Institut Informatique
Claude Chappe

Master STS Mention Informatique
Parcours ISI

Programmation distribuée
M1 / 178EN003

C4.4 – Introduction à *JPA (Java Persistence API)*

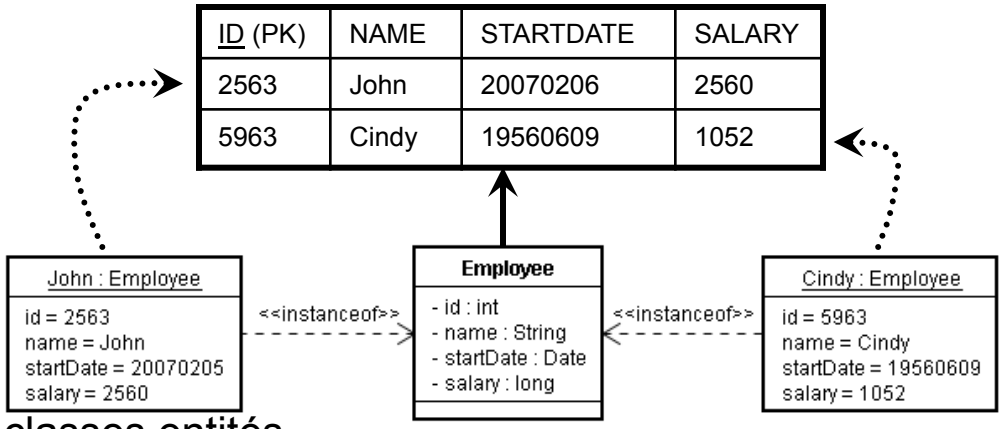
Thierry Lemeunier
thierry.lemeunier@univ-lemans.fr
www-lium.univ-lemans.fr/~lemeunie

Plan du cours

- Communication par composant distribué :
  - Introduction à *Java Persistence API*
    - Les entités
    - Les relations entre entités
    - Le cycle de vie des entités
    - Le gestionnaire de persistance et les opérations CRUD
    - Le langage de requête de JPA : JPQL
    - Les transactions

JPA - Les entités (1/6)

- *Java Persistence API* (JPA) fournit une spécification pour manipuler les bases de données relationnelles à partir d'une représentation objet du schéma relationnel : *Object/Relational Mapping* (*ORM*)
- Elle se compose :
  - d'une API (en particulier la classe *EntityManager*) pour gérer les opérations *CRUD* (*Create Read Update Delete*) classiques de gestion d'une BDR
  - d'annotations spécifiques pour représenter le modèle relationnel sous forme d'objets (les entités) et de liens entre entités
  - d'un langage de requête *JPQL* inspiré de SQL
- Principe de l'*ORM* :
  - Les tables de la base sont représentées par des classes entités
  - Une ligne de la table est représentée par une instance de la classe entité



- JPA permet :
  - De générer les tables à partir de classes entités
  - De générer les classes entités à partir de tables existantes

JPA - Les entités (2/6)

- Les classes entités (en bref) :
  - Une entité est une classe *POJO* (*Plain Old Java Object*)
  - Elle est annotée par *@Entity* (obligatoire)
  - Elle possède un constructeur par défaut public ou protégé
  - Elle implémente *Serializable* si une instance est amenée à être détachée du serveur (par exemple donnée au client)
  - L'état d'une entité est représenté par les valeurs de ses propriétés :
    - Entité = Classe JavaBean persistante
    - Les annotations de persistance sont :
      - soit sur les attributs
      - soit sur les accesseurs *JavaBean*
  - Les entités supportent l'héritage et le polymorphisme (non abordé ici !)
  - Les identificateurs d'entités (obligatoires) :
    - L'unicité de chaque instance est assurée par une clé primaire
    - Une clé peut être simple (un entier long par exemple) ou composée :
      - Une clé simple sur un champs ou un accesseur de propriété annotée *@Id*
      - Une clé composée est décrite par une classe (cf. doc pour les détails !)
  - Des annotations de classe permettent de décrire les liens relationnels avec les autres entités

JPA - Les entités (3/6)

- Quelques annotations :
  - @Table(name = "nom_de_la_table")
    - Indique le nom de la table de stockage des données
    - Optionnel (par défaut le nom est celui de la classe entité)
  - @GeneratedValue(strategy = la_stratégie)
    - Utilisé en option avec l'annotation @Id
    - Définit la stratégie de création de l'identifiant unique de l'objet
    - GenerationType.AUTO : génération auto. gérée par JPA (valeur par défaut)
    - GenerationType.IDENTITY : génération auto. gérée par la BD
    - GenerationType.TABLE : identifiant stocké dans une table utilisée par JPA
    - GenerationType.SEQUENCE : provient d'une séquence gérée par la BD

```
@Entity
@Table(name = "t_address")
public class Address {
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private Long id;
    private String street1;
}
```

JPA - Les entités (4/6)

