Java Persistence API

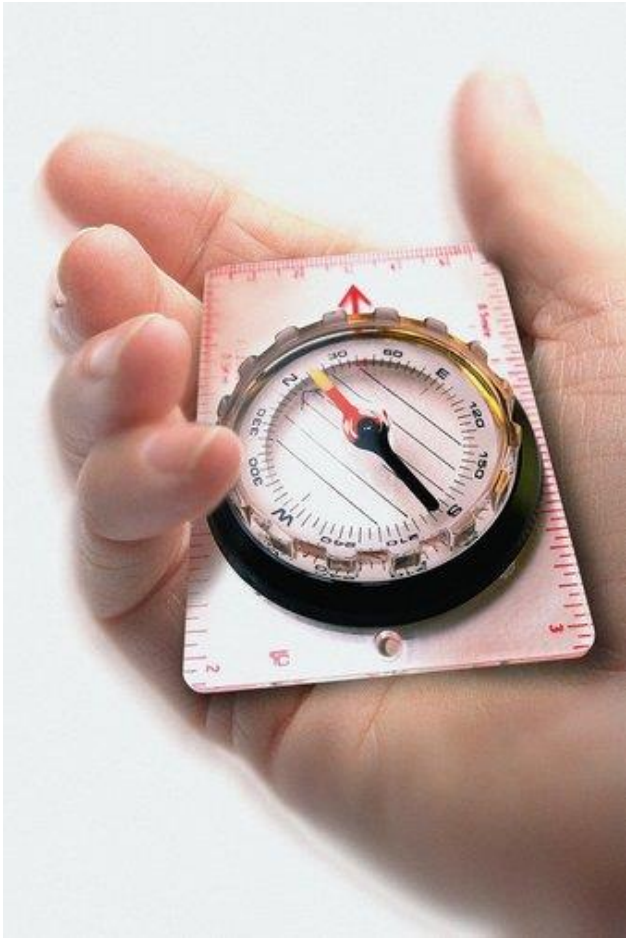
Object Relationnal Mapping

Objectifs du cours

En complétant ce cours, vous serez en mesure de:

- Expliquer ce qu'est JPA
- Utiliser l'API Java Persistence pour conserver les données
- Utiliser des modèles de couche de persistance célèbres

Plan de cours



- JPA Entity
- JPA – Fonctions avancées
- JPQL
- Bonnes pratiques

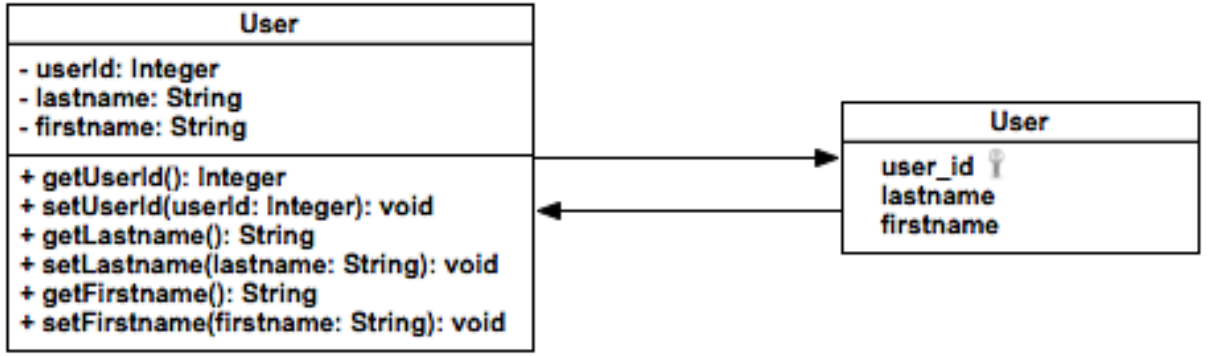
Java Persistence API

JPA ENTITY

Ou comment gérer notre base de données de manière transparente

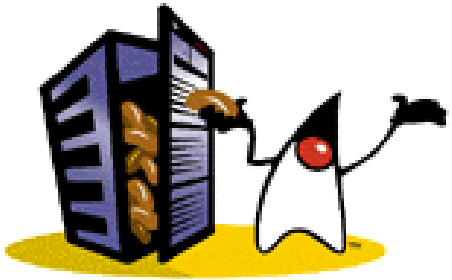
Overview

- JPA fait partie de la plate-forme Java EE
  - Inspiré de frameworks comme Hibernate et JDO
  - S'appuie fortement sur la fonction d'annotation
- La relation entre les objets et les tables se fait automatiquement (ORM : mappage « objet-relationnel »)



Overview

- Nécessaires :
  - Une base de données relationnelle
  - Un pilote JDBC sous forme de jar
  - Un fichier de configuration XML pour l'accès à la base de données
  - Une classe JavaBean, qui deviendra une entité JPA avec quelques annotations
  - Un JPA Entity Manager



Relational database

- La majorité des bases de données relationnelles
  - MySQL
  - PostGreSQL
  - Oracle
  - SQL Server
  - DB2
  - …

Entity annotation

- Une entité JPA est juste un POJO avec des propriétés privées, des getters et des setters, un constructeur par défaut :

```
public class Contact implements Serializable {
   // my properties
   private int id;
   private String name;
   private String firstname;

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

Entity annotation

- Agrémenté d'annotations :

```
@Entity
public class Contact implements Serializable {
   // my properties
   @Id
   private int id;
   private String name;
   private String firstname;

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

Entity annotation

- @Entity une annotation est mise sur la classe : elle sert à déclarer une classe en tant que « JPA Entity »
- @Table l'annotation permet de définir le nom de la table sur laquelle la classe est mappée (optionnel)

