

Développement Java

# JDBC & JAVA : JPA



# Objectifs pédagogiques

- Utiliser l'API JPA au sein d'une application connectée à une base de données SQL.



# C'est quoi JPA ?

- **Java Persistence API**
- Description d'un mécanisme pour gérer la correspondance entre des objets Java et une base de données
- Standard faisant partie intégrante de la plateforme Java EE



# Les implémentations de JPA

- Hibernate (<http://www.hibernate.org/>)
- EclipseLink (<https://www.eclipse.org/eclipselink/>)
- OpenJPA (<http://openjpa.apache.org/>)
- TopLink (<https://oss.oracle.com/toplink-essentials-jpa.html>)



# Petite histoire...avant JPA

- Historiquement, Hibernate et TopLink sont apparus bien avant la création de JPA
- Java a introduit JPA 1.0 en 2006, JPA 2.0 en 2009 puis JPA 2.1 en 2013
- Il standardise les solutions de gestion de persistance

# ORM et JPA

- **Object-Relational Mapping**
- **Un ORM** est Système permettant de faire des correspondances entre les schémas de la base de données et les classes d'un programme
- **JPA** masque le moyen de stockage au développeur, en lui permettant de travailler uniquement sur un **modèle objet**

# Utilisation des EJB

- Enterprise JavaBean
- Arrivés en 1998 mais réellement utilisable en 2006 (EJB 3.0)
- Les « *Javabeans Entity* » seront nos classes Java définissant des correspondances entre les attributs et les champs des tables relationnelles de la base de données
- Ces correspondances sont le **Mapping Relationnel/Objet (ORM)**
- Des annotations JPA définissent les relations

# Exemple d'une entité JPA (1/2)

- Une entité JPA est une classe implémentant *Serializable*, avec des attributs privés, des *getters et setters* et un constructeur par défaut

```
public class Contact implements Serializable {  
  
    private int id;  
    private String name;  
    private String firstname;  
  
    public Contact(){ /** default constructor */ }  
  
    // ... setters and getters ...  
}
```



# Exemple d'une entité JPA (2/2)

- ...avec des annotations JPA

```
@Entity
public class Contact implements Serializable {

    @Id
    private int id;
    private String name;
    private String firstname;

    public Contact(){ /** default constructor */ }

    // ... setters and getters ...
}
```

# Les annotations JPA

- <https://docs.oracle.com/javaee/6/api/index.html?javax/persistence>
- **@Entity**
  - Précise que la classe est une entité JPA
- **@Table(name="CONTACTS")**
  - Précise que le nom de la table pour laquelle la classe est mappée. Sans elle, la table porterait le nom de la classe

```
@Entity
@Table(name="CONTACTS")
public class Contact {
}
```

# Les annotations JPA

- **@Id**
  - Précise que l'attribut est la clé primaire de la base de données
- **@GeneratedValue(strategy=GenerationType.XXXXXXXXXX)**
  - Précise la manière dont la clé primaire est générée
- **GenerationType** est une énumération de constantes :
  - GenerationType.AUTO
  - GenerationType.IDENTITY
  - GenerationType.SEQUENCE
  - GenerationType.TABLE



# Les annotations JPA

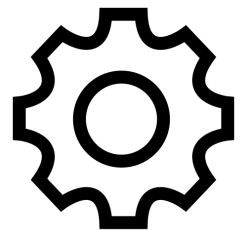
- **GenerationType.AUTO**

- La génération de la clé primaire est laissée à l'implémentation, elle crée une séquence unique sur tout le schéma via une table spécifique

- **GenerationType.IDENTITY**

- La génération de la clé primaire se fera à partir d'une Identité propre au SGBD. Il utilise un type de colonne spéciale à la base de données.

Pour MySQL, il s'agit d'un **AUTO\_INCREMENT**



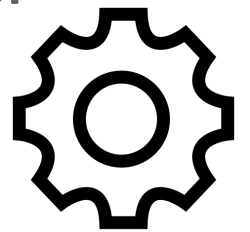
# Les annotations JPA

- **GenerationType.TABLE**

- La génération de la clé primaire se fera en utilisant une table dédiée qui stocke les noms et les valeurs des séquences. Cette stratégie doit être utilisée avec une autre annotation qui est **@TableGenerator**

- **GenerationType.SEQUENCE**

- La génération de la clé primaire se fera par une séquence définie dans le SGBD, auquel on ajoutera l'attribut generator. Cette stratégie doit être utilisée avec une autre annotation qui est **@SequenceGenerator**



# Les annotations JPA

- **@Column("password")**
  - Précise que l'attribut sera stocké dans une colonne dont le nom est la chaîne de caractères, ici *password*

```
@Entity
@Table(name="CONTACTS")
public class Contact implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    @Column(name="PASS_WORD")
    private String password;
```

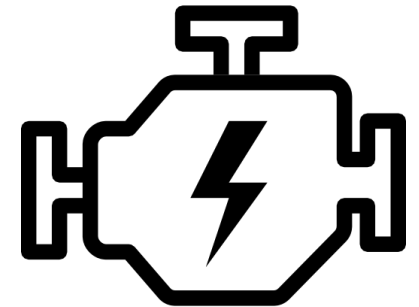
# Autres annotations JPA

- Il y a bien d'autres annotations :

Annotations	Rôles
@Basic	Annotation par défaut si aucune n'est spécifiée
@Transient	Précise que la valeur de l'attribut ne sera pas persistée
@Lob	Précise que la valeur de l'attribut sera persisté en tant que <b>BLOB</b> ( <b>B</b> inary <b>L</b> arge <b>O</b> bject)
@Temporal	Précise que la valeur sera une date ou une heure
@Enumerated	Précise que la valeur est une énumération

# Le « *Persistence Unit* »

- Élément clé de la technologie JPA
- *Persiste* les entités dans la base de données
- Nécessite un fournisseur de persistance (Hibernate)
- Nécessite une configuration (persistence.xml)





# Le « *Persistence Unit* » 1/2

- La configuration (persistence.xml)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
  version="3.0">

  <persistence-unit name="persistence-unit">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
      <property name="jakarta.persistence.jdbc.url"
        value="jdbc:mysql://localhost:8889/myDatabase?serverTimezone=UTC"/>
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.password" value="root"/>
      <property name="hibernate.hbm2ddl.auto" value="create"/>
    </properties>
  </persistence-unit>

</persistence>
```

# Le « Persistence Unit » 2/2

```
<persistence-unit name="persistence-unit">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <class>fr.sauvageb.demo.User</class>
  <exclude-unlisted-classes>true</exclude-unlisted-classes>
  <properties>

    <property name="jakarta.persistence.jdbc.url"
      value="jdbc:mysql://localhost:8889/myDatabase?serverTimezone=UTC&createDatabaseIfNotExist=true"/>

    <!-- Affiche les requêtes SQL dans la console -->
    <property name="hibernate.show_sql" value="true"/>
    <!-- Affiche les requêtes SQL dans la console -->
    <property name="hibernate.format_sql" value="true"/>
    <!-- Stratégie pour la génération du schéma de la base données -->
    <!-- validate : valide le schéma de la base de données en correspondance avec les entités -->
    <!-- update : met à jour le schéma en comparant les entités et le schéma actuel de la base de données -->
    <!-- create : supprime le schéma de la base de données puis utilise les entités pour le recréer -->
    <!-- create-drop : effectue l'option create à la fermeture de l'EntityManagerFactory -->
    <property name="hibernate.hbm2ddl.auto" value="create"/>