- Quelques annotations (suite) :
  - @Column
    - Indique les propriétés d'une colonne : le nom, le type, la taille, si la valeur *null* est autorisé, si la valeur est unique, *etc.*
    - Optionnel (par défaut le nom est celui de l'attribut)

```
@Entity
@Table(name = "t_address")
public class Address {
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable=false, unique=true)
    private String name;

    @Column(nullable=false)
    private String street1;

    @Column(nullable=false, length=100)
    private String city;

     @Column(name="zip_code", nullable=false, length=10)
     private String zipcode;
}
```

JPA - Les entités (5/6)

- Quelques annotations (suite) :
  - @Temporal
    - Indique une colonne représentant une date ou une heure
    - @Temporal(TemporalType.DATE) : une date sans l'heure
    - @Temporal(TemporalType.TIME) : une heure sans date
    - @Temporal(TemproralType.TIMESTAMP) : une date avec une heure
  - @Transient
    - Indique un attribut non persistant
    - Optionnel (par défaut tous les attributs sont persistants)

```
@Entity
@Table(name = "t_customer")
public class Customer {
    @Column(name="date_of_birth")
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;

    @Transient
    private Integer age;
}
```

JPA - Les entités (6/6)

- Quelques annotations (suite et fin) :
  - @Embedded et @Embeddable
    - Permet d'englober deux classes entités dans une seule table
    - Les attributs des deux classes sont stockées dans la table

```
@Entity
@Table(name = "t_order")
public class Order {
    @Id
    private Long id;

    @Embedded
     private CreditCard creditCard;
}
```

```
@Embeddable
public class CreditCard {
     private String creditCardNumber;
     private CreditCardType creditCardType;
     private String creditCardExpDate;
}
```

Correspond à la table *t_order*

Plan du cours

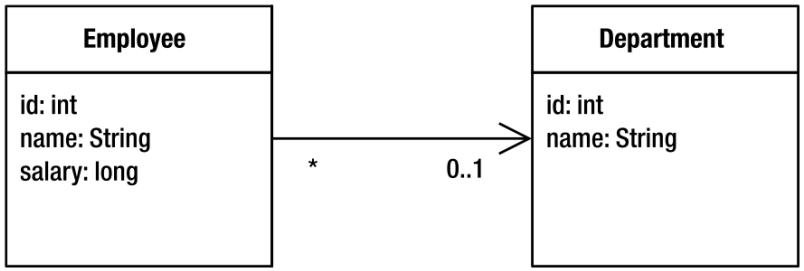
- Communication par composant distribué :
  - Introduction à *Java Persistence API*
    - ✓ Les entités
    - Les relations entre entités

JPA – Les relations entre entités (1/9)

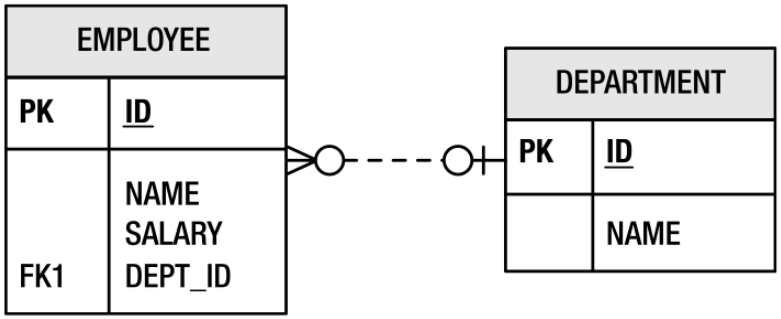
- Les relations entre entités (en bref)
  - Les relations entre entités peuvent être : @*OneToOne*, @*OneToMany*, @*ManyToOne*, @*ManyToMany*
  - Une relation peut être unidirectionnelle ou bidirectionnelle
  - Chaque relation à une direction de navigation donnée par l'entité « propriétaire » de la relation :
    - Relation unidirectionnelle : le propriétaire est l'entité qui voit l'autre entité
    - Relation bidirectionnelle :
      - Le côté non-propriétaire doit indiquer le propriétaire par la valeur *mappedBy* dans le cas des relations @OneToOne, @OneToMany, or @ManyToMany
      - Dans la relation @ManyToOne, le côté « many » est toujours propriétaire
      - Dans la relation @OneToOne, le propriétaire est l'entité/table qui contient la clé étrangère
      - Dans la relation @ManyToMany, le propriétaire peut être l'une ou l'autre
  - La direction de navigation est utilisée par les requêtes (dont les requêtes de stockage et de destruction en cascade)

JPA – Les relations entre entités (2/9)

- @ManyToOne
  - Relation unidirectionnelle : le propriétaire est l'entité qui voit l'autre entité
  - Relation bidirectionnelle : le côté « many » est toujours propriétaire



