Persistance

- modélisation -

- gestion -

Persistance en Java

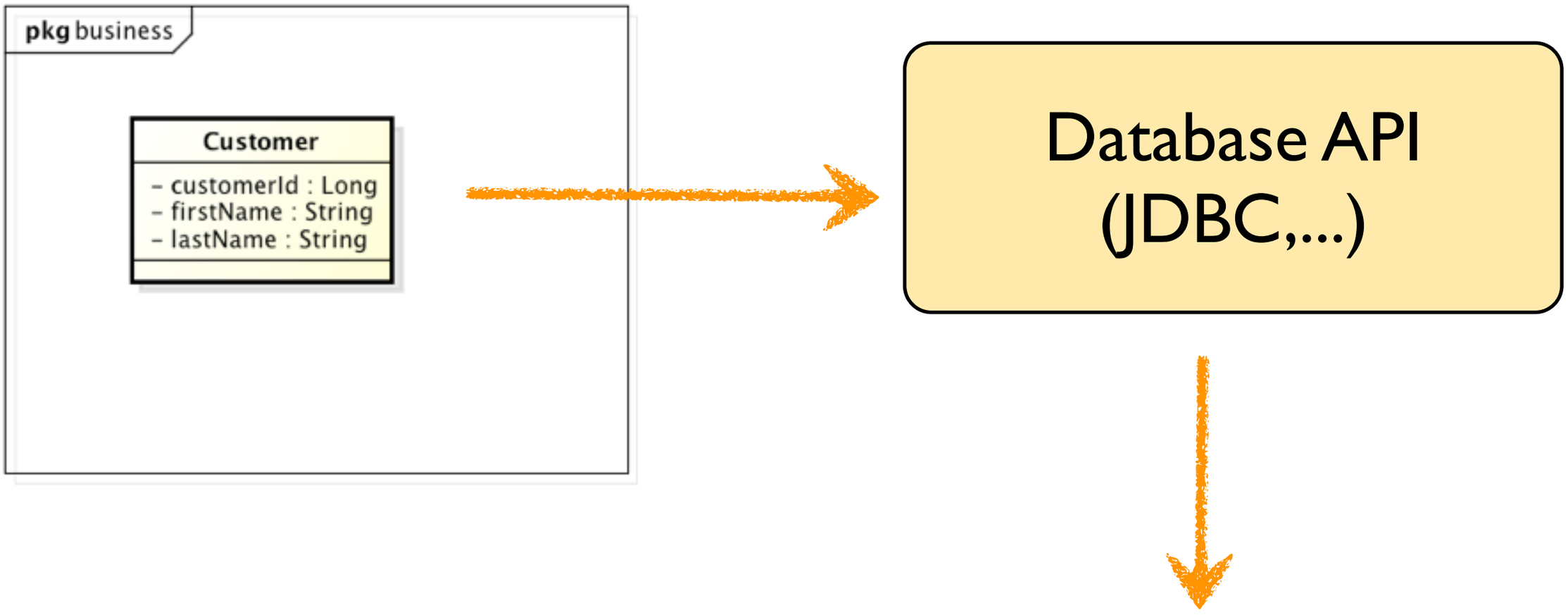
Comment assurer la persistance des objets ?

- par sérialisation
- en utilisant une base de données relationnelle
- en utilisant une base de données objet
- en utilisant un mapper Objet/BD

Persistance par sérialisation

- Inconvénients
  - maintenance
  - scalabilité
  - pas de requêtes complexes
  - ...

Persistance avec BD



CUSTOMER_ID	CUSTOMER_FIRSTNAME	CUSTOMER_LASTNAME

Persistance avec BD

- utilisation d'une API pour l’accès à la BD
  - écriture des requêtes et des mises à jours à la main
  - transformation des classes pour garantir la correspondance