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

}
```

Annotation des propriétés

- **@Id** l'annotation est définie sur la propriété ou sur le getter représentant la clé primaire dans la base de données
- Il est possible de définir comment générer la clé avec l'annotation :
  - **@GeneratedValue**(strategy=GenerationType.XXX)
- Les constantes proposées sont :
  - IDENTITY - TABLE
  - SEQUENCE - AUTO

Annotation des propriétés

- Quelques annotations importantes :

Annotation	Description
@Basic	Si aucune annotation spécifique n'est déclarée, celle-ci est utilisée
@Transient	Quand on ne veut pas rendre une propriété persistante
@Lob	Permet de stocker de grosses chaînes, des tableaux d'octets, …
@Temporal	Utilisé pour conserver les dates, les heures
@Enumerated	Spécifier un champ énuméré

Persistence providers

- Il existe différentes implémentations JPA :

HIBERNATE

ORACLE TOPLINK

eclipselink

OpenJPA

- Le code reste le même, seul le fichier de configuration change

Persistence unit

- L'unité de persistance est l'élément clé de la technologie JPA Entity
- Il « persiste » les entités dans la base de données.
- Nécessite un fournisseur de persistance et d'autres configurations dans un fichier spécial : **persistence.xml**

persistence.xml Example 1/2

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

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

        <provider>
            org.hibernate.ejb.HibernatePersistence
        </provider>

        <properties>
            <property
           name="javax.persistence.jdbc.driver"
           value="com.mysql.jdbc.Driver" />
...
```

persistence.xml Example 2/2

```
...
        <property
          name="javax.persistence.jdbc.user"
          value="root" />
        <property
          name="javax.persistence.jdbc.password"
          value="root" />
        <property
          name="javax.persistence.jdbc.url"
          value="jdbc:mysql://host:3306/MyDB" />
         <property
           name="hibernate.hbm2ddl.auto"
           value="update" />
      </properties>
    </persistence-unit>
  </persistence>
```

JDBC driver

- Chaque base de données fournit un pilote JDBC pour y accéder via Java
- Selon la base de données utilisée, le JAR approprié doit être placé dans les bibliothèques

Entity Manager

- Comme son nom l'indique, l'objet Entity Manager va gérer toutes les opérations sur les entités : insertion, modification, suppression de celles-ci dans la base de données
- Aucun code SQL n'est requis, nous manipulons directement les objets Java :

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

Entity Manager

- Quelques opérations courantes :
  - void persist(Object entity)
  - <T> T merge(T entity)
  - void remove(Object entity)
  - <T> T find(Class<T> entityClass, Object primaryKey)

- Grâce à eux, il n'y a presque pas d'écriture de requête

EntityManager

- Comment le récupérer ?
  - En utilisant une EntityManagerFactory !

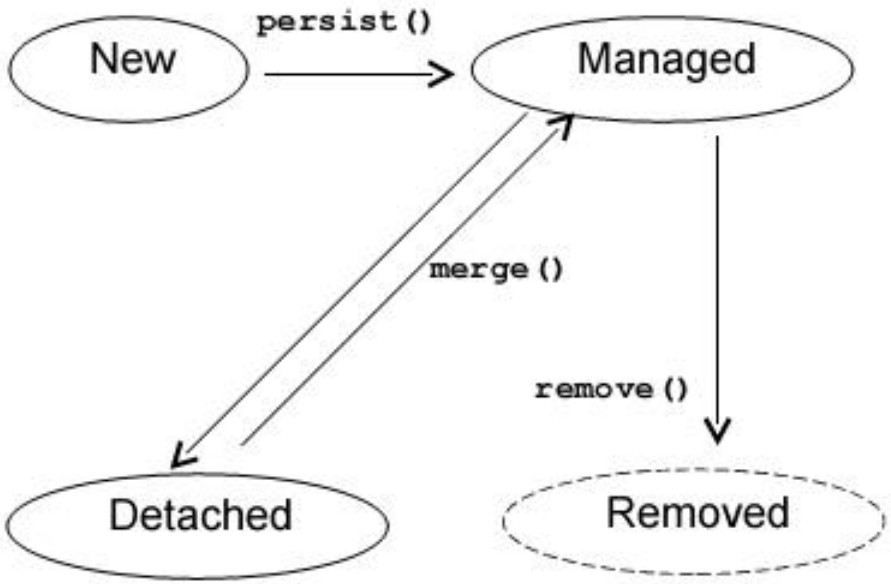
```
EntityManagerFactory emf = null;
emf = Persistence.createEntityManagerFactory("My-PU");
EntityManager em = emf.createEntityManager();
Contact contact = em.find(Contact.class, 1);
em.close();
emf.close();
```

EntityManager

- Les objets EntityManager ne sont pas thread-safe…
  - N'en définissez pas une comme variable d'instance de servlet !
- … mais EntityManagerFactory l'est.
  - Vous pouvez utiliser la même instance pour toutes vos applications
- Pensez à fermer vos objets EntityManager et EntityManagerFactory !

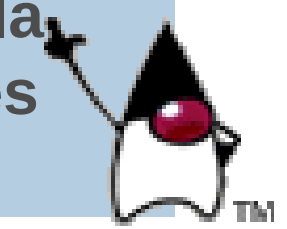
Entity states

- **Transient**: Après l'appel de la clé **new**
- **Managed**: après l'appel de la méthode **persist()**
- **Detached**: lorsque l'objet est manipulé sur le client
- **Removed**: objet supprimé de la base de données

