPA — ENTITÉ (TABLE)

## @Entity

Classe « persistée » dans une table

Chaque instance correspond à un enregistrement

Les attributs correspondent aux colonnes

Les relations sont exprimées avec des annotations (ou fichier de configuration XML)

```
@Entity
public class Produit implements Serializable {
    //...
}
```

# MAPPING JPA — ENTITÉ (TABLE)

## @Table

Spécifie le mapping à la base de données

### Quelques options

name	Nom de la table dans la base de données
indexes	Index (hors clé primaire et clés étrangères) dans la table
uniqueConstraints	Contraintes de clé unique

```
@Entity
@Table(name="produit")
public class Produit implements Serializable {
    //...
}
```

# MAPPING JPA — CLÉ PRIMAIRE

## @Id

Indique que l'attribut est utilisé comme la clé primaire

Obligatoire pour chaque entité

Il faut préciser la stratégie de génération des identifiants (@GeneratedValue)

Par exemple pour une clé auto-incrémentée, la stratégie à utiliser est « IDENTITY »

`@GeneratedValue(strategy=GenerationType.IDENTITY)`

# MAPPING JPA — CLÉ PRIMAIRE

@Id

```
@Entity
@Table(name="personne")
public class Personne implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    //...
}
```



# MAPPING JPA — CLÉ PRIMAIRE

@Id est limité sur un seul champ

S'il y a besoin de spécifier une clé primaire sur deux ou plusieurs champs, il faut utiliser, au choix

@IdClass

@EmbeddedId

# MAPPING JPA — CLÉ PRIMAIRE

## @IdClass

Indique une clé-primaire composée

```
public class AchatId implements Serializable
{
    private int produitId;
    private int clientId;
}
```

```
@Entity
@IdClass(AchatId.class)
public class Achat
{
    @Id private int produitId;
    @Id private int clientId;
}
```

# MAPPING JPA — CLÉ PRIMAIRE

## @EmbeddedId

Indique une clé-primaire composée

```
@Embeddable
public class AchatId implements Serializable
{
    private int produitId;
    private int clientId;
}
```

```
@Entity public class Achat
{
    @EmbeddedId private AchatId id;
}
```

# MAPPING JPA — EXERCICE

Créer un nouveau projet « eshop-model » (MAVEN)

Créer une classe **Personne**

Id, Nom, Prénom

# MAPPING JPA — ATTRIBUTS (CHAMPS)

## @Column

Permet d'indiquer le nom de la colonne de l'attribut dans la table

Utile si le nom de la colonne est différent du nom de l'attribut de l'objet

Utile pour définir la définition de la colonne en base de données

## @Temporal(TemporalType.DATE)

Préciser que la colonne est de type DATE sur les attributs `java.util.Date` ou `java.util.Calendar`

# MAPPING JPA — ATTRIBUTS (CHAMPS)

```
@Entity
@Table(name="produit")
public class Produit implements Serializable
{
    //...
    @Column(name="PRO_LIBELLE", columnDefinition="VARCHAR(50)")
    @NotEmpty
    @Size(max=50)

    private String libelle;
    //...
}
```

# MAPPING JPA – HÉRITAGE

## @Inheritance

Permet d'indiquer l'hérédité entre deux objets (héritage représenté en base de données)

Il faut préciser la stratégie d'héritage

### SINGLE\_TABLE

Une seule table pour toutes les classes

Utilisation d'un champ discriminant

@DiscriminatorColumn sur la classe mère

@DiscriminatorValue sur la classe fille

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_PERSONNE")
```

```
@DiscriminatorValue("Client")
```

### JOINED

Une table par classe

@PrimaryKeyJoinColumn sur la classe fille

```
@PrimaryKeyJoinColumn(name="CLI_ID", referencedColumnName="PER_ID")
```

### TABLE\_PER\_CLASS

Une table par classe contenant toutes les informations (redondance)

# MAPPING JPA — HÉRITAGE

## @Inheritance