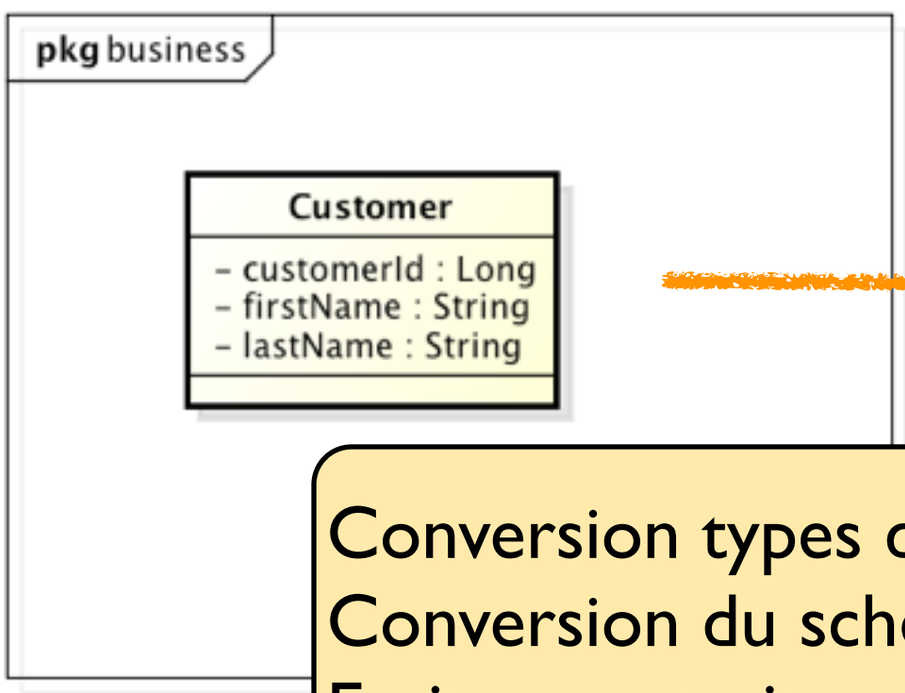
Persistance avec BD

- Inconvénients
  - gestion manuelle de la persistance
  - gestion manuelle des transformations des types
  - gestion des changements (dans les classes / schémas)
  - gestion des transactions

Persistance avec BD objet

- Bases de données basées sur un modèle objet
  - un seul modèle
  - conversion plus simple des données
  - modèle navigationnel (évite les jointures)
  - bonnes performances mais mauvaise scalabilité

Persistance par mapping objet/BD



mapping automatique

EclipseLink
Hibernate
...

Conversion types de données (SQL/Java)
Conversion du schéma (relationnel/objet)
Ecriture et traitement des requêtes

CUSTOMER_ID	CUSTOMER_FIRSTNAME	CUSTOMER_LASTNAME

Persistance JPA 2

- JPA (Java persistence API) est la partie de la spécification EJB qui concerne la persistance des composants dans une base de données relationnelle
- JPA peut être utilisé par toutes les applications Java, même en dehors de Java EE
- API ORM (Object-Relational Mapping) standard pour la persistance des objets Java

Comment ca marche ?

POJOs

@Annotations

Persistence manager

données

Les entités

- les classes dont les instances peuvent être persistantes
- représentent sous forme d'objets des données situées dans une base de donnée
- représentées par des POJOs
  - identité unique et visible
  - état persistent et visible
  - non accessible à distance

```
@Entity
public class Customer implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long customerId;

    private String firstName;
    private String lastName;

    public Customer() {
    }

    public Customer(String firstN, String lastN) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
...
}
```

entité

primary key

génération automatique

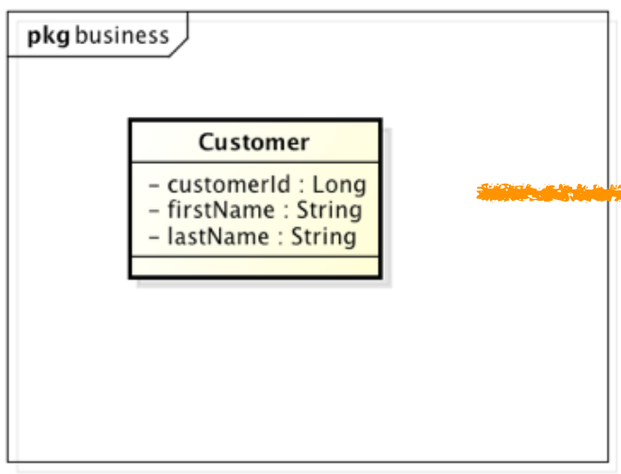
Les annotations

- identifient les entités (`@Entity`)
- décrivent le modèle du domaine (associations, multiplicités,...)
- décrivent les ORM (tables, colonnes, ...)