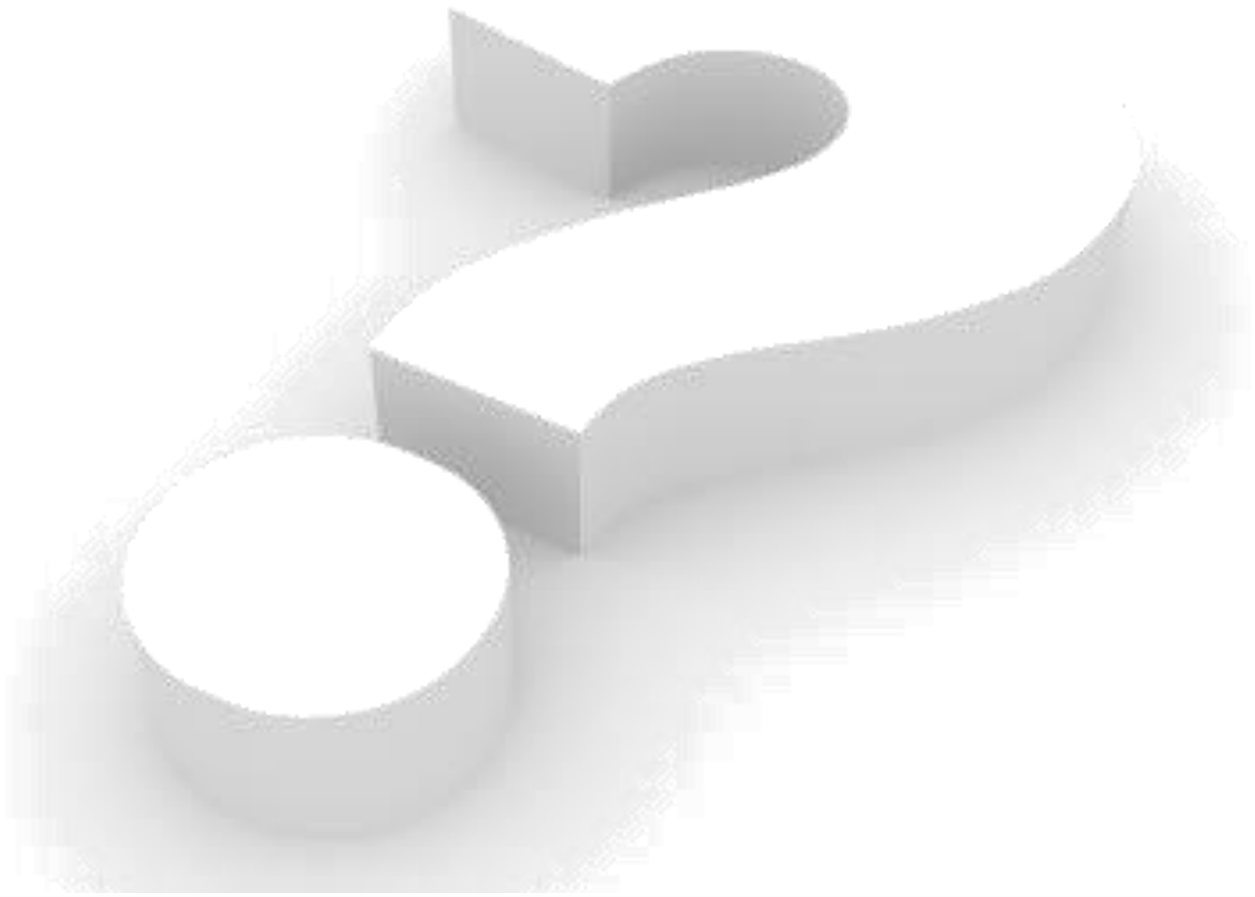


Quizz

Quelle annotation permet de déclarer une entité JPA ?	@Entity
Quelle annotation permet de déclarer une clé primaire ?	@Id
Comment déclarer la connexion avec une Base ?	En déployant un fichier décrivant une unité de persistance
A quoi sert Entity Manager ?	Ses méthodes gèrent la persistance des entités



Questions ?



Exercices (1/3)

- Ajouter des bibliothèques Hibernate à votre projet
- Ajoutez également la bibliothèque MySQL JDBC
- Créer une classe JavaBean nommée **Category**
  - Dans un package **com.cci.supcommerce.entity**
  - Avec **id** en **Long** et **name** en **String**
- Transformez-le en une entité JPA
  - La table doit être nommé **categories**
  - Le champ **id** doit être la clé primaire de la table

Exercices (2/3)

- Créer une unité de persistance
- Créez une page JSP nommée **addCategory.jsp**
- Créez un **HttpServlet** nommé **AddCategoryServlet**
  - Liez-le à **/auth/addCategory** url-pattern
  - Remplacer la méthode **init()**
    - Créer un objet EntityManagerFactory
  - Remplacer la méthode **destroy()**
    - Fermez l'objet EntityManagerFactory

Exercices (3/3)

- Toujours avec le **HttpServlet** nommé **AddCategoryServlet**
  - Remplacer la méthode **doPost()**
    - Récupérer les paramètres du formulaire
    - Créer un nouvel objet **Category**
    - Définir les paramètres dans l'objet
    - Utiliser un EntityManager pour persister l'objet

- Remplacer la méthode **doGet()**
  - Transférer (Forward) la demande à la page JSP

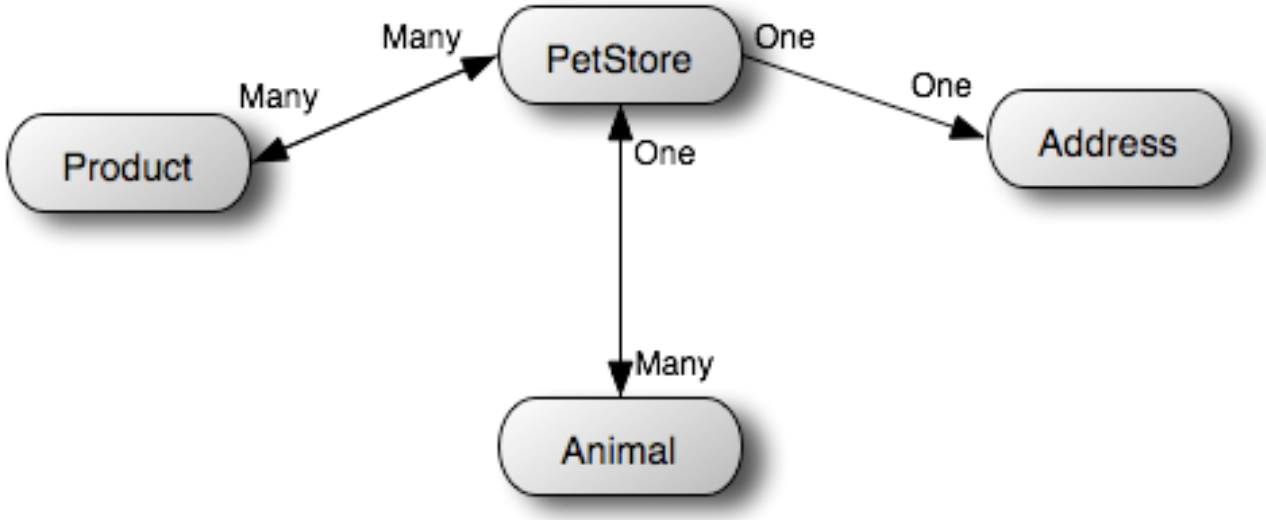
Java Persistence API

JPA – FONCTIONS AVANCÉES

Entities dependencies, inheritance

Relation entre les entités

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



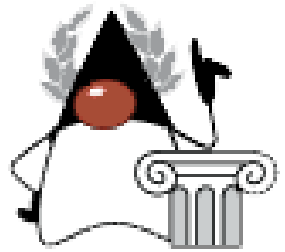
Relationship between Entity Beans

- Les relations entre les entités sont décrites avec des annotations placées sur la propriété ou sur le getter
  - Différentes stratégies sont disponibles (clés étrangères, tables de jointure)
- JPA gère également la relation d'héritage entre les entités avec des annotations