Employee *est le propriétaire de la relation*



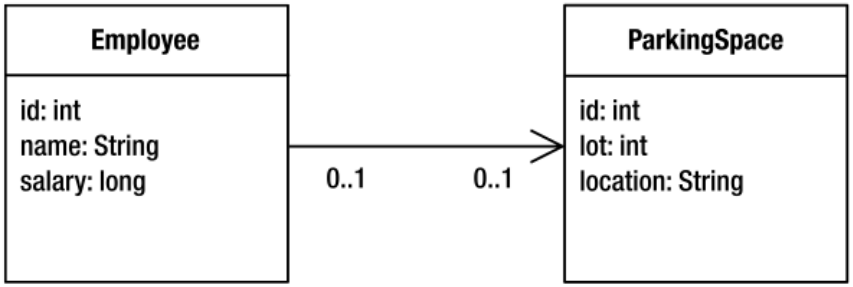
Jointure par clé étrangère

```
@Entity
public class Employee {

    @Id private int id;

    @ManyToOne
    @JoinColumn(name="DEPT_ID")
    private Department department;

}
```

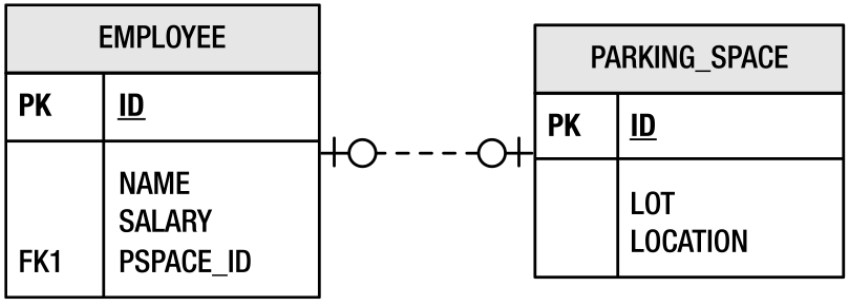
On doit indiquer le nom réel de la colonne

JPA – Les relations entre entités (3/9)

- @OneToOne unidirectionnelle
  - Relation unidirectionnelle : le propriétaire est l'entité qui voit l'autre entité



Employee *est le propriétaire de la relation*

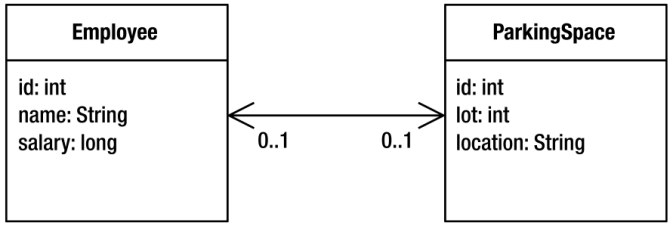


Jointure par clé étrangère

```
@Entity
public class Employee {
   @Id private int id;

   private String name;

   @OneToOne
   @JoinColumn(name= "PSPACE_ID")
   private ParkingSpace parkingSpace;

}
```

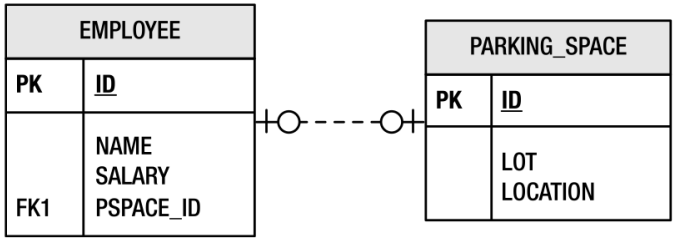
JPA – Les relations entre entités (4/9)

- @OneToOne bidirectionnelle
  - Relation bidirectionnelle : le propriétaire est l'entité/table qui contient la clé étrangère
    - L'annotation @JoinColumn est du côté du propriétaire
    - La valeur *mappedBy* est du côté inverse du propriétaire



Employee *est le propriétaire de la relation*

```
@Entity
public class Employee {
   @Id private int id;

   private String name;

   @OneToOne
   @JoinColumn(name= "PSPACE_ID")
   private ParkingSpace parkingSpace;
}
```

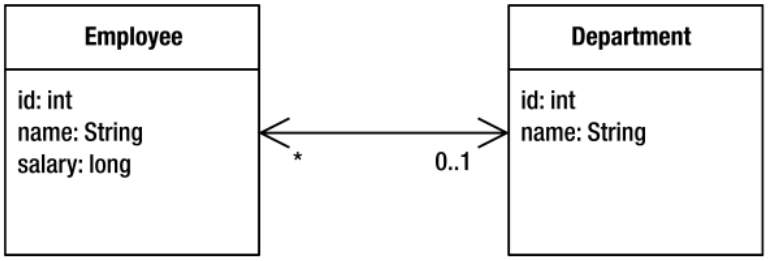


Jointure par clé étrangère dans la table Employee

```
@Entity
public class ParkingSpace {
   @Id private int id;

   private int lot;
   private String location;

   @OneToOne(mappedBy="parkingSpace")
   private Employee employee;
}
```