    <!--      <property name="jakarta.persistence.schema-generation.database.action" value="drop-and-create"/>-->
    <!--      <property name="jakarta.persistence.schema-generation.create-script-source" value="create-schema.sql" />-->
    <!--      <property name="jakarta.persistence.sql-load-script-source" value="load-data.sql" />-->
    <!--      <property name="jakarta.persistence.schema-generation.drop-script-source" value="drop-schema.sql" />-->
  </properties>
```

# Création du persistence.xml 1/2

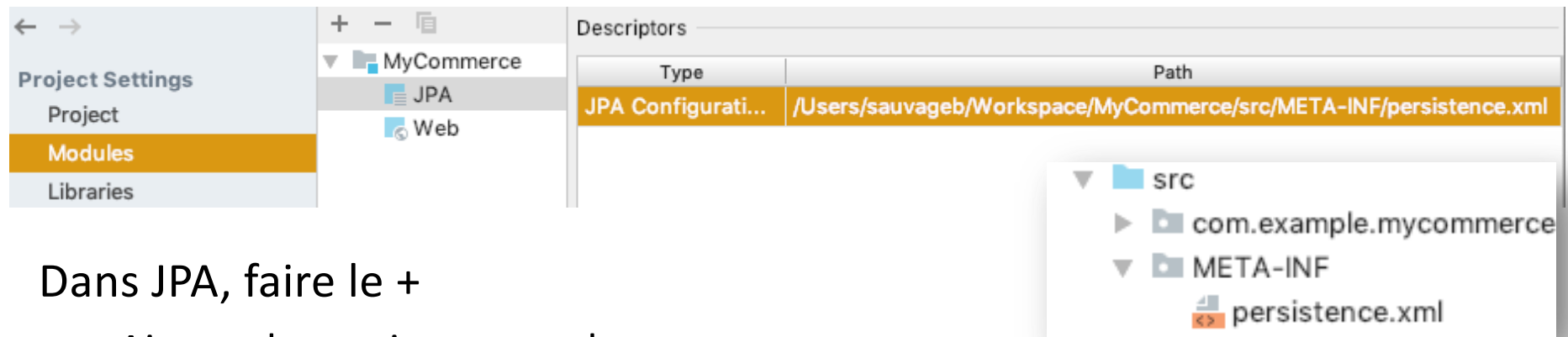
## Sur un projet utilisant Maven :

- Cliquez-droit sur votre projet -> Open module Settings
- Dans modules, faire le +
  - Ajouter le module JPA
- Dans JPA, faire le +
  - Ajouter le persistence.xml
  - Choisir dans ressources/META-INF/persistence.xml

# Création du persistence.xml 2/2

## Sur un projet n'utilisant pas Maven :

- Cliquez-droit sur votre projet -> Open module Settings
- Dans modules, faire le +
  - Ajouter le module JPA



- Dans JPA, faire le +
  - Ajouter le persistence.xml
  - Choisir dans resources/META-INF/persistence.xml

# Le pilote JDBC

- Chaque base de données fournit un pilote JDBC pour y accéder via Java
- En fonction de la base de données utilisée, le fichier JAR approprié devrait être mis dans les bibliothèques

**Exemple :** pour la base de données MySQL

- <https://www.mysql.com/fr/products/connector/> (sans Maven)
- <https://mvnrepository.com/artifact/com.mysql/mysql-connector-j>



# Le « EntityManager »

- L'objet **EntityManager** permet de gérer toutes les opérations sur les entités (CRUD)
- **Aucun code SQL** n'est requis, nous manipulons les objets Java directement :

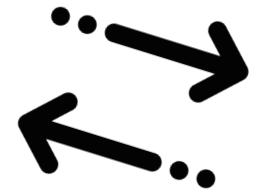
```
Country c = new Country("France");  
EntityManager em = ...  
em.persist(c);  
em.close();
```

# Le « EntityManager » et ses méthodes

- L'objet **EntityManager** fournit des méthodes (CRUD) :

Méthodes	Rôles
void <b>persist</b> (Object entity)	<b>C</b> REATE
T find(Class<T> entityClass, Object primaryKey)	<b>R</b> EAD
Object <b>merge</b> (Object entity)	<b>U</b> PGDATE
void remove(Object entity)	<b>D</b> ELETE

- Plus besoin d'écrire les requêtes du CRUD



# Récupérer le « EntityManager »

- Utiliser l'objet **EntityManagerFactory**

```
EntityManagerFactory emf =  
Persistence.createEntityManagerFactory("My-PU");  
  