@

One-To-One

- **@OneToOne** L'annotation décrit une relation un-à-un entre deux entités.
- Il existe 3 stratégies différentes :
  - **@JoinColumn**
    - une clé étrangère est utilisée
  - **@PrimaryKeyJoinColumn**
    - 2 entités dépendantes ont la même clé primaire
  - **@JoinTable**
    - une table de jointure contient des clés primaires



One-To-One

- Par exemple, une entité de magasin n'a qu'une seule adresse:

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

- Dans la table de l'animalerie, une clé étrangère est utilisée

One-To-Many and Many-To-One

- Les annotations **@OneToMany** et **@Many-To-One** relient une entité à une collection d'une autre entité
  - Exemple:
    - Une personne a plusieurs comptes bancaires et chaque compte a un propriétaire unique
- Représenté soit par une table de jointure, soit par une colonne sous forme de clé étrangère
  - @JoinTable
  - @JoinColumn

One-To-Many and Many-To-One

- Exemple de code avec une animalerie vendant de nombreux animaux :

Store Entity

```
@OneToMany(mappedBy="petStore")
private Collection<Animal> animals;
```

Animal Entity

```
@ManyToOne
@JoinColumn(name="store_fk")
private PetStore petStore;
```

- Une colonne de clé étrangère est ajoutée dans la table des animaux

Many-ToMany

- L'annotation **@ManyToMany** lie 2 entités entre elles
  - Exemple :
    - Un produit peut avoir plusieurs catégories et une catégorie contient plusieurs produits

- **@JoinTable** est la seule option

Many-To-Many

- Comment annoter vos entités :

Store Entity	`@ManyToMany`
	`@JoinTable(name="STORE_PRODUCT")`
	`private Collection<Product> products;`

Product Entity	`@ManyToMany(mappedBy="products")` `private Collection<PetStore> stores;`

- Une table de jointure représente la relation entre les magasins et les produits

Cascading

- Toutes les annotations de relation précédentes possèdent l'attribut cascade
- Une opération appliquée à une entité est répercutée sur les entités dépendantes
  - Exemple : lorsqu'un utilisateur est persisté, ses comptes le sont aussi.
- Quatre types:
  - `PERSIST|MERGE|REMOVE|REFRESH`
  - `CascadeType.ALL` : combine les 4

Cascading

- L'attribut cascade est défini à côté de l'annotation
  - Entité PetStore avec une adresse unique

PetStore Entity	

```
@OneToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name="address_fk")
private Address address;
```

- Si l'adresse n'existe pas dans la base de données, elle est conservée en même temps que le magasin

Lazy loading (Chargement paresseux)

- Toutes les annotations de relation précédentes possèdent l'attribut fetch
- Lorsque vous récupérez une entité, les propriétés à valeurs multiples ne sont pas chargées par défaut.
  - Exemple : lorsqu'un utilisateur est chargé, ses comptes ne sont pas récupérés.
- 2 types : `LAZY` | `EAGER`

Lazy loading

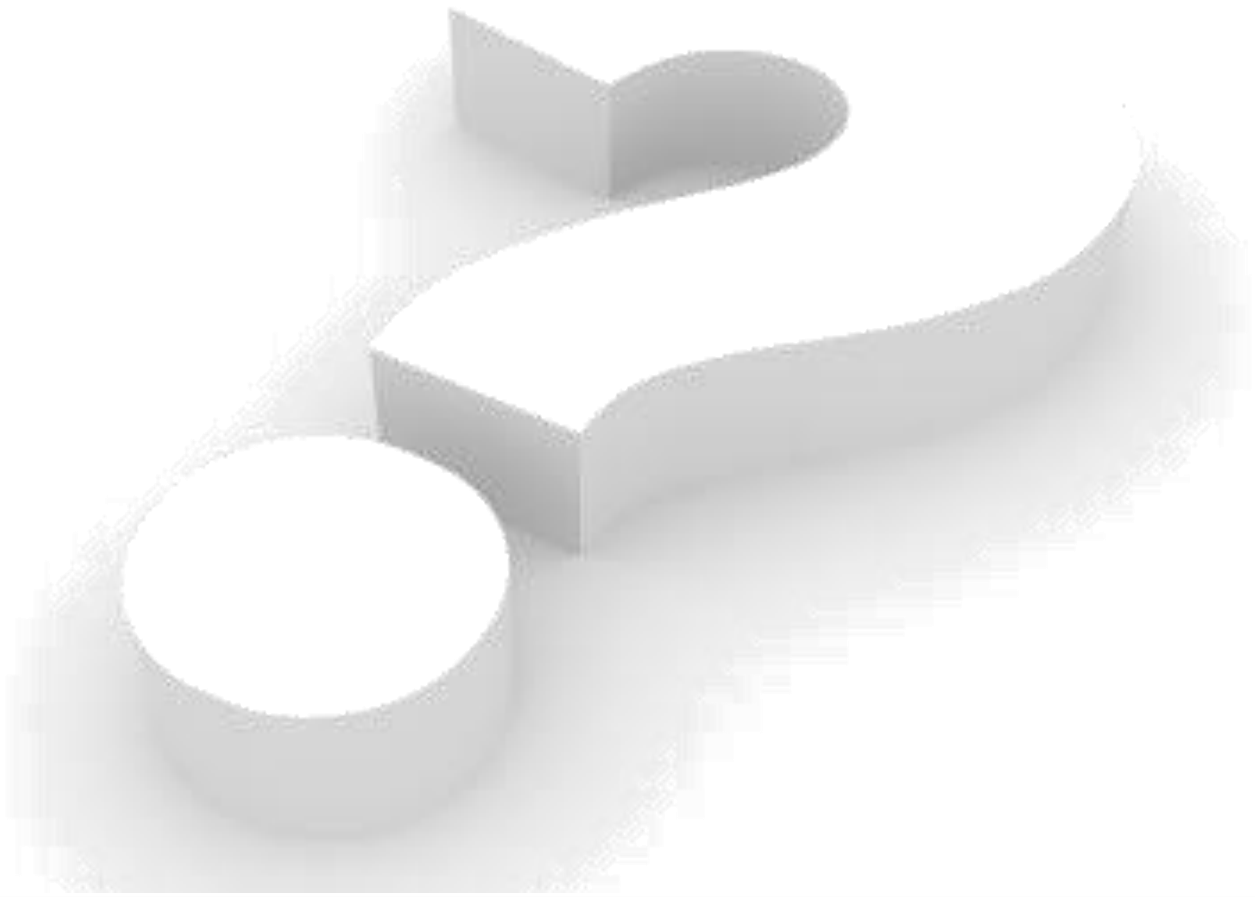
- Par défaut, le mode "paresseux" est appliqué pour les propriétés multi-valuées (List, Set, Map, …)
  - Changez-le en mettant la propriété fetch sur "eager" sur l'annotation

PetStore Entity

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

- Lorsqu'une animalerie est extraite de la base de données, sa collection est initialisée

Questions ?