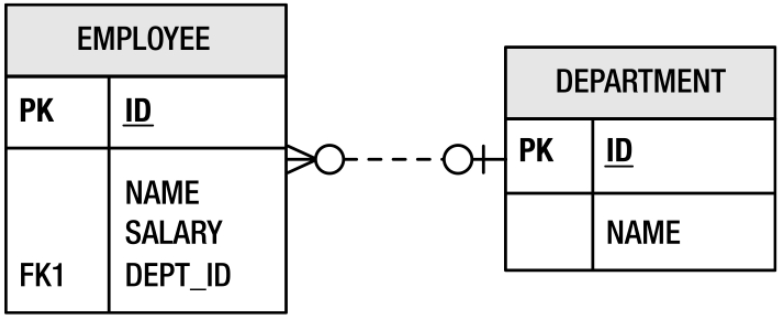
JPA – Les relations entre entités (5/9)

- @OneToMany bidirectionnelle
  - Relation bidirectionnelle : le côté « many » est toujours propriétaire



Employee *est le propriétaire de la relation*

```
@Entity
public class Employee {
   @Id private int id;

   @ManyToOne
   @JoinColumn(name="DEPT_ID")
   private Department department;
}
```

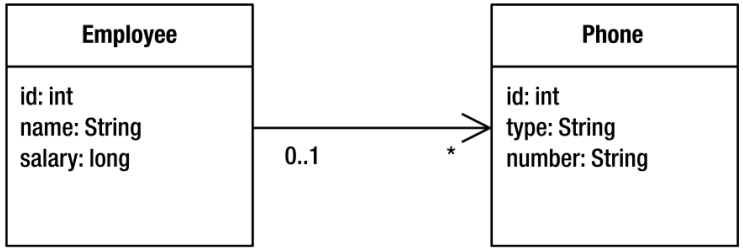


Jointure par clé étrangère ou par table de jointure

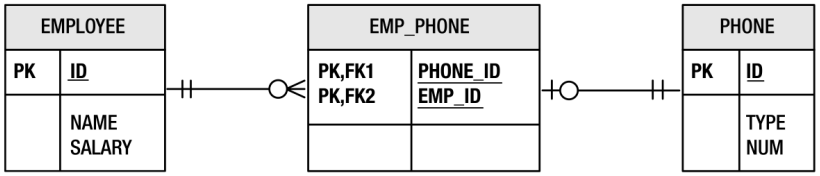
```
@Entity
public class Departement {
   @Id private int id;

   private String name;

   @OneToMany(mappedBy="department")
   private Collection<Employee> employees;
}
```

JPA – Les relations entre entités (6/9)

- @OneToMany unidirectionnelle
  - Relation unidirectionnelle : le propriétaire est l'entité qui voit l'autre entité



Employee *est le propriétaire de la relation*

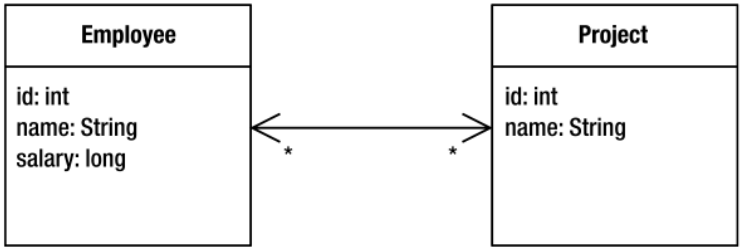


Jointure obligatoire par table de jointure

```
@Entity
public class Employee {
   @Id private int id;

   private String name;

   @OneToMany
   @JoinTable(name="EMP_PHONE",
         joinColumns={@JoinColumn(name="EMP_ID")},
         inverseJoinColumns={@JoinColumn(name="PHONE_ID")})
    private Collection<Phone> phones;
}
```

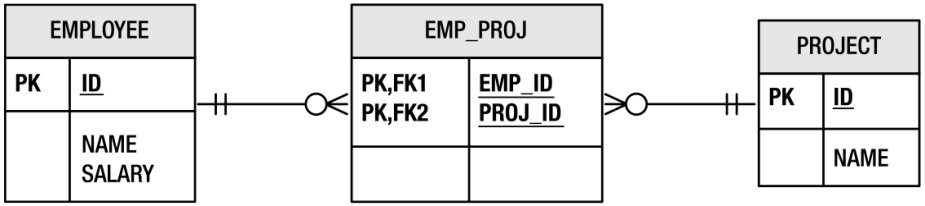
JPA – Les relations entre entités (7/9)

- @ManyToMany
  - Relation unidirectionnelle : le propriétaire est l'entité qui voit l'autre entité
  - Relation bidirectionnelle : le propriétaire peut être l'une ou l'autre entité



Employee *est le propriétaire de la relation*

```
@Entity
public class Employee {
    @Id private int id;

    private String name;

    @ManyToMany
    @JoinTable(name="EMP_PROJ",
            joinColumns={@JoinColumn(name="EMP_ID")},
            inverseJoinColumns={@JoinColumn(name="PROJ_ID")})
    private Collection<Project> projects;
}
```