```
@Entity
@Table(name="personne")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_PERSONNE")
public class Personne implements Serializable {
    //...
}
```

```
@Entity
@DiscriminatorValue("Client")
public class Client extends Personne {
    //...
}
```



# MAPPING JPA — EXERCICE

Créer une classe **Client** qui hérite de **Personne**

Id, Nom, Prénom, Age, Date de naissance

Créer une classe **Fournisseur** qui hérite de **Personne**

Id, Nom, Prénom, Société

Modifier les classes **Personne**, **Client** et **Fournisseur**

Mapper les colonnes avec JPA

Préciser le nom de la colonne pour les propriétés

Préciser leur définition si besoin

# MAPPING JPA — RELATIONS

@OneToOne

Relation 1:1

@OneToMany

Relation 1:n

List<Object>

@ManyToOne

Relation n:1

Object

@ManyToMany

Relation n:n

# MAPPING JPA — RELATIONS

## Chaque relation a son inverse

@OneToOne / @OneToOne

@OneToMany / @ManyToOne

@ManyToOne / @OneToMany

@ManyToMany / @ManyToMany

# MAPPING JPA — RELATIONS

@JoinColumn (remplace @Column dans ce cas)

Permet de préciser la colonne de jointure

```
@Entity
@Table(name="produit")
public class Produit implements Serializable {
    //...
    @ManyToOne
    @JoinColumn(name="PRO_FOURNISSEUR_ID")
    private Fournisseur fournisseur;
    //...
}
```

# MAPPING JPA — RELATIONS

## @JoinTable

Permet de préciser la table et les colonnes de jointure

```
@Entity
@Table(name="produit")
public class Produit implements Serializable {
    //...

    @ManyToMany
    @JoinTable(
        name="achat",
        uniqueConstraints=@UniqueConstraint(columnNames = { "CMD_PRODUIT_ID", "CMD_CLIENT_ID" }),
        joinColumns=@JoinColumn(name="CMD_PRODUIT_ID", referencedColumnName="PRO_ID"),
        inverseJoinColumns=@JoinColumn(name="CMD_CLIENT_ID", referencedColumnName="CLI_ID"))
    private List<Client> achats;
    //...
}
```

# MAPPING JPA — RELATIONS

## @JoinTable

`name` Précise le nom de la table de jointure

`uniqueConstraints` Précise les colonnes de clé unique

`joinColumns` Précise les informations pour l'entité en cours

`name` Nom de la colonne clé étrangère de l'entité en cours (dans la table de jointure)

`referencedColumnName` Nom de la colonne de référence (table de l'entité en cours)

`inverseJoinColumns` Précise les informations pour l'entité ciblée

`name` Nom de la colonne clé étrangère de l'entité ciblée (dans la table de jointure)

`referencedColumnName` Nom de la colonne de référence (table de l'entité ciblée)

# MAPPING JPA — RELATIONS

Sur les relations inverse, il est possible de préciser la source

Option « mappedBy » dans les annotations @ManyToOne, @ManyToMany et @OneToMany

On précise ici le nom de l'attribut source

```
@Entity
@Table(name="produit")
public class Produit implements Serializable {
    //...
    @ManyToOne
    @JoinColumn(name="PRO_FOURNISSEUR_ID")
    private Fournisseur fournisseur;
    //...
}
```