EntityManager em = emf.createEntityManager();  
Contact contact = em.find(Contact.class, 1);  
em.close();  
emf.close();
```

- "My-PU" est la clé *name* dans le *persistence.xml*



# Bonnes pratiques

- Utiliser l'objet **EntityManagerFactory**
  - Une seule instance dans toute votre application
  - Fermer l'instance à l'arrêt de l'application
- Utiliser l'objet **EntityManager**
  - Une nouvelle instance pour chaque utilisation
  - Fermer l'instance après chaque utilisation

## Les raisons :

EntityManager n'est pas thread-safe contrairement à EntityManagerFactory

# Les entités JPA et leurs états 1/4

Avec JPA, les entités peuvent avoir différents états :

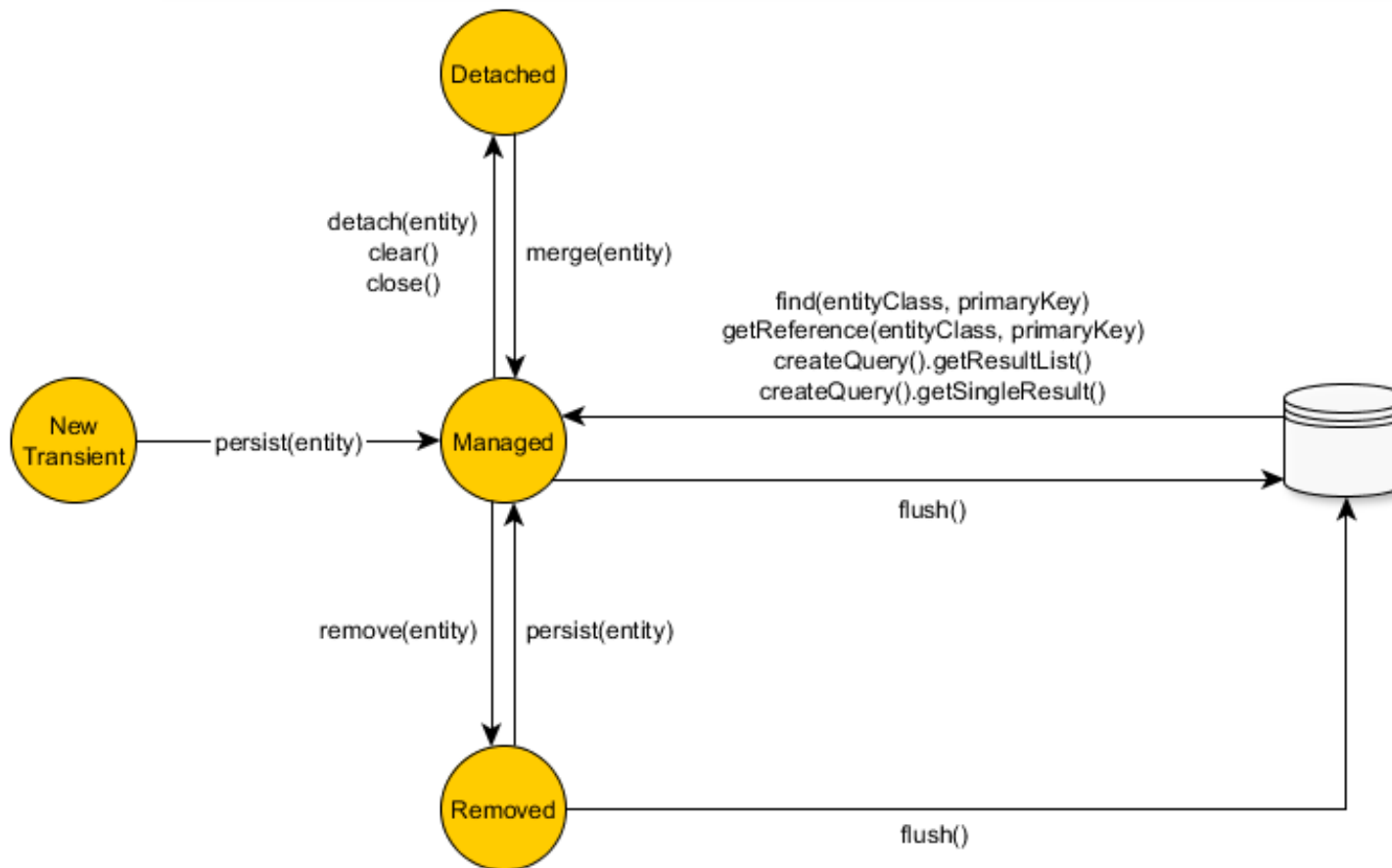
- **Transient**
  - Une entité est dans un état transitoire lorsqu'elle vient d'être créée en mémoire mais n'a pas encore été associée à une unité de persistance (EntityManager).
  - Les entités transitoires ne sont pas gérées par l'EntityManager et ne sont pas encore synchronisées avec la base de données.
  - Elles ne sont pas encore persistantes et ne sont pas encore suivies par JPA.
- **Managed**
  - Une entité est dans un état géré lorsque vous l'avez associée à un EntityManager en utilisant **persist()** ou **merge()**, ou lorsque vous l'avez récupérée via l'EntityManager
  - Les entités gérées sont suivies par JPA, et tout changement apporté à ces entités est automatiquement propagé à la base de données lorsque la transaction est validée.

# Les entités JPA et leurs états 2/4

- Detached
  - Une entité est dans un état détaché lorsque vous l'avez dissociée de l'EntityManager, généralement après la fin d'une transaction ou en utilisant la méthode **detach()**
  - Les entités détachées ne sont plus suivies par JPA, et les modifications apportées à ces entités ne sont pas automatiquement synchronisées avec la base de données.
  - Vous pouvez réattacher une entité à un EntityManager pour la gérer à nouveau
- Removed
  - Une entité est dans un état supprimé lorsqu'elle a été supprimée de la BDD en utilisant la méthode `remove()` de l'EntityManager.
  - Les entités supprimées restent dans l'unité de persistance, mais elles sont marquées pour suppression lors de la prochaine validation de la transaction.
  - Une fois la transaction validée, l'entité est supprimée de la BDD et devient détachée

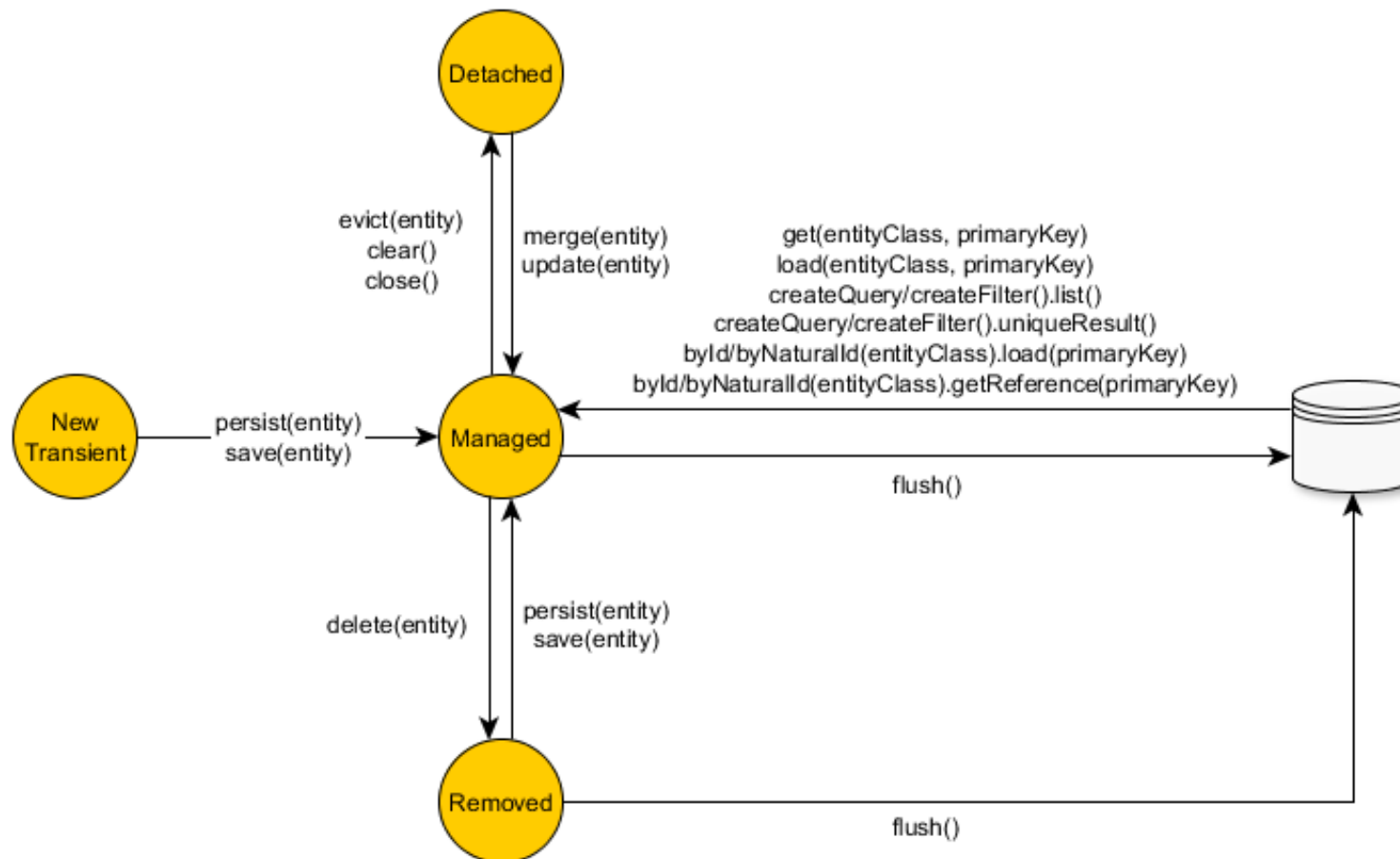
# Les entités JPA et leurs états 3/4

- Avec JPA, les entités peuvent avoir différents états :



# Les entités JPA et leurs états 4/4

- Avec certaines implémentations spécifiques, cela varie :



# Quizz

Questions	Réponses
Quelle annotation permet de déclarer une entité JPA ?	@Entity
Quelle annotation permet de déclarer une clé primaire ?	@Id
Comment déclarer la connexion avec la base de données ?	En créant un persistence.xml décrivant le Persistence Unit
Quelle est l'utilité de l'EntityManager ?	Utiliser le CRUD sur les entités


# JPA et les Transactions (1/3)

- **Une transaction** est généralement utilisée pour combiner plusieurs écritures sur une base de données en une seule opération atomique
- Les opérations de lecture ne nécessitent pas de transaction
- **Par défaut**, les transactions doivent être validées explicitement
  - Nous devons gérer les transactions manuellement


# JPA et les Transactions (2/3)