Exercices (1/2)

- Créer une classe JavaBean nommée **Product**
  - Dans le package **com.cci.sun.supcommerce.entity**
  - Avec les mêmes attributs que la classe **SupProduct**

- Transformez-le en une entité JPA
  - Le tableau doit être nommé **products**
  - Le champ **id** doit être la clé primaire de la table

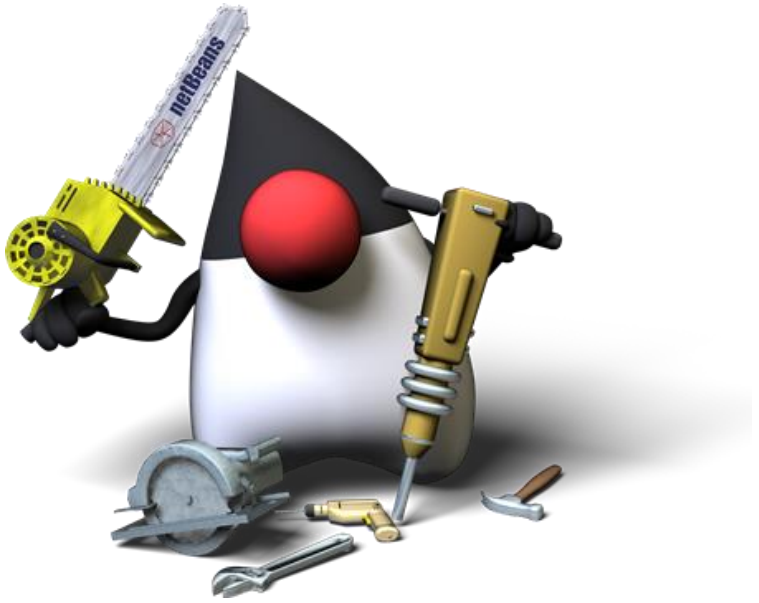
Exercices (2/2)

- Définir une relation entre les entités Product et Category
  - Un produit ne peut avoir qu'une seule catégorie
  - Une catégorie peut avoir plusieurs produits

- Mettre à jour **InsertSomeProductServlet**
  - Remplacer l'objet **SupProduct** par un objet **Product**
  - Utiliser EntityManager au lieu de la classe **SupProductDao**

Java Persistence API

JPQL



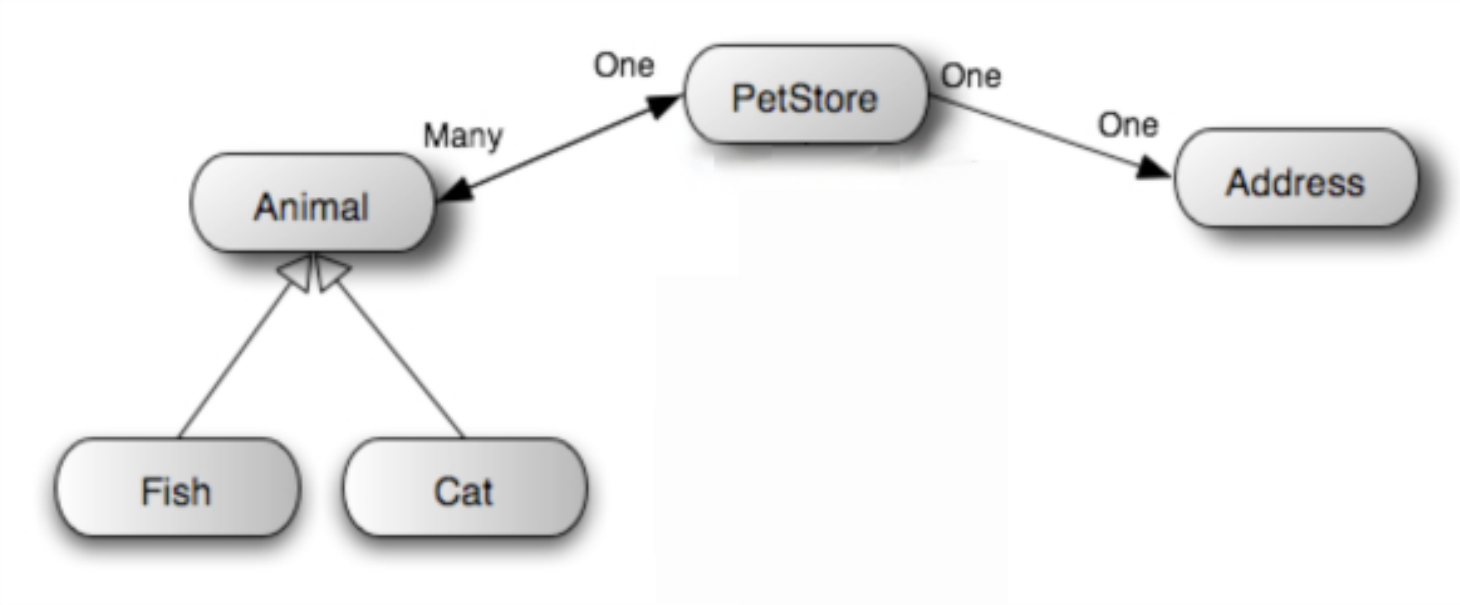
Le SQL "alimenté par l'objet"

Présentation

- Le gestionnaire d'entités gère les opérations CRUD* de base
- Java Persistence Query Language effectue des requêtes sur des entités stockées dans une base de données relationnelle
- Cela ressemble beaucoup à SQL, de nombreuses requêtes sont disponibles
- Au lieu de travailler sur des tables de base de données, il manipule les objets Java de manière transparente

Presentation

- JPQL manipule les objets via une représentation interne dans le conteneur Entity Beans



- C'est ce qu'on appelle le "schéma abstrait"

Comment

- Pour rédiger une requête, nous avons besoin :
  - Un gestionnaire d'entité
  - Le langage JPQL
  - Un objet de requête
- Le gestionnaire d'entités est capable de créer des objets de requête
- La requête est ensuite exécutée