```
@Entity
@Table(name="fournisseur")
public class Fournisseur extends Personne {
    //...
    @OneToMany(mappedBy="fournisseur")
    private List<Produit> produits;
    //...
}
```

# MAPPING JPA — EXERCICE

Créer une classe **Produit**

Id, Libellé, Prix, Fournisseur

Modifier la classe **Client**

Ajouter une liste de produits (qu'il achète)

Préciser le nom des colonnes et des relations Préciser la table de jointure (« achat »)



# MAPPING JPA — RELATIONS

Aller plus loin dans les relations : ajouter des options aux annotations

## Stratégie de chargement

2 types

Lazy Loading (défaut)

Eager Loading

## Stratégie de cascade

All

Remove

Merge

Persist

Refresh

# MAPPING JPA — RELATIONS

## Stratégie de chargement « Lazy Loading »

- Chargement à la demande

- Les données ne sont chargées que si demandées

- Fonctionne dans un contexte de session

- Ne fonctionne pas en dehors !

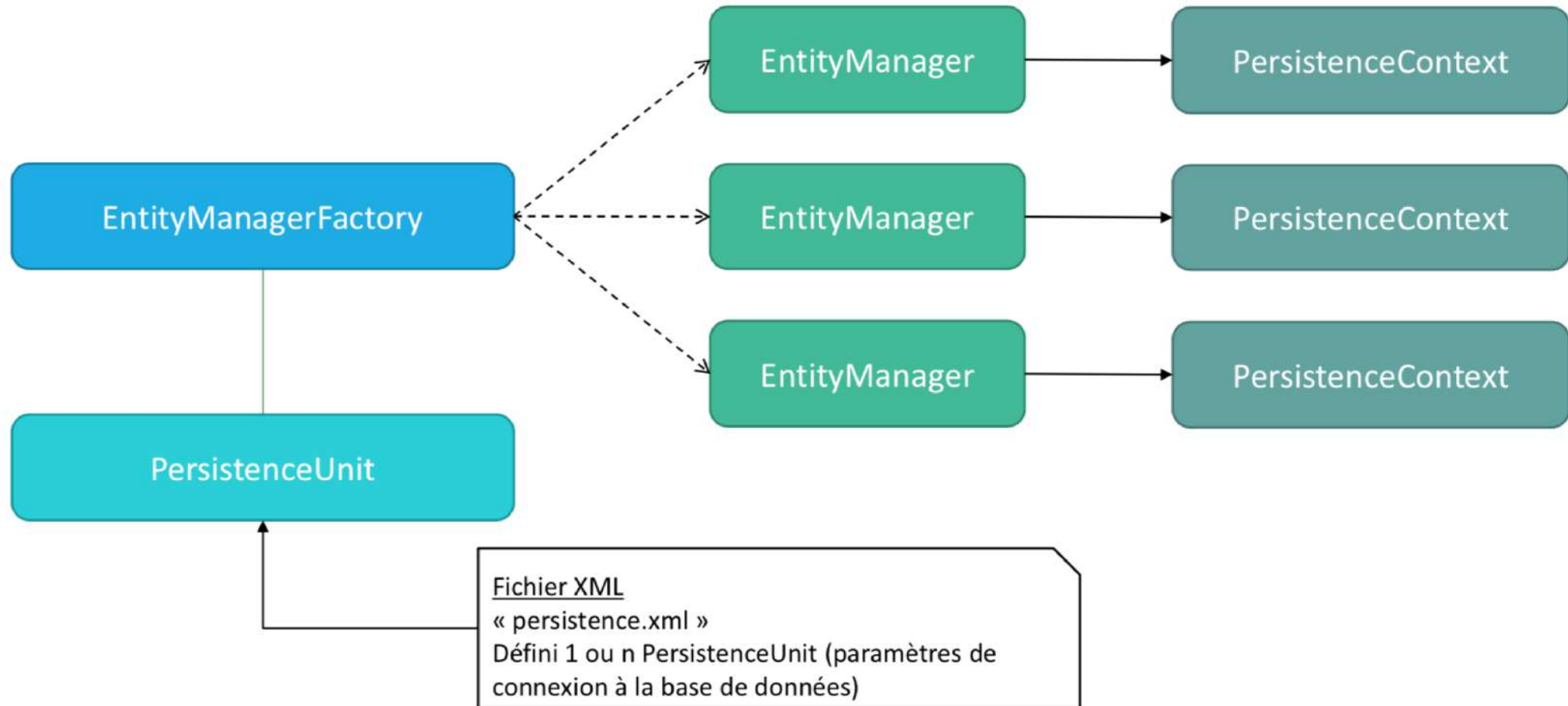
## Stratégie de chargement « Eager Loading »

- Chargement brutal

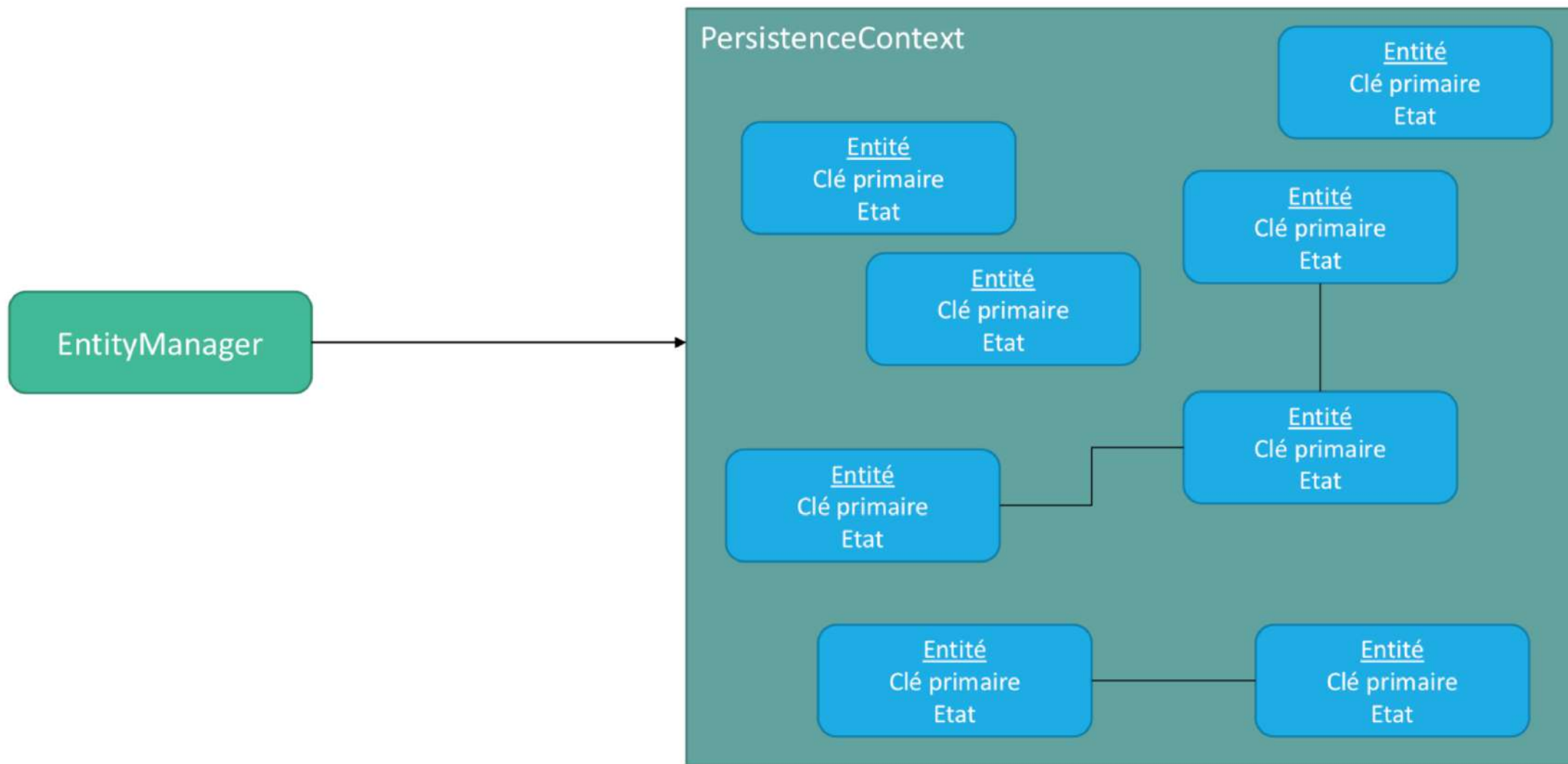
- Les données sont chargées en même temps que l'objet

- Attention, la montée en charge (en mémoire) peut aller très vite, à éviter !

# ENTITY MANAGER



# ENTITY MANAGER



# ENTITY MANAGER

Fait le lien entre les données de la base de données et les objets entités

Opérations essentielles

- persist

- merge

- remove

- find

- createQuery

- ...

Décide quand et comment récupérer les mises à jour (base de données)

Gère l'état des instances dont il a la charge

Ces instances sont dites « Managed »

# ENTITY MANAGER

PERSIST