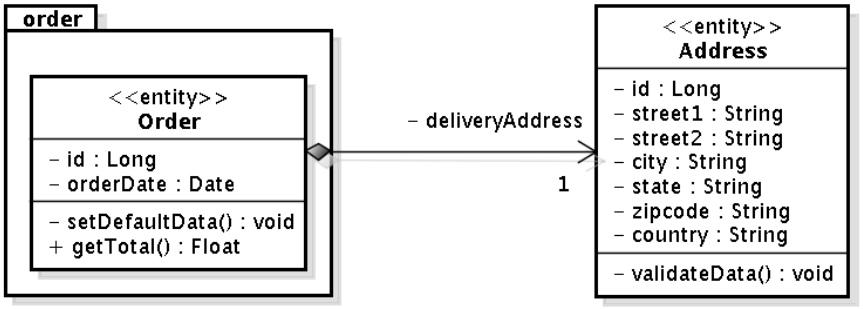


Jointure obligatoire par table de jointure

```
@Entity
public class Project {
    @Id private int id;

    private String name;

    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
}
```

JPA – Les relations entre entités (8/9)

- Rendre la relation obligatoire : mettre *nullable=false* dans *@JoinColumn*




```
@Entity
@Table(name = "t_order")
public class Order {
    @Id @GeneratedValue private Long id;

    @OneToOne
    @JoinColumn(name = "address_fk", nullable = false)
    private Address deliveryAddress;
}
```

- Chargement d'une association :
  - La valeur *fetch* permet de spécifier si les objets associés se chargent directement (EAGER) ou de manière différée (LAZY)


```
@Entity
public class Category {
    @OneToMany(mappedBy="category", fetch=FetchType.LAZY)
    private List<Product> products;
}
```

```
@Entity
public class Product {
    @ManyToOne(fetch=FetchType.EAGER)
    private Category category;
}
```

JPA – Les relations entre entités (9/9)

- Ordonner une collection d'entités :
  - L'annotation @OrderBy prend en paramètre les noms des attributs sur lesquels on souhaite effectuer un tri, ainsi que le mode optionnel (ASC ou DESC)

```
@Entity
public class Category {
    @OneToMany(mappedBy="category", fetch=FetchType.LAZY)
    @OrderBy("name ASC")
    private List<Product> products;
}
```

```
@Entity
public class Product {
    private String name;
    @ManyToOne(fetch=FetchType.EAGER)
    private Category category;
}
```

- Action en cascade :
  - Propager une opération sur une entité à ses entités associées
  - Par exemple, une suppression d'une entité doit entraîner la suppression des entités associées
  - L'attribut *cascade* peut prendre les valeurs : DETACH, MERGE, PERSIST, REFRESH, REMOVE, ALL (cf. cycle de vie des entités ci-après)

```
@Entity
public class Category {
    @OneToMany(cascade={CascadeType.REMOVE})
    private List<Product> products;
}
```

```
@Entity
public class Customer {
    @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "address_fk", nullable = true)
    private Address homeAddress;
}
```

Plan du cours

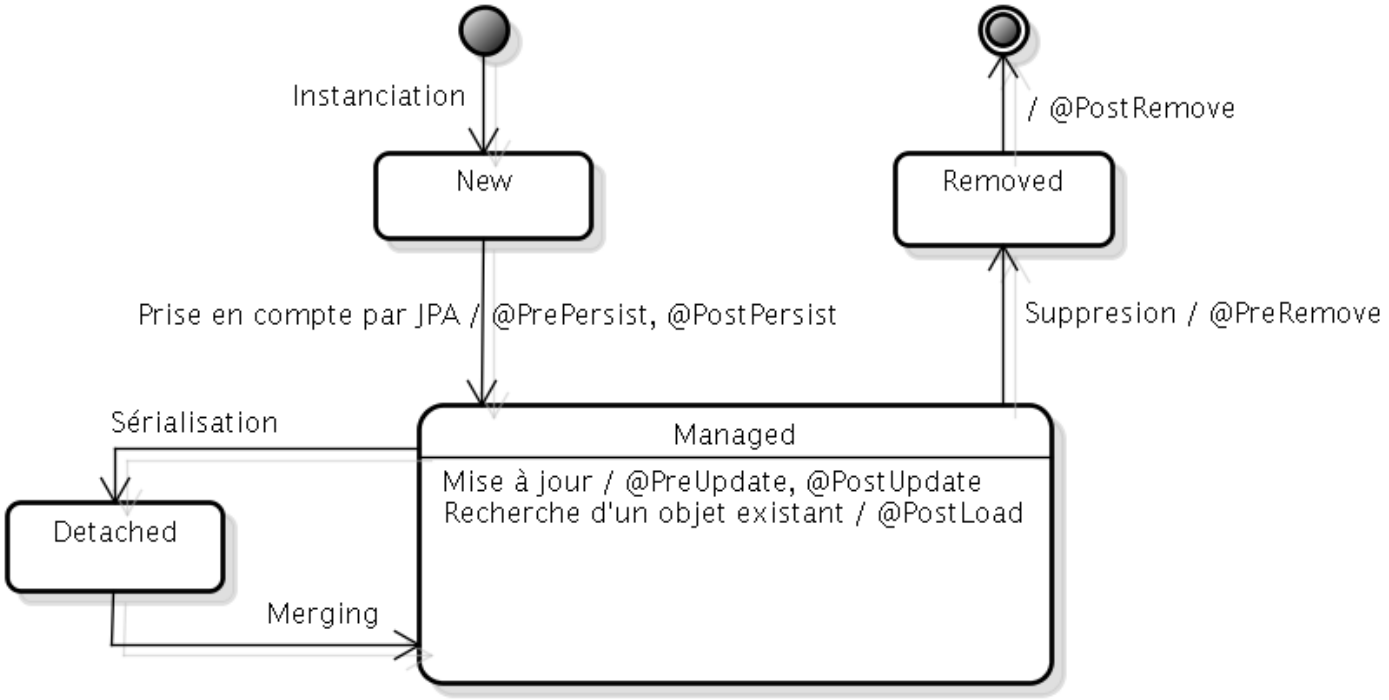
- Communication par composant distribué :
  - Introduction à *Java Persistence API*
    - ✓ Les entités
    - Les relations entre entités
    - Le cycle de vie des entités