Déclaration SELECT

- Récupère toutes les entrées d'une table d'entités
  - Obtenir un gestionnaire d'entité
  - Créer un objet Query puis l'exécuter

```
Query query = em.createQuery("SELECT c FROM Cat AS c");
List<Cat>list = query.getResultList();
```

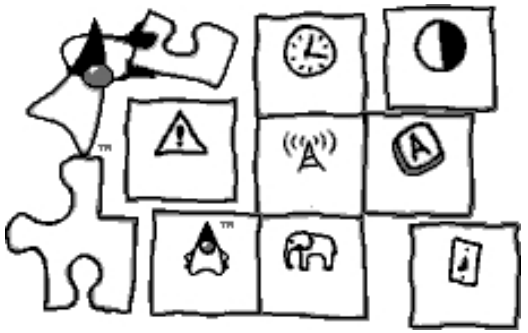
La clause WHERE

- Appliquer des conditions sur une demande

```
Query  query = em.createQuery("SELECT cat FROM Cat AS cat
   WHERE cat.animalId = 5");

Cat myCat = (Cat)query.getSingleResult();
```

- Certaines fonctions
  - BETWEEN - IS NULL
  - LIKE - …
- Ordonner les résultats avec ORDER BY



Déclarations DELETE et UPDATE

- Supprimer des entités à l'aide de JPQL

```
Query  query = em.createQuery("DELETE FROM Cat AS cat WHERE
   cat.earLength = 2");

int nbrDeleted = query.executeUpdate();
```

- Mettre à jour les entités à l'aide de JPQL

```
Query  query = em.createQuery("UPDATE Cat AS cat SET
   cat.earLength = 3 WHERE cat.earLength = 4");

int nbrUpdated = query.executeUpdate();
```

Requêtes avec paramètres

- Les paramètres peuvent être placés dans les requêtes
  - Paramètre numérique

```
Query query = em.createQuery("SELECT cat FROM Cat AS cat
   WHERE cat.animalId = ?1");
query.setParameter(1, 5);
Cat myCat = (Cat)query.getSingleResult();
```

  - Paramètre String

```
Query query = em.createQuery("SELECT cat FROM Cat AS cat
   WHERE cat.animalId = :id");
query.setParameter("id", 5);
Cat myCat = (Cat)query.getSingleResult();
```

Fonctions d'agrégation

- Les fonctions d'agrégation peuvent être utilisées avec la clause SELECT
  - MIN - SUM
  - AVG - …
  - COUNT

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

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

Fonctions d'agrégation

- Un opérateur spécial permet aux requêtes de travailler à travers des relations : IN
- Exemple :
  - Je veux obtenir les magasins contenant le produit nommé “Product” :

```
Query query = em.createQuery("SELECT s FROM Store AS s,
   IN(s.products) AS p WHERE p.name = 'Product'");

List<Store> stores = (List<Store>) query.getResultList();
```

Requêtes nommées

- Il est possible de déclarer des requêtes nommées sur la classe d'entité
  - Ils sont précompilés au déploiement

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

- Comment les appeler

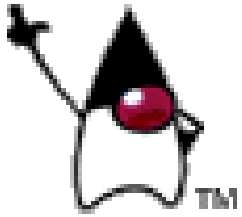
```
Query query = em.createNamedQuery("listBeverages");
```

Remplir les blancs

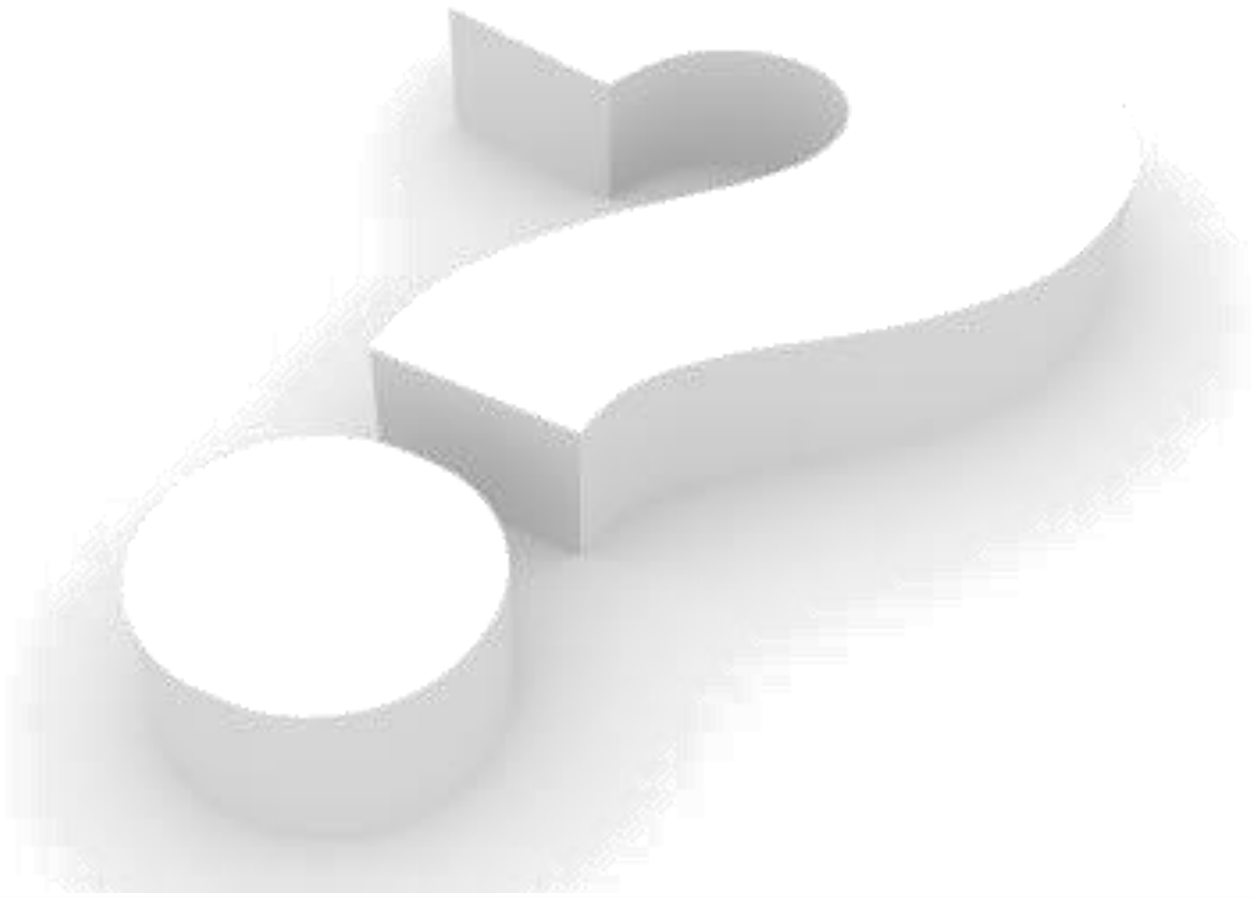
JPQL est un langage proche de ..SQL...