```
public void add(Produit produit) {  
    em.persist(produit);  
}
```

MERGE

```
public Carte save(Produit produit) { return  
    em.merge(produit);  
}
```

FIND

```
public Carte findById(int id) { return  
    em.find(Produit.class, id);  
}
```

REMOVE

```
public void delete(Produit produit) {  
    em.remove(produit);  
}
```

# ENTITY MANAGER

## L'opération « merge »

Crée une copie de l'entité passée en paramètre

Il y a donc deux instances différentes de cette même entité

Une non-managée par EntityManager (celle passée en paramètre de la méthode *merge()*)

Une managée par EntityManager (celle retournée par la méthode *merge()*)

Pour cette raison, on utilise *merge()* pour sauvegarder et supprimer

Utiliser merge lors de la suppression garanti sa gestion par l'Entity Manager

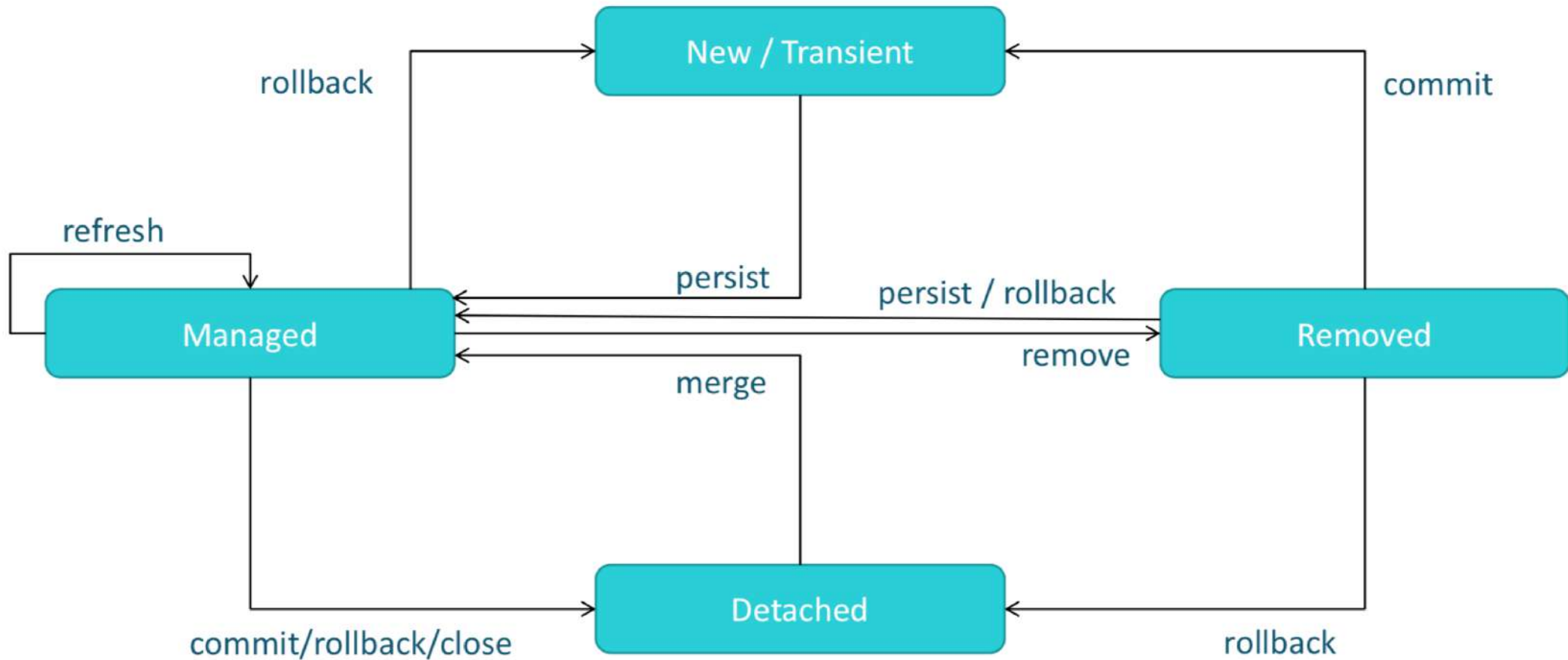
Il ne peut pas supprimer un objet qu'il ne gère pas

# ENTITY MANAGER

Etat de l'instance	Comportement
New (ou Transient)	Non géré
Managed	Géré
Removed	Supprimé (suppression logique)
Detached	Détaché (plus géré)



# ENTITY MANAGER



# ENTITY MANAGER

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("NomPersistenceUnit");
EntityManager em = emf.createEntityManager();

List<Produit> myProduits = em.createQuery("select p from Produit p", Produit.class).getResultList();
//...

//On oublie pas de fermer EntityManager et EntityManagerFactory
em.close();
emf.close();
```