JPA – Le cycle de vie des entités (1/3)

- Une entité est dans l'un des quatre états suivants :
  - *New* : l'entité n'a pas d'identifiant et n'est pas liée à JPA
  - *Managed* : l'entité a un identifiant et est liée à JPA
  - *Detached* : l'entité a un identifiant mais n'est plus liée à JPA
  - *Removed* : l'entité va être détruite de la base de données
- Une entité est *New* lorsqu'elle vient d'être crée
- Une nouvelle entité devient *Managed* quand elle est stockée dans la base par une opération de persistance
- Une entité *Managed* devient *Removed* par une opération de destruction
- Une entité *Managed* devient *Detached* lorsqu'elle est sérialisée puis désérialisée
- Après modification, une entité est synchronisée avec la base lorsque la transaction est validée ou par appel explicite à l'opération *flush* du gestionnaire de données (*EntityManager*)

JPA – Le cycle de vie des entités (2/3)

- Avant ou après un changement d'état, il y a possibilité d'effectuer certains méthodes annotées :
  - Avant insertion en base @PrePersist, après insertion @PostPersist
  - Avant suppression @PreRemove, après suppression @PostRemove
  - Avant mise à jour @PreUpdate, après mise à jour @PostUpdate
  - Après chargement @PostLoad



JPA – Le cycle de vie des entités (3/3)

- Dans l'exemple, l'entité Category valide ses données avant son insertion en base (@PrePersist) ou sa mise à jour (@PreUpdate). Si les données ne sont pas valides, une exception est lancée :

```
@Entity
public class Category {
    @Id @GeneratedValue
    private Long id;

    private String name;
    private String description;

    @PrePersist @PreUpdate
    private void validateData() {
        if (name == null || "".equals(name))
            throw new ValidationException("Invalid name");
        if (description == null || "".equals(description))
            throw new ValidationException("Invalid description");
     }
}
```

Plan du cours

- Communication par composant distribué :
  - Introduction à *Java Persistence API*
    - ✓ Les entités
    - ✓ Les relations entre entités
    - ✓ Le cycle de vie des entités
    - Le gestionnaire de persistance et les opérations CRUD

JPA – EntityManager et les op. CRUD (1/3)

- Le gestionnaire de contexte de persistance
  - Un contexte de persistance est un ensemble de classes entités liées à une même source de données et les opérations/transactions que l'on peut faire sur ces entités
  - Chaque *EntityManager* gère un contexte de persistance
  - Les deux modes d'utilisation d'un contexte de persistance :
    - Gestion par le conteneur
      - Le conteneur fait une injection automatique d'un même gestionnaire de contexte pour propager une transaction en cours dans tous les composants impliqués pas la transaction
      - Il faut obtenir un *EntityManager* avec *@PersistenceContext*
    - Gestion par l'application
      - Le programmeur gère le contexte de persistance
      - Il n'y a pas de propagation du gestionnaire de contexte
      - Cela permet d'effectuer des opérations isolées
      - Il faut obtenir une fabrique *EntityManagerFactory* avec *@PersistenceUnit*

JPA – EntityManager et les op. CRUD (2/3)