```
EntityManager em = this.emf.createEntityManager();
EntityTransaction et = em.getTransaction();
et.begin();
try {
    Category category = em.find(Category.class, 1);
    em.remove(category);
    et.commit();
} catch (RuntimeException re) {
    if (et.isActive())
        et.rollback();
} finally {
    em.close();
}
```

Validation des  
requêtes dans la  
base de données



Annulation de la  
transaction dans le  
cas où celle-ci se  
passe pas





# JPA et les Transactions (3/3)


- Il est également possible de ne pas gérer les transactions
  - Définition de la valeur **autocommit** à **true**

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">

  <persistence-unit name="PU" transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
    ...
      <property name="hibernate.connection.autocommit" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```



# Exercice (1/3) : Blog

- Ajouter une implémentation JPA à votre projet (Hibernate)
- Ajouter le pilote JDBC MySQL
- Créer une classe JavaBean **Post**
  - Mettre la classe dans le sous-package **dao.entity**
  - Le post est représenté par une date de création, un nom, une description, etc
- Convertir cette classe en entité JPA
  - La table s'appellera **posts**
  - L'id sera la clé primaire



# Exercice (2/3) : Blog

- **Afin de tester votre code**
  - Enregistrer 3 posts en base de données
  - Vérifiez que les posts sont présents en base de données



# Exercice (3/3) : Blog

- **Mettre en place un menu interactif**
  - Utilisez le scanner pour saisir les informations d'un post
  - Permettre l'ajout de plusieurs posts



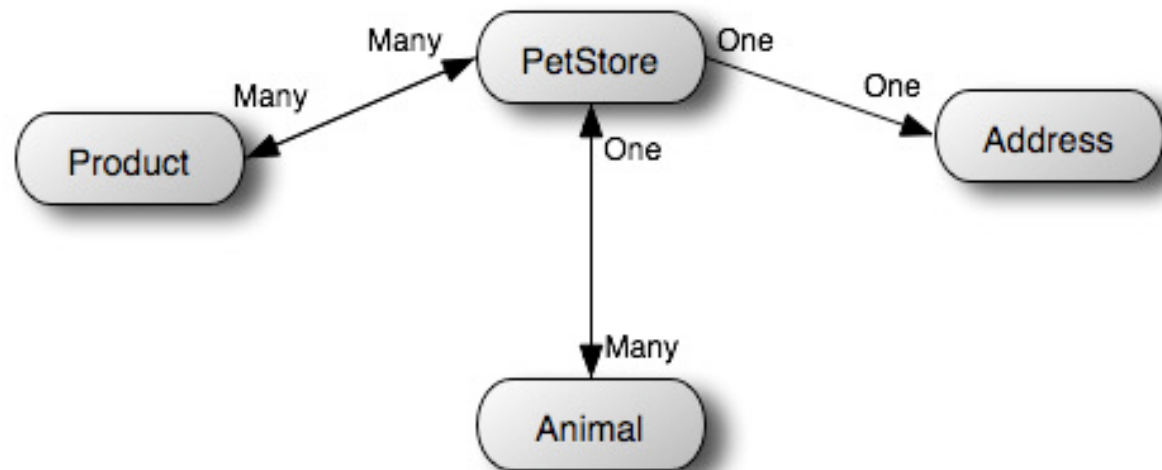
Java Persistence API

# JPA – LES FONCTIONS AVANCÉS

*Dépendances entre les entités, héritage*

# Relations entre les entités

- Les entités ont souvent des relations entre elles :
  - One-To-One
  - One-To-Many
  - Many-To-One
  - Many-To-Many



# Relations entre les entités

- Relations entre les entités sont décrites avec des annotations au dessus des attributs
- Différentes stratégies sont possibles
- JPA gère également la notion d'héritage entre les entités avec des annotations



# One-To-One

- L'annotation **@OneToOne** décrit une relation one-to-one entre deux entités
- 3 Stratégies possibles:
  - **@JoinColumn**
    - Colonne supplémentaire contenant une foreign key
  - **@PrimaryKeyJoinColumn**
    - 2 entités indépendante ont la même primary key
  - **@JoinTable**
    - Une table supplémentaire avec jointure





# One-To-One : @JoinColumn

## @JoinColumn

- Cette stratégie est couramment utilisée pour les relations @OneToOne
- Elle ajoute une colonne de clé étrangère (foreign key) dans la table de l'entité propriétaire pour établir la relation
- Exemple : Une table "Person" contient une colonne "address\_id" qui fait référence à l'ID de l'adresse de la personne

# One-To-One : @PrimaryKeyJoinColumn

## @PrimaryKeyJoinColumn

- Cette stratégie implique que les 2 entités partagent la même clé primaire
- Signifie que l'entité propriétaire a la même clé primaire que l'entité associée
- Moins courante que **@JoinColumn**, mais utilisée lorsque les 2 entités ont une relation étroite et partagent une clé primaire commune

**Exemple :** Une table "Person" et une table "PersonDetails" partagent la même clé primaire.

# One-To-One : @JoinTable

## @JoinTable

- Stratégie plus courante pour les relations many-to-many, mais peut être utilisée pour les relations @OneToOne (table de jointure explicite)
- Crée une table de jointure distincte contenant les clés étrangères des 2 entités pour établir la relation
- Utile pour gérer des attributs supplémentaires dans la relation

**Exemple :** Une table "Person" et une table "Passport" sont associées par une table de jointure "PersonPassport" avec des colonnes supplémentaires, comme la date de délivrance

# One-To-One

- Par exemple, une entité *Store* a une seule adresse :