# ENTITY MANAGER

## Attention

Il faut **gérer la transaction** ! (voir extrait ci-dessous)

```
EntityManager tx = em.getTransaction(); //Récupérer la transaction  
  
tx.begin(); //Démarrer la transaction  
  
tx.commit(); //Appliquer les traitements en base de données  
  
tx.rollback(); //Annuler les traitements
```

# PATTERN DAO

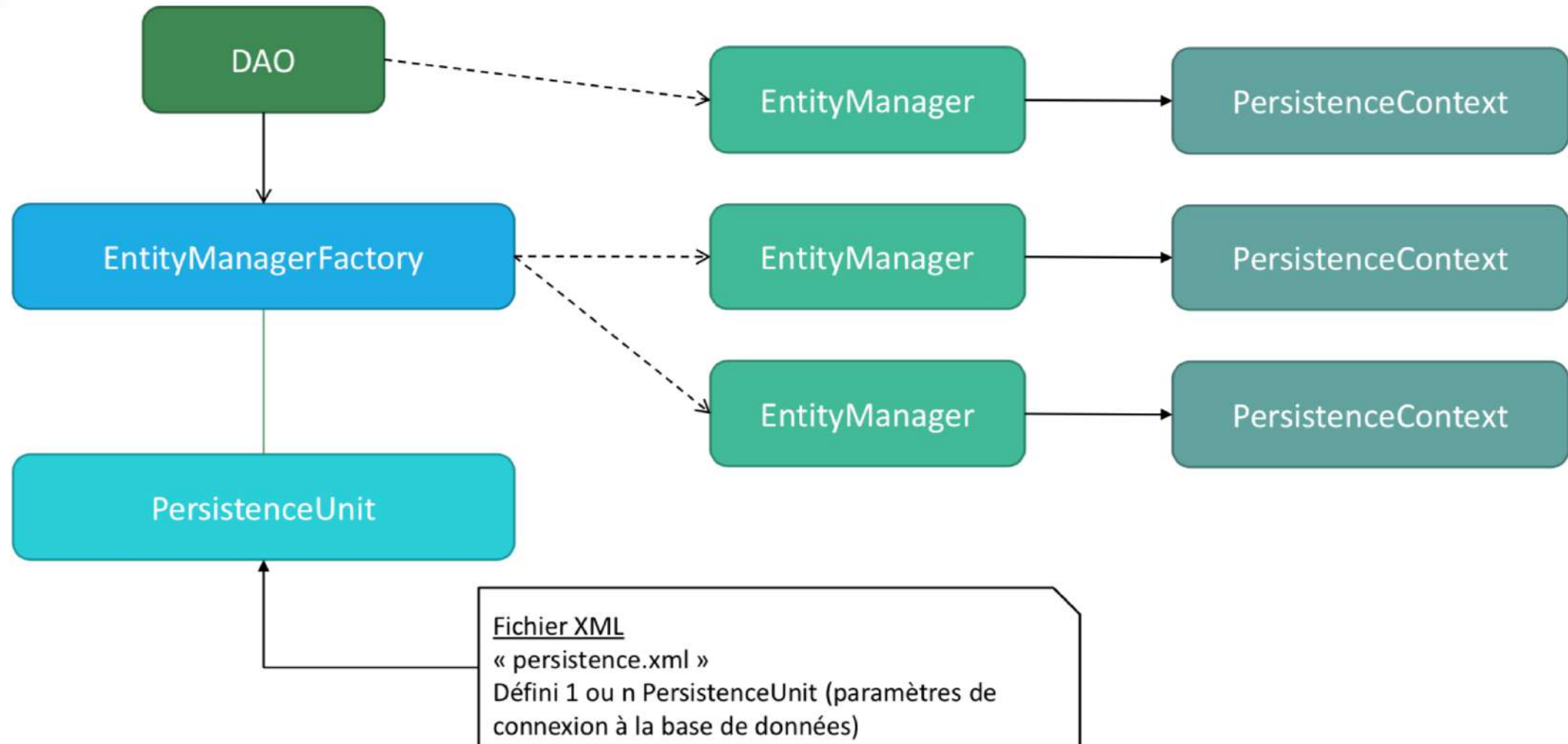
Data Access Object

Chaque DAO spécialisée a la responsabilité de traiter les données (**CRUD**)

C'est donc lui, et lui seul, qui accède à l'*EntityManager* et qui le manipule

```
public interface IDAO<T> {  
    public T findById(int id);  
    public List<T> findAll();  
    public T save(T entity);  
    public boolean delete(T entity);  
}
```

# PATTERN DAO



# PATTERN DAO — EXERCICE

## Créer une DAO **DAOClient**

Trouver, ajouter, modifier, supprimer un client

## Créer une DAO **ProduitDAO**

Trouver, ajouter, modifier, supprimer un produit

Manipuler ces DAO depuis le programme principal ? Ajouter un nouveau client

Lister les produits dans la console

# JPA-QL (JPQL)

Langage inspiré du SQL

Pensé avec le paradigme objet

```
public Produit findByLibelle(String libelleProduit) {  
    Query myQuery = em.createQuery("from Produit p where p.libelle = :lelibelle", Produit.class);  
  