- L'*EntityManager* est initialisé avec une unité de persistance
- Une unité de persistance *PU* est une description XML indiquant :
  - Le provider JPA (implémentation de l'API JPA)
  - Une source de données (en fait un accès JNDI à cette source)
  - Les propriétés du provider JPA
- Une source de données *DS* représente la connexion physique à la base
- Une *DS* peut être définie dans la console du serveur (GlassFish)

```
@Stateless
public class CustomerBean implements CustomerRemote {

    @PersistenceContext (unitName = "petstorePU")
    private EntityManager em;

    public Customer createCustomer(Customer customer,
                                   Address homeAddress) {
        customer.setHomeAddress(homeAddress);
        em.persist(customer);
        return customer;
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
    <persistence-unit name="petstorePU" transaction-type="JTA">
        <provider>oracle.toplink.essentials.PersistenceProvider</provider>
        <jta-data-source>jdbc/petstoreDS</jta-data-source>
        <properties>
            <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
        </properties>
    </persistence-unit>
</persistence>
```

JPA – *EntityManager* et les op. CRUD (3/3)

- Les opérations *CRUD* :
  - Elles sont prises en charge par un *EntityManager*
  - Elles agissent sur l'état des entités :
    - Opération Create : sauvegarde d'une entité avec *persist(objet_entité)*
    - Opération Read : charger une entité avec *find(nom_classe, identifiant)*
    - Opération Update : mise à jour de la base
      - Automatique si l'entité est dans l'état *Managed*
      - Manuelle si l'entité est *Detached* : *merge(objet_entité_détaché)*
    - Opération Delete : suppression de la base avec *remove(objet_entité)*
  - *flush()* force la synchronisation de l'entité vers la base
  - *refresh(objet_entité)* restaure depuis la base vers l'entité

```
@PersistenceContext EntityManager em;
...
public LineItem createLineItem(Order order, int quantity) {
   LineItem li = new LineItem(order, quantity);
   order.getLineItems().add(li);
   em.persist(li);
   return li;
}
```

```
public void removeOrder(Integer orderId) {
   try {
      Order order = em.find(Order.class, orderId);
      em.remove(order);   // Cascade remove
   }
...
```

[Source : The Java™ EE 5 Tutorial Third Edition]

Plan du cours

- Communication par composant distribué :
  - Introduction à *Java Persistence API*
    - ✓ Les entités
    - ✓ Les relations entre entités
    - ✓ Le cycle de vie des entités
    - ✓ Le gestionnaire de persistance et les opérations CRUD
    - Le langage de requête de JPA : JPQL

JPA – JPQL (1/2)

- JPQL : Java Persistence Query Language
- Langage de requêtes (inspiré de SQL) manipulant des objets et retournant des objets
- Création et exécution d'une requête *javax.persistence.Query* :

```
Query query = em.createQuery("SELECT c FROM Customer c WHERE c.login=:param");
query.setParameter("param", login);
Customer customer = (Customer) query.getSingleResult();
```

```
public List findWithName(String name) {
    return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE :custName")
                .setParameter("custName", name)
                .setMaxResults(10)
                .getResultList();
}
```

- Requêtes statiques avec l'opération *createNamedQuery* :

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
…
customers = em.createNamedQuery("findAllCustomersWithName").setParameter("custName", "Smith").getResultList();
```

JPA – JPQL (2/2)

- Langage très riche :
  - Filtrer les résultats (IN, NOT, EXIST, LIKE, IS NULL...)
  - Ne pas avoir de doublon (DISTINCT)
  - Contrôler la taille des listes (IS EMPTY, IS NOT EMPTY, CONTAINS...)
  - Fonctions sur les chaines (LOWER, UPPER, CONCAT...)
  - Fonctions sur les nombres (ABS, SQRT, MOD...)
  - Fonctions sur les ensembles (COUNT, MAX, MIN, SUM)
  - Ordonner les résultats (ORDER BY)
  - *Etc.*

```
Query query = em.createQuery("SELECT c FROM Customer c ORDER BY c.lastname");
List<Customer> customers = query.getResultList();
```