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "CUSTOMER_ID")
    private Long customerId;
    @Column(name = "CUSTOMER_FIRSTNAME")
    private String firstName;
    @Column(name = "CUSTOMER_LASTNAME")
    private String lastName;
...
}
```

mapping table

mapping colonnes



CUSTOMERS

CUSTOMER_ID	CUSTOMER_FIRSTNAME	CUSTOMER_LASTNAME

Le fournisseur de persistance

- système chargé de la communication avec le SGBD
  - stockage, chargement, mise à jour
- **gère la synchronisation des objets avec la base de données**
- **EclipseLink → référence spécification JPA 2.0** (`http://www.eclipse.org/eclipselink/`)
- Hibernate Entity Manager, OpenJPA, BEA Kodo

Le fournisseur de persistance

- il est nécessaire d'indiquer au fournisseur de persistance comment il peut se connecter à/ utiliser la base de données
  - fichier persistence.xml situé dans un répertoire META-INF dans le `classpath`

Exemple (dans container)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="...>
  <persistence-unit name="glaPU"
                    transaction-type="JTA">
    <jta-data-source>jdbc/gla</jta-data-source>
    <class>persistence.Customer</class>
    <properties>
      <property name="eclipselink.ddl-generation"
                value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

nom

la ressource de données

création de tables si inexistantes

Le manager de persistance

- fournit les méthodes pour la gestion des entités
  - les rendre persistantes
  - les supprimer de la base de données
  - retrouver leurs valeurs dans la base
  - les mettre à jour

CRUD (Create, Read, Update, Delete)

```
@Stateless(name = "CustomerManager")
public class CustomerManagerBean implements
                CustomerManager {

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

    public Customer addCustomer(Customer customer) {
        em.persist(customer);
        return customer;
    }

    public Customer firstCustomer() {
        Customer c = em.find(Customer.class, 1L);
        return customer;
    }
}
```

injection manager persistance

création

recherche par clé

Comment ca marche ?

- Modéliser le domaine (entités, associations)
- Object-relational mapping (ORM)
- Gestion des entités avec le `EntityManager`
- Rechercher les entités

Modéliser le domaine

- entités
- associations

Les entités

- des POJOs qui peuvent être persistants

```
@Entity
public class Customer {
```

- avec une identité unique et visible

```
@Id
@GeneratedValue(strategy=enumerationType.AUTO)
private Long customerId;
```

Règles `Entity`

- doit posséder un attribut qui représente la clé primaire dans la BD (@Id)
- un constructeur sans arguments
- ne doit pas être `final, interface,` classe interne
- aucune méthode ou champ persistant ne doit être final
- si passée par en paramètre d'une méthode, elle doit implémenter Serializable

Persister les données

- l'état persistant d'un objet est maintenu en utilisant ses attributs ou ses propriétés
- 2 types de persistance
  - par champ (***field-based access***)
  - par propriété (***property-based access***)
- **convention de nommage :** *setter* **and** *getter*
  - getProperty()
  - setProperty(PropertyType property)

Conditions

- **field-based access**
  - la variable d'instance *peut* ne pas avoir de getter et de setter
- **property-based access**
  - les *getter* et *setter* sont tous les deux obligatoires

Type d'accès

- défini par la position des annotations dans la classe entité

```
@Id
private Long customerId; // field-accesss
```

- annotation `@Access` :
  - `@Access(AccessType.PROPERTY)`
  - `@Access(AccessType.FIELD)`

Quel type d'accès ?

- En programmation → encanpsulation
  - utiliser les setter/getter
- En JPA c'est différent → accès par champ
  - pas besoin des setters et getters pour toutes les propriétés
  - des setter/getter sans champs correspondant (à déclarer **`@Transient`**)
  - possible d'utiliser **`@Access(PROPERTY)`** sur certains attributs

Attributs persistants

- par défaut, tous les attributs sont persistants (peut être spécifié explicitement avec `@Basic`)
- spécifier des attributs non-persistants
  - `@Transient`
  - `transient`

Attributs persistants