    //On insère les paramètres  
    myQuery.setParameter("lelibelle", libelleProduit);  
  
    return myQuery.getSingleResult();  
}
```

```
public List<Produit> getProduits () {  
    Query myQuery = em.createQuery("from Produit", Produit.class);  
    return myQuery.getResultList();  
}
```

# JPA-QL (JPQL)

## Chargement Lazy Loading

```
public Produit findWithAchats(int id) {  
    Query myQuery = em.createQuery("from Produit p left join fetch p.achats a where p.id = :id", Produit.class);  
    //...  
}
```



# REQUÊTES NOMMÉES

Configuration par annotation sur l'entité

@NamedQueries

Liste des requêtes nommées

@NamedQuery

Une requête nommée

name

Nom de la requête nommée

query

Requête JPQL

```
@NamedQueries({  
    @NamedQuery(  
        name="Produit.findByLibelle",  
        query="select p from Produit p where p.libelle = :lelibelle")  
    })
```

```
Query myQuery = em.createNamedQuery("Produit.findByLibelle", Produit.class);
```

# REQUÊTES NATIVES

JPQL est le langage recommandé

Mais il se peut qu'on ait besoin d'utiliser une spécificité SQL ou structurelle

Dans ce cas, on peut utiliser les requêtes natives

```
Query myQuery = em.createNativeQuery(  
    "SELECT * FROM produit WHERE PRO_LIBELLE = :lelibelle", Produit.class);
```