```
public class Store {  
    ...  
    @OneToOne  
    @JoinColumn(name="address_fk")  
    private Address address;  
    ...  
}
```

- Dans la table *store*, une foreign key est utilisée

# One-To-Many and Many-To-One 1/3

- Les annotations **@OneToMany** et **@ManyToOne** lient une entité à une collection d'une autre entité
- Exemple :
  - Une personne a plusieurs comptes bancaires, et chacun des comptes ont un propriétaire unique
- Stratégies possibles :
  - **@JoinTable** (Table intermédiaire de jointure)
  - **@JoinColumn** (Ajout d'une colonne FK)

# One-To-Many and Many-To-One 2/3

## @JoinColumn

```
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToMany(mappedBy = "proprietaire")
    private List<CompteBancaire> comptes;

    // getters et setters

}
```

```
@Entity
public class CompteBancaire {

    @Id
    @GeneratedValue(strategy = ...)
    private Long id;

    private String numero;

    @ManyToOne
    @JoinColumn(name = "proprietaire_id")
    private Personne proprietaire;

    // getters et setters

}
```

# One-To-Many and Many-To-One 3/3

## @JoinTable

```
@Entity
public class Personne {

    @OneToMany
    @JoinTable(
        name = "personne_compte",
        joinColumns = @JoinColumn(name = "personne_id"),
        inverseJoinColumns = @JoinColumn(name =
"compte_id")
    )
    private List<CompteBancaire> comptes;

}
```

```
@Entity
public class CompteBancaire {

    @Id
    @GeneratedValue(strategy = ...)
    private Long id;

    private String numero;

    @ManyToOne
    private Personne proprietaire;

    // getters et setters
}
```

# Many-To-Many 1/2

- L'annotation **@ManyToMany** associe 2 entités entre elles

## Exemple :

- Un produit peut avoir plusieurs catégories et une catégorie peut avoir plusieurs produits
- Stratégie possible :
  - **@JoinTable** est la seule option



# Many-To-Many 2/2

## @JoinTable

```
@Entity
public class Personne {

    @ManyToMany
    @JoinTable(
        name = "personne_compte",
        joinColumns = @JoinColumn(name = "personne_id"),
        inverseJoinColumns = @JoinColumn(name =
"compte_id")
    )
    private List<CompteBancaire> comptes;

}
```

```
@Entity
public class CompteBancaire {

    @Id
    @GeneratedValue(strategy = ...)
    private Long id;

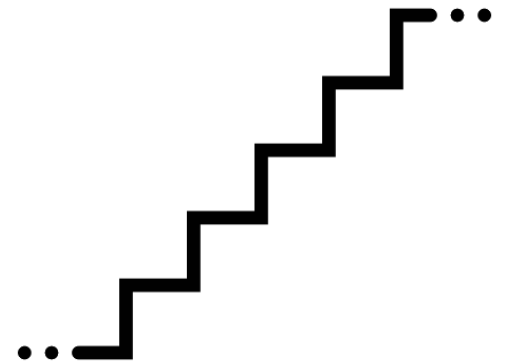
    private String numero;

    @ManyToMany(mappedBy= "comptes")
    private List<Personne> proprietaires;

    // getters et setters
}
```

# La Cascade des relations

- Ces différentes annotations de relations possèdent une propriété cascade
- Une opération appliquée à une entité est répercutée sur les entités dépendantes
- Il existe 4 types de cascade :
  - `PERSIST|MERGE|REMOVE|REFRESH`
  - `CascadeType.ALL` : 4 combinés



# Cascade PERSIST 1/2

- Lorsque vous persistez (ajoutez) une entité parente, les entités enfants associées seront également persistées
- Si vous ajoutez une nouvelle entité parente, les entités enfants liées seront automatiquement enregistrées dans la base de données

```
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToMany(mappedBy = "proprietaire", cascade = CascadeType.PERSIST)
    private List<CompteBancaire> comptes;
```

```
@Entity
public class CompteBancaire {

    @Id
    @GeneratedValue(strategy = ...)
    private Long id;

    private String numero;

    @ManyToOne
    @JoinColumn(name = "proprietaire_id")
    private Personne proprietaire;
```

## Cascade PERSIST 2/2

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-pu");  
EntityManager em = emf.createEntityManager();
```

```
// Création d'une nouvelle personne et d'un compte bancaire associé
```

```
Personne personne = new Personne("Boris SAUVAGE");  
CompteBancaire compteBancaire = new CompteBancaire("12345");
```

```
personne.getComptes().add(compteBancaire);  
compteBancaire.setProprietaire(personne);
```

```
// La cascade PERSIST permet de persister automatiquement le compte bancaire
```

```
em.getTransaction().begin();  
em.persist(personne);  
em.getTransaction().commit();
```

```
em.close();  
emf.close();
```

# Cascade MERGE 1/1

- Lorsque vous fusionnez une entité parente, les entités enfants associées seront également fusionnées
- S'applique généralement lors de la mise à jour d'une entité parente, les entités enfants sont également mises à jour

```
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToMany(mappedBy = "proprietaire", cascade = CascadeType.MERGE)
    private List<CompteBancaire> comptes;
```

```
@Entity
public class CompteBancaire {

    @Id
    @GeneratedValue(strategy = ...)
    private Long id;

    private String numero;

    @ManyToOne
    @JoinColumn(name = "proprietaire_id")
    private Personne proprietaire;
```

# Cascade REMOVE 1/1

- Lorsque vous supprimez une entité parente, les entités enfants associées seront également supprimées
- Garantit que la suppression d'une entité parente entraînera également la suppression de ses entités enfants

```
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToMany(mappedBy = "proprietaire", cascade = CascadeType.REMOVE)
    private List<CompteBancaire> comptes;
```

```
@Entity
public class CompteBancaire {
    @Id
    @GeneratedValue(strategy = ...)
    private Long id;

    private String numero;

    @ManyToOne
    @JoinColumn(name = "proprietaire_id")
    private Personne proprietaire;
```

# Cascade REFRESH 1/1

- Cette cascade signifie que lorsque vous actualisez une entité parente, les entités enfants associées seront également actualisées

```
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToMany(mappedBy = "proprietaire", cascade = CascadeType.REFRESH)
    private List<CompteBancaire> comptes;
```

```
@Entity
public class CompteBancaire {
    @Id
    @GeneratedValue(strategy = ...)
    private Long id;

    private String numero;

    @ManyToOne
    @JoinColumn(name = "proprietaire_id")
    private Personne proprietaire;
```

# Cascade ALL 1/1

- Combinaison de tous les types de cascade, ce qui signifie que toutes les opérations (PERSIST, MERGE, REMOVE, REFRESH) sont répercutées sur les entités enfants

```
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToMany(mappedBy = "proprietaire", cascade = CascadeType.ALL)
    private List<CompteBancaire> comptes;
```

```
@Entity
public class CompteBancaire {
    @Id
    @GeneratedValue(strategy = ...)
    private Long id;

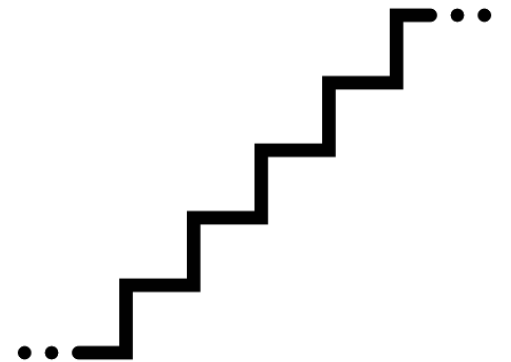
    private String numero;

    @ManyToOne
    @JoinColumn(name = "proprietaire_id")
    private Personne proprietaire;
```