```
@Entity
public class Customer implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long customerId;
    private String firstName;
    private String lastName;
    ...
    transient private String completeName;
    @Transient
    private int age;
...
}
```

attributs non-persistants

Types persistants

- Types primitifs : `int`, `double`,...
- Wrappers : `Integer`, `Double`, ...
- `String`
- Java API `Serializable` : `BigInteger`, `Date`, ...
- `Serializable` définis par l'utilisateur
- Enumérations
- Tableaux : `byte[]`, `char[]`
- Collections : `Set<Customer>`
- `@Embeddable`

Identité des entités

- une entité doit avoir un attribut qui l'identifie d'une manière unique
  - correspond à la clé primaire dans la table associée
  - l'attribut doit être défini dans l'entité racine d'une hiérarchie d'héritage (pour toute la hiérarchie d'héritage)

Identité des entités

- attribut
  - `@Id`
- composite (pas recommandé)
  - `@EmbeddedId`
  - `@IdClass`

Types de la clé

- type primitif Java : `int`, ...
- wrapper type primitif : `Integer`, ...
  - `java.lang.String`
  - `java.util.Date`, `java.sql.Date`
- eviter les types numériques non entiers (`Double`, ...), `TimeStamp`, ...

Clé composite

- entité identifiée par plusieurs attributs
- clé primaire représentée par une classe Java dont les attributs correspondent aux attributs identifiant l'entité
  - `public`
  - constructeur sans paramètre
  - `Serializable`
  - redéfinir `equals` et `hashcode`

@IdClass

```
@Entity
@IdClass(CustomerPK)
public class Customer {
     @Id  private String firstName;
     @Id  private String lastName;
     ...
}

public class CustomerPK implements Serializable{
     private String firstName;
     private String lastName;
     public CustomerPK() {}
     public boolean equals(Object other) { ... }
     public int hashCode(){ return super.hashCode();}
}
```

attributs clé

- redondance

+ le modèle du domaine reste clair

@EmbeddedId

```
@Entity
public class Customer {
     @EmbeddedId  private CustomerPK customerPK;
     ...
}

@Embeddable
 public class CustomerPK{
     private String firstName;
     private String lastName;
     public CustomerPK() {}
     public boolean equals(Object other) { ... }
     public int hashCode(){ return super.hashCode();}
 }
```

même type d'accès (AccessType)

+ non-redondance
- modèle du domaine moins clair

`@Embeddable`

- objets persistants mais qui ne peuvent pas avoir une identité (dans la BD)
- identifiés par les entités qui les englobent (i.e. partage l'identité avec les entités les englobant) en utilisant l'annotation `@Embedded`
- leurs attributs sont sauvegardés dans la table associée à l'entité englobante
- leurs associations se font entre l'entité englobante et l(es) autre(s) entité(s)
- une instance ne peut pas être référencée par plusieurs entités différentes

```
@Entity
public class Customer implements Serializable {
     @Id
     private Long customerId;
     private String firstName;
     private String lastName;
     ...
     @Embedded
     private Address address;
     ...
}