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

La manipulation des requêtes se fait avec la classe..Query..



Questions ?



Exercices (1/3)

- Remplacez tous vos objets SupProduct par des objets Product
- Utiliser EntityManager au lieu de la classe SupProductDao
  - Pensez à les fermer !

- Créer un **HttpServlet** nommé **CheaperProductsServlet**
  - Remplacer la méthode **doGet()**
    - Récupérez tous les produits avec un prix < 100 – **Utilisez JPQL !**
    - Ajoutez-les en tant qu'attributs de requête
    - Transférer la demande à listProduct.jsp

Exercices (2/3)

- Mettez à jour **AddProductServlet**
  - Dans la méthode **doGet()**
    - Récupérez toutes les catégories et les mettre dans l'attribut de requête
  - Dans la méthode **doPost()**
    - Récupérez l'identifiant de la catégorie dans les paramètres de la requête
    - Récupérez avec elle la catégorie depuis la base de données
    - Définissez-la à l'intérieur de l'objet **product** avant de le persister

Exercices (3/3)

- Mettez à jour la page **addProduct.jsp**
  - Ajoutez dans le formulaire un champ de sélection pour choisir la catégorie
- Mettez à jour la page **showProduct.jsp**
  - Afficher le nom de la catégorie du produit

Java Persistence API

BONNES PRATIQUES

DAO & Factory patterns

Data Access Object Pattern

- Diverses méthodes sont disponibles pour stocker des informations
  - Base de données relationnelle
  - Base de données orientée objet
  - Flat files
  - LDAP
  - …

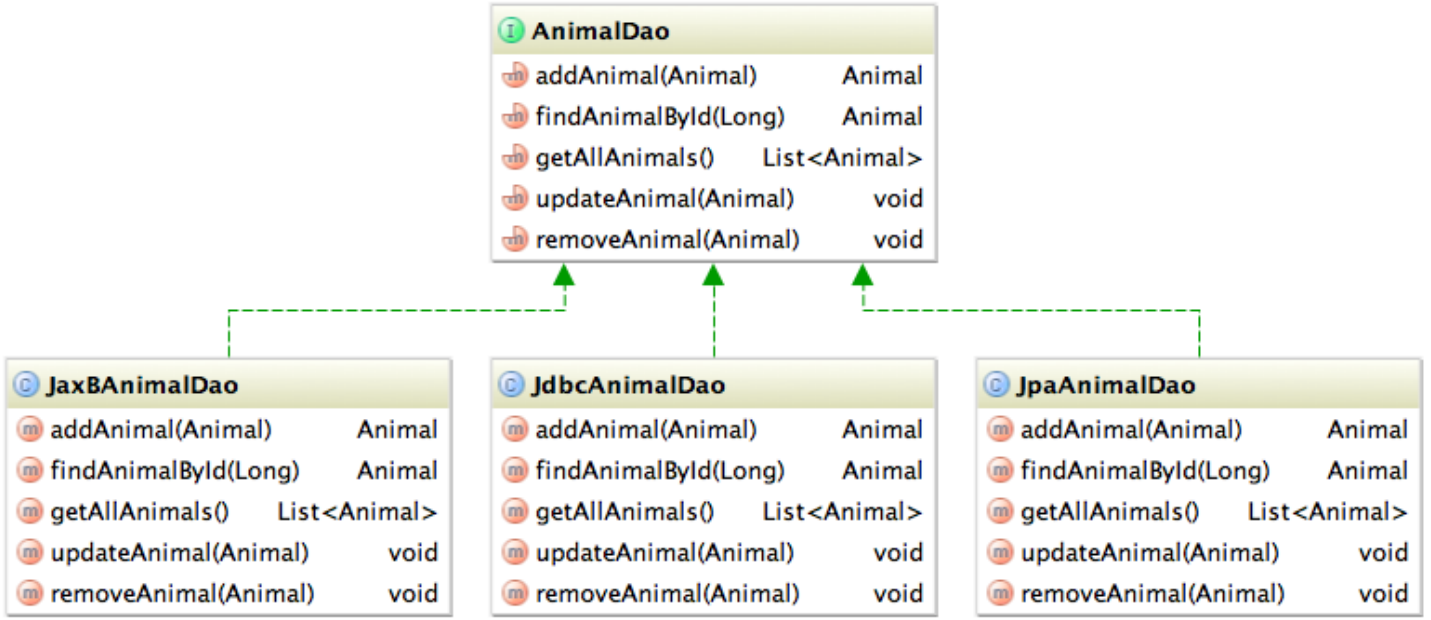
Data Access Object Pattern

- Si votre application passe à une autre méthode
  - Comment limiter l'impact sur le code ?
  - Comment faire évoluer facilement l'application ?

- Solution : ajouter une couche abstraite pour centraliser l'accès aux données
  - Avec les **Data Access Object** (objets d'accès aux données)

Data Access Object Pattern

- Une interface définit les méthodes d'accès aux données nécessaires
- Plusieurs implémentations différentes