```
Query query = em.createQuery("SELECT i FROM Item i WHERE
                                    UPPER(i.name) LIKE :keyword OR
                                    UPPER(i.product.name) LIKE :keyword
                                    ORDER BY i.product.category.name, i.product.name");
query.setParameter("keyword", "%"+keyword.toUpperCase()+"%");
```

Plan du cours

- Communication par composant distribué :
  - Introduction à *Java Persistence API*
    - ✓ Les entités
    - ✓ Les relations entre entités
    - ✓ Le cycle de vie des entités
    - ✓ Le gestionnaire de persistance et les opérations CRUD
    - ✓ Le langage de requête de JPA : JPQL
    - Les transactions

JPA – Les transactions (1/6)

- Les transactions
  - Une transaction est un ensemble d'opérations vues comme une seule opération atomique : si une seule des opérations échoue alors la transaction échoue

begin transaction
débiter un compte n°X de n euros
créditer un compte n°Y de n euros
mise à jour du fichier de trace
commit transaction

  - Une transaction suit les propriétés ACID :
    - Atomicité : toutes les opérations sont valides ou aucune n'est valide
    - Consistance : la base se trouve dans un état cohérent avant et après la transaction même si elle échoue
    - Isolation : les transactions concurrentes ne voient pas de résultats partiels car chaque transaction est isolée des autres (transactions en séquence)
    - Durabilité : garantit que les mises à jour de la base peuvent survivre à une panne (on utilise un fichier de log pour faire des *undos* et revenir dans l'état avant la panne)
  - Les transactions peuvent être gérées par le conteneur ou le composant

JPA – Les transactions (2/6)

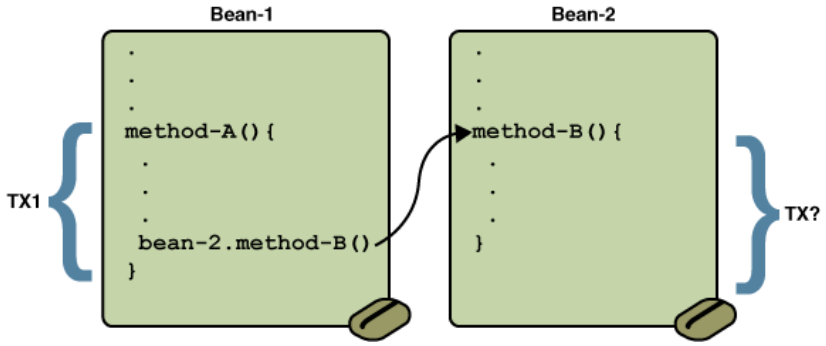
- Transaction gérée par l'application
  - Le programmeur gère explicitement toute la transaction
  - Il peut utiliser au choix :
    - *JTA* (Java Transaction API) :
      - Elle permet de faire des transactions impliquant plusieurs bases hétérogènes
      - Le programmeur utilise les méthodes *begin*, *commit* et *rollback* de l'interface *javax.transaction.UserTransaction*
    - Un driver *JDBC* : il est spécifique à un seul type de base !

- Transaction gérée par le container
  - Le conteneur gère implicitement toute la transaction
  - Le développeur et/ou l'assembleur décrit la gestion transactionnelle d'un composant ou de ses méthodes métiers individuellement
  - Il utilise l'annotation @*TransactionAttribute(*valeur*)*

JPA – Les transactions (3/6)

- Transaction gérée par le container (suite)
  - Les valeurs permissent sont :
    - REQUIRED :
      - Nécessite une transaction
      - rejoint la transaction en cours ou le container en crée une nouvelle
    - REQUIRES_NEW :
      - Nécessite une nouvelle transaction
      - si une transaction existe, elle est suspendue puis reprise après
    - MANDATORY :
      - Nécessite une transaction déjà existante
      - S'il n'y a pas de transaction *TransactionRequiredException* est levée
    - NOT_SUPPORTED :
      - Ne supporte pas les transactions
      - Si une transaction existe elle est suspendue puis reprise après
    - SUPPORTS :
      - Peut fonctionner avec si elle existe ou sans si elle n'existe pas
    - NEVER :
      - Ne supporte aucune transaction
      - *javax.persistence.PersistenceException* est levée si une transaction existe

JPA – Les transactions (4/6)



method-B Transaction Attribute	Bean-1 Transaction	Bean-2 Transaction
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Mandatory	None	Error
	T1	T1
NotSupported	None	None
	T1	None
Supports	None	None
	T1	T1
Never	None	None
	T1	Error

[Source : The Java™ EE 5 Tutorial Third Edition]

JPA – Les transactions (5/6)

- Exemple de transaction gérée par le conteneur :

Annotation indiquant au conteneur que toutes les méthodes de la classe utilisent le mode transactionnel REQUIRED

```
@Stateless
@TransactionAttribute(value=TransactionAttributeType.REQUIRED)
public class CustomerBean implements CustomerRemote {
    @PersistenceContext(unitName = "petstorePU")
    private EntityManager em;

    public Customer createCustomer(Customer customer, Address homeAddress {
        customer.setHomeAddress(homeAddress);
        em.persist(customer);
        return customer;
    }

    @TransactionAttribute(value = TransactionAttributeType.NEVER)
    public Customer findCustomer(final Long customerId) {
        Customer customer = em.find(Customer.class, customerId);
        return customer;
    }
}
```

Cette méthode utilise le mode transactionnel défini pour la classe REQUIRED

Cette méthode re-définit sa politique transactionnel. Elle ne tient pas compte du mode REQUIRED mais utilise NEVER

JPA – Les transactions (6/6)

- Exemple de transaction gérée par l'application :

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA {
   public static void main(String[] argv) {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
      EntityManager em = emf.createEntityManager();

      EntityTransaction transac = em.getTransaction();

      try {
         transac.begin();

         Personne personne = em.find(Personne.class, 4);
         if (personne == null)
            System.out.println("Personne non trouvée");
         else
            em.remove(personne);

         transac.commit();
      } catch (Exception e) { transac.rollback() }

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

Obtention d'une fabrique en indiquant une unité de persistance

Obtention d'un gestionnaire à partir de la fabrique

Création d'une transaction

Début de la transaction

Opérations CRUD sur le gestionnaire

Validation de la transaction

Annulation de la transaction