# Concernant le type de Cascade

- `CascadeType.XXX`
- Lorsqu'une instance *Store* est XXX, l'opération est effectuée en cascade à l'instance de l'*Address* référencée, qui est également XXX
- La cascade peut continuer de manière récursive
  - Par exemple, pour toutes les entités référencées par l'objet *Address*



# Le Lazy loading 1/2

- Ces différentes annotations de relations possèdent une propriété *fetch*
  - Il existe 2 Types : **LAZY** et **EAGER**
- Quand une entité est récupérée, les attributs ne sont pas chargés **par défaut**
  - Les attributs associés ne seront chargés qu'à la première utilisation, ce qui peut **améliorer les performances** si vous ne souhaitez pas toujours charger les données associées

```
@OneToMany(mappedBy="petStore", fetch=FetchType.LAZY)
private Collection<Animal> animals
```

- Quand une entité est récupérée, les attributs sont automatiquement chargés
  - peut être pratique lorsque vous savez que vous avez toujours besoin de ces données

```
@OneToMany(mappedBy="petStore", fetch=FetchType.EAGER)
private Collection<Animal> animals
```

## Le Lazy loading 2/2

- Par défaut, les relations multi-valeurs (List, Set, Map, ...) sont chargées en **Lazy**
  - **Ce mode par défaut est fortement conseillé**
- Pour récupérer les animaux de PetStore, il est nécessaire d'appeler le getter dans la requête avec l'EntityManager

```
TypedQuery<PetStore> query = em.createQuery("SELECT p FROM PetStore p JOIN FETCH p.animals  
WHERE p.id=:id", PetStore.class);  
query.setParameter("id", 1L);  
PetStore petStore = query.getSingleResult();  
petStore.getAnimals();
```

- Sinon, une erreur sera levée :

```
Exception in thread "main" org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role:  
PetStore.animals: could not initialize proxy - no Session
```

# Exercice (1/2)

- Créer une classe JavaBean nommée **Comment**
  - Dans un sous-package **dao.entity**
  - Définir des attributs pour définir un commentaire
- Transformer cette en une entité JPA (Annotations)
- La table doit être nommée **comments**
  - L'attribut id doit être la primary key de la table

# Exercice (2/2)

- Définir une relation entre les entités *Post* et *Comment*
  - Un *commentaire* est rattaché à un *post*
  - Un *Post* peut avoir plusieurs *commentaires*
- Mettre à jour votre code afin de pouvoir écrire des commentaires
- **Pourrions-nous centraliser nos méthodes CRUD ?**
  - Si oui, comment ? Sinon, pourquoi ?

Java Persistence API

# LES BONNES PRATIQUES

*DAO & pattern factory*



Les bonnes pratiques

# DAO (Data Access Object Pattern)

- Différentes *méthodes* sont disponibles pour stocker les informations
  - Base de données Relationnelle
  - Base de données orientée Objet
  - Fichier plat
  - LDAP
  - ...

Les bonnes pratiques

# Data Access Object Pattern

Si votre application change de méthode

- Comment limiter les changements dans le code ?
- Comment facilement faire évoluer l'application ?

Solution : ajouter une couche d'abstraction pour centraliser l'accès aux données

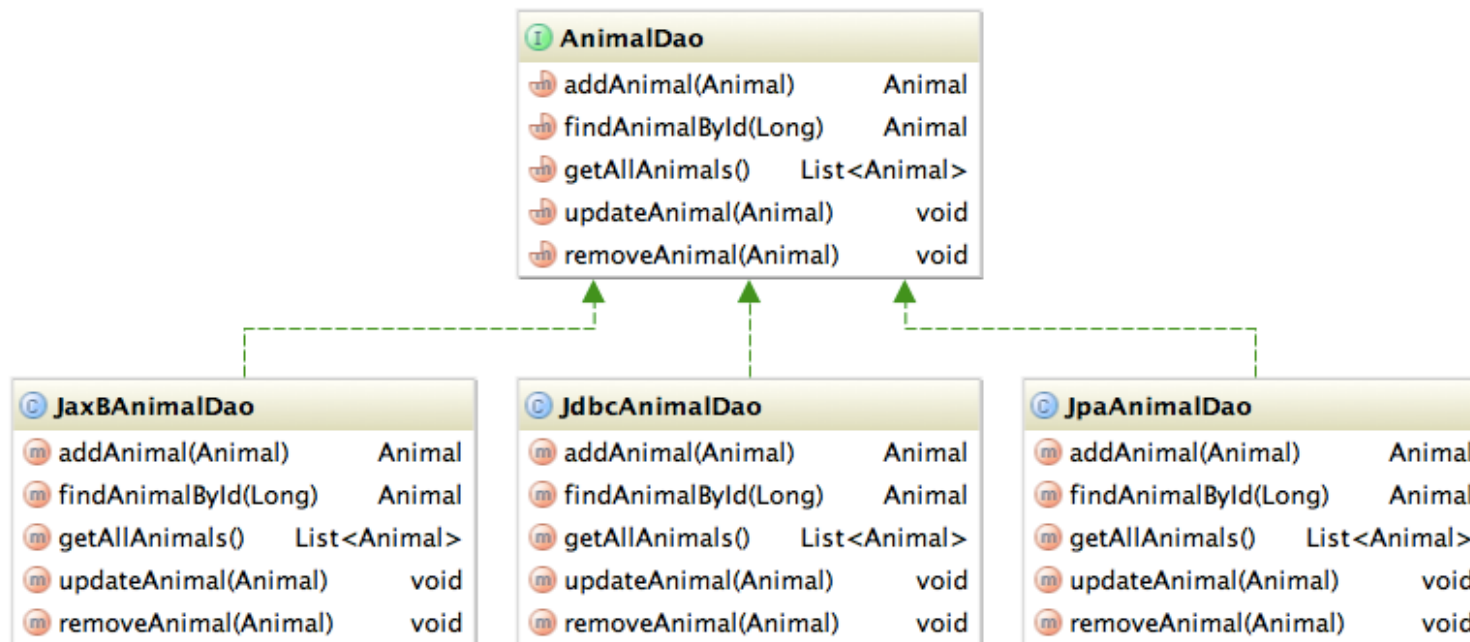
**C'est patron de conception Data Access Objects**



Les bonnes pratiques

# Data Access Object Pattern

- Une interface définit les méthodes d'accès aux données (CRUD)
- Plusieurs implémentations différentes