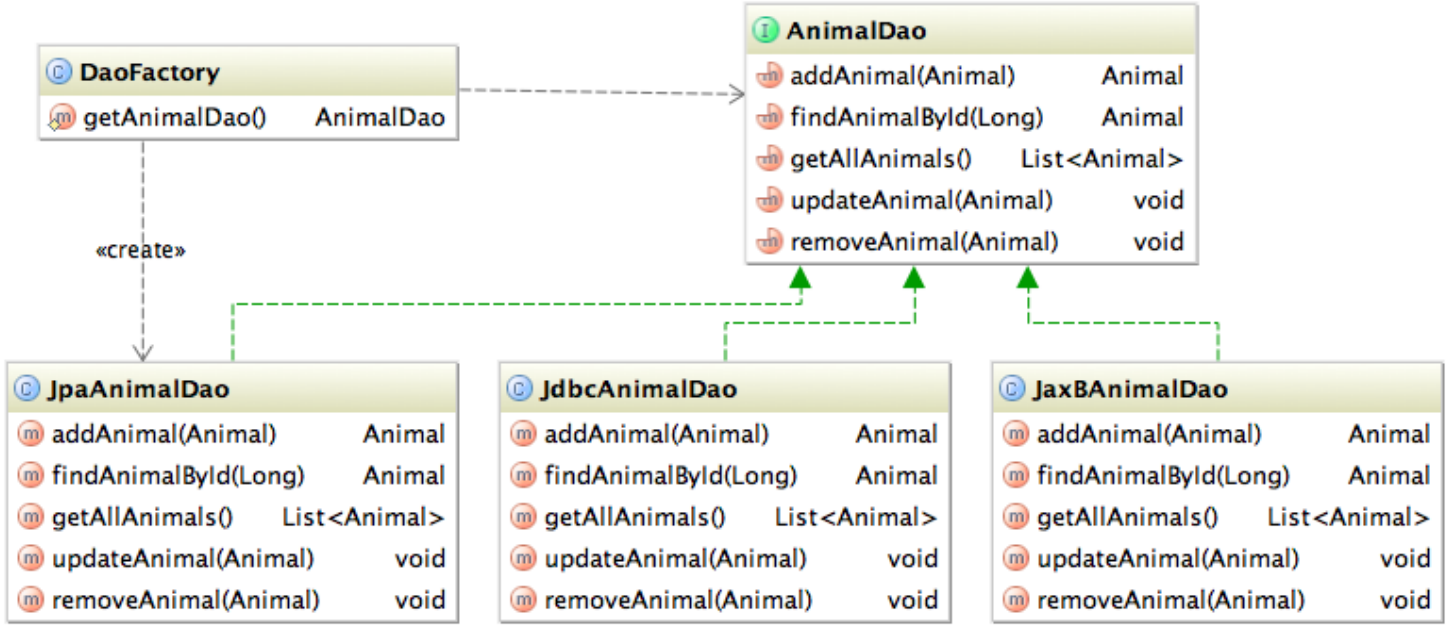


Data Access Object Pattern

- Comment supprimer la dépendance entre les autres classes et les implémentations DAO ?
  - Utiliser **l'inférence de type**
    - Définissez vos variables avec le **type d'interface** au lieu du type d'implémentation
  - Utiliser une **factory** pour créer un objet DAO
    - Déléguez la création d'instances en un seul point
    - Lorsque vous voudrez changer l'implémentation à utiliser, modifiez simplement la fabrique !

Factory Pattern

- Déléguez la création d'instances en un seul point
  - Quand vous voudrez changer l'implémentation à utiliser
    - Modifiez simplement votre usine !



Factory Pattern

- Exemple:

```
public class DaoFactory {
   //Private constructor prevent instantiation
   private DaoFactory(){}

   public static AnimalDao getAnimalDao() {
      return new JpaAnimalDao(
          PersistenceManager.getEntityManagerFactory());
   }
}
```

EntityManagerFactory

- Les instances sont coûteuses à créer mais thread safe…
- Comment n'utiliser qu'une seule instance ?
  - Créez sa propre usine !
- Comment le détruire lorsque l'application web se termine ?
  - Créez un **ServletContextListener** !

Good practices

EntityManagerFactory

- Factory exemple 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

- Factory exemple 2/2:

```
  //Private constructor prevent instantiation
  private PersistenceManager(){}

  public static void closeEntityManagerFactory() {
    if(emf != null && emf.isOpen()) emf.close();
  }
}
```

ServletContextListener

- ServletContextListener exemple 1/2:

```
public class PersistenceAppListener
                          implements ServletContextListener {
   // Call on application initialization
   public void contextInitialized(ServletContextEvent evt){
       // Do nothing
   }

   // Call on application destruction
   public void contextDestroyed(ServletContextEvent evt) {
       PersistenceManager.closeEntityManagerFactory();
   }
}
```

ServletContextListener

- ServletContextListener exemple 2/2:

```
<web-app>
   ...
   <listener>
      <listener-class>
         com.cci.myapp.listener.PersistenceAppListener
      </listener-class>
   </listener>
   ...
</web-app>
```

Exercices (1/5)

- Créez un nouveau paquet
  - Nommez-le **com.cci.supcommerce.util**

- Créer une classe à l'intérieur
  - Nommez-le **PersistenceManager**
  - Définissez une méthode statique qui renvoie toujours la même instance de **EntityManagerFactory**
  - Définissez une méthode statique pour fermer cette instance de fabrique

Exercices (2/5)

- Créer un nouveau paquet
  - Nommez-le **com.cci.supcommerce.listener**
- Créez une classe à l'intérieur
  - Nommez-la **PersistenceAppListener**
  - Implémentez **ServletContextListener**
  - Dans la méthode **contextDestroyed(…)**
    - Fermez votre instance EntityManagerFactory
  - Déclarez votre nouvau listener dans le fichier web.xml ou avec la bonne annotation

Exercices (3/5)

- Créer un nouveau paquet
  - Nommez-le **com.cci.supcommerce.dao**
- Créez deux nouvelles interfaces à l'intérieur
  - Nommez le premier **ProductDao**
    - Définissez toutes les méthodes d'accès aux données dont vous avez besoin pour gérer les entités Produit
  - Nommez le deuxième **CategoryDao**
    - Définissez toutes les méthodes d'accès aux données dont vous avez besoin pour gérer les entités de catégorie

Good practices

Exercices (4/5)

- Créer un nouveau paquet
  - Nommez-le **com.cci.supcommerce.dao.jpa**
  - Créez deux nouvelles classes à l'intérieur
    - Nommez le premier **JpaProductDao**
      - Implémentez l'interface **ProductDao**
      - Définissez un constructeur avec un paramètre **EntityManagerFactory**
    - Nommez le second **JpaCategoryDao**
      - Implémentez l'interface **CategoryDao**
      - Définissez un constructeur avec un paramètre **EntityManagerFactory**

Exercices (5/5)

- Créez une classe **DaoFactory**
  - Dans le package **com.cci.supcommerce.dao**
  - Définissez un constructeur privé
  - Définissez deux méthodes
    - Une qui renvoie une nouvelle instance de **ProductDao**
    - Une autre qui renvoie une nouvelle instance de **CategoryDao**

- Utilisez votre DAO au lieu d'EntityManager dans vos Servlets !

Fin

Merci de votre attention