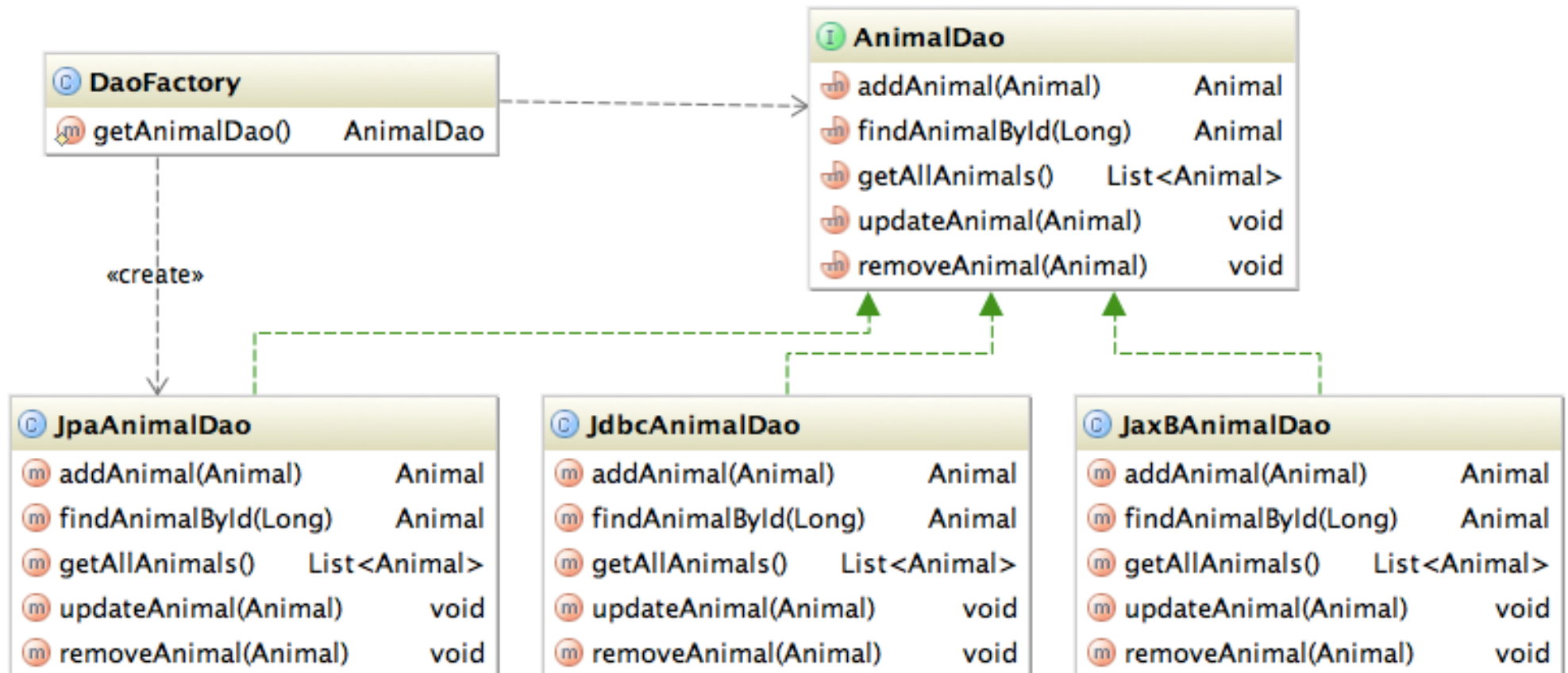


# Data Access Object Pattern

- Comment supprimer la dépendance entre les autres classes et l'implémentation DAO ?
  - Utiliser le **type inference**
    - Définissez vos variables avec le **type de l'interface** au lieu du type de l'implémentation
  - Utiliser une **factory** pour créer les objets DAO
    - Déléguez la création des instances à un seul endroit
    - Quand vous souhaitez changer l'implémentation à utiliser, modifiez le *factory* !

Les bonnes pratiques

# Factory Pattern



# Factory Pattern

- Exemple:

```
public class DaoFactory {  
    //Private constructor prevent instantiation  
    private DaoFactory() {}  
  
    public static AnimalDao getAnimalDao() {  
        return new JpaAnimalDao(  
            PersistenceManager.getEntityManagerFactory());  
    }  
}
```

Les bonnes pratiques

# EntityManagerFactory

Les Instances sont coûteuses à la création mais sont thread-safe...

- Il est important d'utiliser une seule instance. Comment ?
  - Créer un singleton lazy-loading
- Il est important de fermer l'instance quand l'application s'arrête. Comment ?
  - Créer un **Hook** qui détecte l'arrêt du programme

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {  
    // Code à exécuter lors de la fermeture du programme  
    System.out.println("Le programme se ferme. Exécution du code de fermeture...");  
}));
```

# EntityManagerFactory (1/2)

```
public class PersistenceManager {  
    private static EntityManagerFactory emf;  
  
    // Lazy initialization  
    public static EntityManagerFactory getEntityManagerFactory() {  
        if(emf == null){  
            emf = Persistence.createEntityManagerFactory("My-PU");  
        }  
        return emf;  
    }  
}
```

## EntityManagerFactory (2/2)

```
private PersistenceManager() {  
    //Private constructor prevent instantiation  
}  
  
public static void closeEntityManagerFactory() {  
    if(emf != null && emf.isOpen()) emf.close();  
}  
}
```

Les bonnes pratiques

## Exercice (1/3)

- Créer un sous-package **dao.util**
- Créer une classe **PersistenceManager** dedans :
  - Nommer la **PersistenceManager**
  - Définir une méthode statique qui retourne toujours la même instance d'un EntityManagerFactory
  - Définir une méthode statique qui ferme cette instance



Les bonnes pratiques

## Exercice (2/3)

- Dans un sous package **dao.jpa**, mettre en place :
- Deux interfaces **PostDAO** et **CommentDAO**
  - Elles définissent un CRUD générique
- Deux classes **PostJpaDAO** et **CommentJpaDAO**
  - Elle implémenterons les interfaces

Les bonnes pratiques

## Exercice (3/3)

- Créer une classe **DaoFactory** dans le package dao
  - Définir un constructeur privé
  - Définir 2 méthodes
    - Une qui retourne une nouvelle instance de **PostDao**
    - Une autre qui retourne une nouvelle instance de **CommentDao**
- Utiliser vos DAO au lieu de l'EntityManager dans votre code

Java Persistence Query Language

# JPQL



JPQL

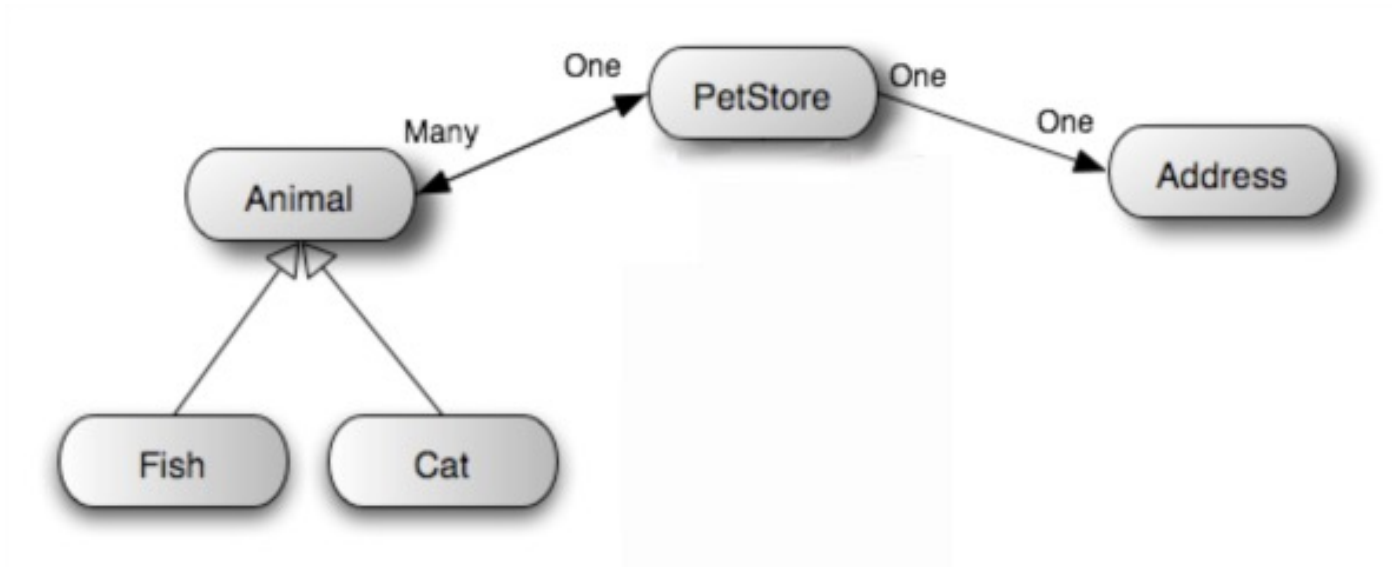
# JPQL

- **Java Persistence Query Language**
- Permet de faire des requêtes avec des entités stockées dans une base de données relationnelle
- Ressemble à SQL
- Manipule des objets Java au lieu de tables de base de données

JPQL

# Présentation

- JPQL manipule les données à travers une représentation objet de la donnée en base



- C'est appelé "abstract schema"

JPQL

# Utilisation du JPQL

- Pour écrire une requête, nous avons besoin :
  - D'un EntityManager
  - Du langage JPQL
  - D'un objet à requêter
- L' EntityManager est capable de créer les requêtes d'objets et les exécuter

JPQL

# Requête SELECT

Récupérer toutes les entrées d'une entité dans une table

- Récupérer un EntityManager
- Créer un objet *Query*
- Exécuter la requête

```
EntityManager em = ...  
  
Query query = em.createQuery("SELECT s FROM Store AS s");  
  
List<Store>list = query.getResultList();
```

JPQL

# Clause WHERE

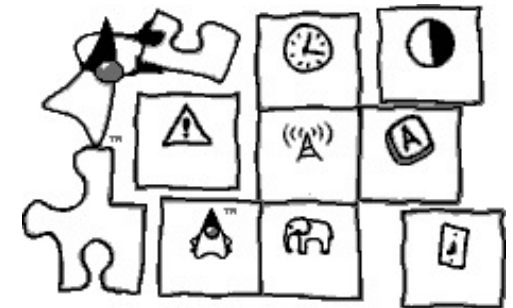
- Appliquer des conditions à une requête

```
EntityManager em = ...
```

```
Query query = em.createQuery("SELECT store FROM Store AS store WHERE  
store.id = 5");
```

```
Store myStore = (Store)query.getSingleResult();
```

- Autres fonctions
  - IS NULL
  - BETWEEN
  - ...
  - LIKE
- Ordre des résultats avec ORDER BY





JPQL

# Requêtes DELETE et UPDATE

- Supprimer des entités en utilisant JPQL

```
Query query = em.createQuery("DELETE FROM Store AS s WHERE s.id = 2");  
int nbrDeleted = query.executeUpdate();
```

- Mettre à jour des entités en utilisant JPQL

```
Query query = em.createQuery("UPDATE Store AS s SET s.name = \"Boutique\"  
    WHERE s.id = 2");  
int nbrUpdated = query.executeUpdate();
```

# Requêtes avec des paramètres

- Les Paramètres peuvent être placés dans les requêtes
  - Paramètres Numériques

```
Query query = em.createQuery("SELECT s FROM Store AS s WHERE s.id =  
?1");  
query.setParameter(1, 5);  
Store myStore = (Store)query.getSingleResult();
```

- Paramètres en chaine de caractères (recommandé)

```
Query query = em.createQuery("SELECT s FROM Store AS s WHERE s.id =  
:id");  
query.setParameter("id", 5);  
Store myStore = (Store)query.getSingleResult();
```

# Les fonctions d'agrégation

- Ces fonctions peuvent être utilisées dans un `SELECT`
  - `MIN`
  - `AVG`
  - `COUNT`
  - `SUM`
  - ...

```
Query query = em.createQuery("SELECT MAX(cat.earLength) FROM Cat AS  
cat");
```

```
Number maxEarLength = (Number) query.getSingleResult();
```

JPQL

# Les fonctions d'agrégation

- Un opérateur spécial permet de requêter dans les collections d'objets au sein d'une relations : **IN**
- Exemple :
  - Je veux récupérer les *boutiques* contenant le *produit* nommé "produit test"

```
Query query = em.createQuery("SELECT s FROM Store AS s, IN(s.products)  
    AS p WHERE p.name = 'Produit test'");  
  
List<Store> stores = (List<Store>) query.getResultList();
```

JPQL

# Les requêtes nommées

- Il est possible de déclarer des requêtes nommées
  - Elles sont précompilées au déploiement

```
@Entity
@NamedQuery(name="listBeverages", query="SELECT beverage FROM Beverage
    AS beverage")
public class Beverage implements Serializable{ ... }
```

- Et pour les appeler :

```
Query query = em.createNamedQuery("listBeverages") ;
```

JPQL

# Compléter les blancs

JPQL est un langage proche du SQL.

L'intérêt est de manipuler des objets plutôt que des tables.

La manipulation des requêtes se fait avec la classe Query.

La majorité des fonctions SQL sont les même avec JPQL.



JPQL

## Exercice (1/2)

- Mettre en place une option pour récupérer des posts par date
- **Avec JPQL** : Récupérer les *posts* dont la date est de moins de 3 jours

JPQL

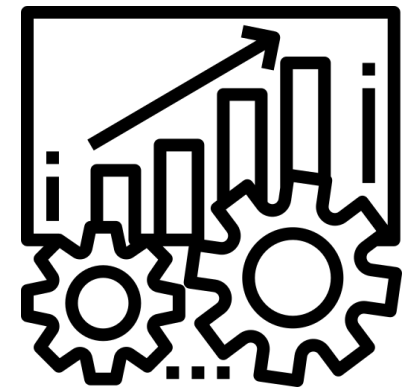
## **Exercice (2/2)**

- Consultez et Tester l'API Criteria Queries (JPA)



# Et sinon....JDBC vs JPA (+JDBC)

- **Les gains de JPA/Hibernate**
  - Mapping objet-relationnel
  - Indépendance vis-à-vis du SGBD
  - Gestion des transactions
  - Possibilité de ne pas écrire les requêtes SQL avec l'API Criteria
- **Les problèmes majeurs**
  - Difficultés de maintenance
  - Lisibilité des requêtes avec l'API Criteria
  - Gestion des transactions
  - Ralentissements possibles



- FIN -