@Embeddable
public class Address {
   private String street;
   private String city;
   private String zipCode;
   private String country;
...
}
```

même type d'accès (`AccessType`)

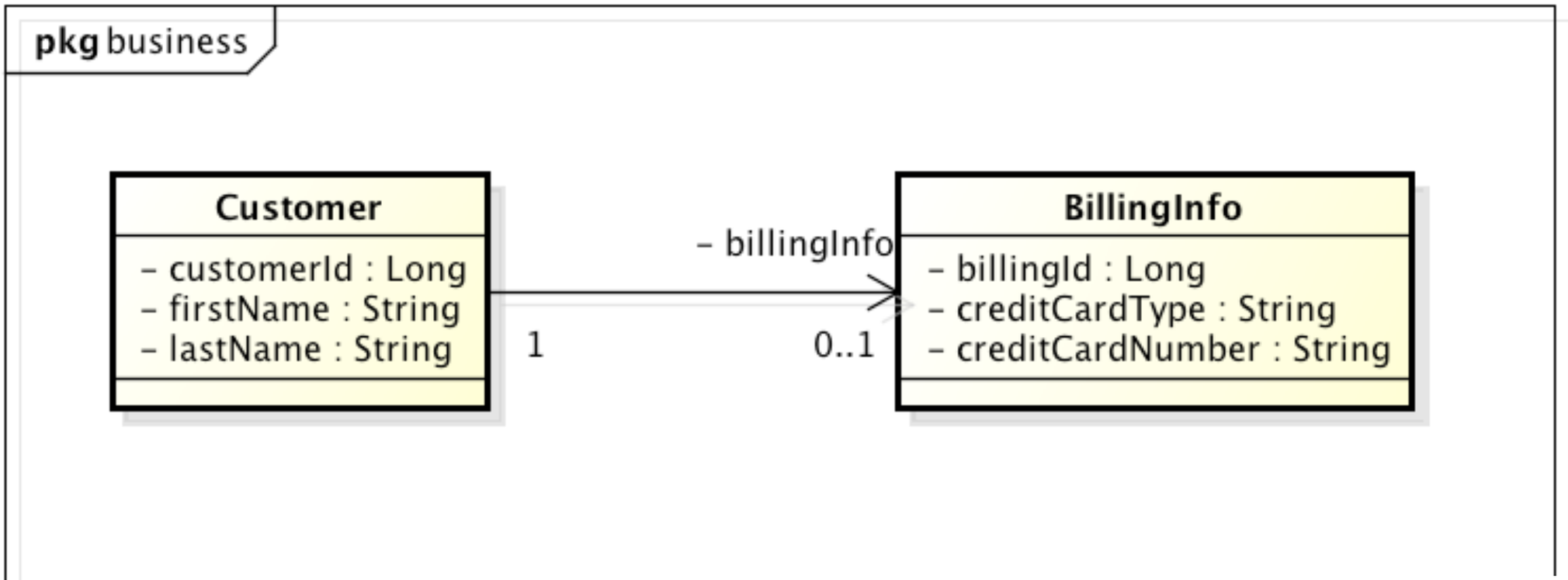
Les associations

- Comment transformer des relations entre classes en des relations entre tables ?

Types de relations

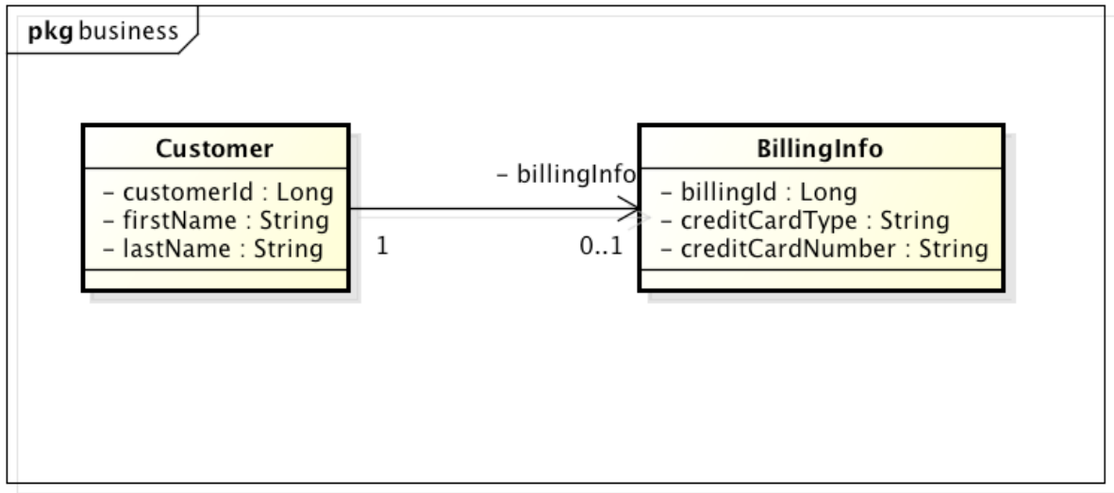
- `@OneToOne`
- `@OneToMany`
  - `@ManyToOne`
  - `@ManyToMany`
- unidirectionnelle ou bidirectionnelle

One-to-one



- Représentée typiquement par une clé étrangère dans une BD

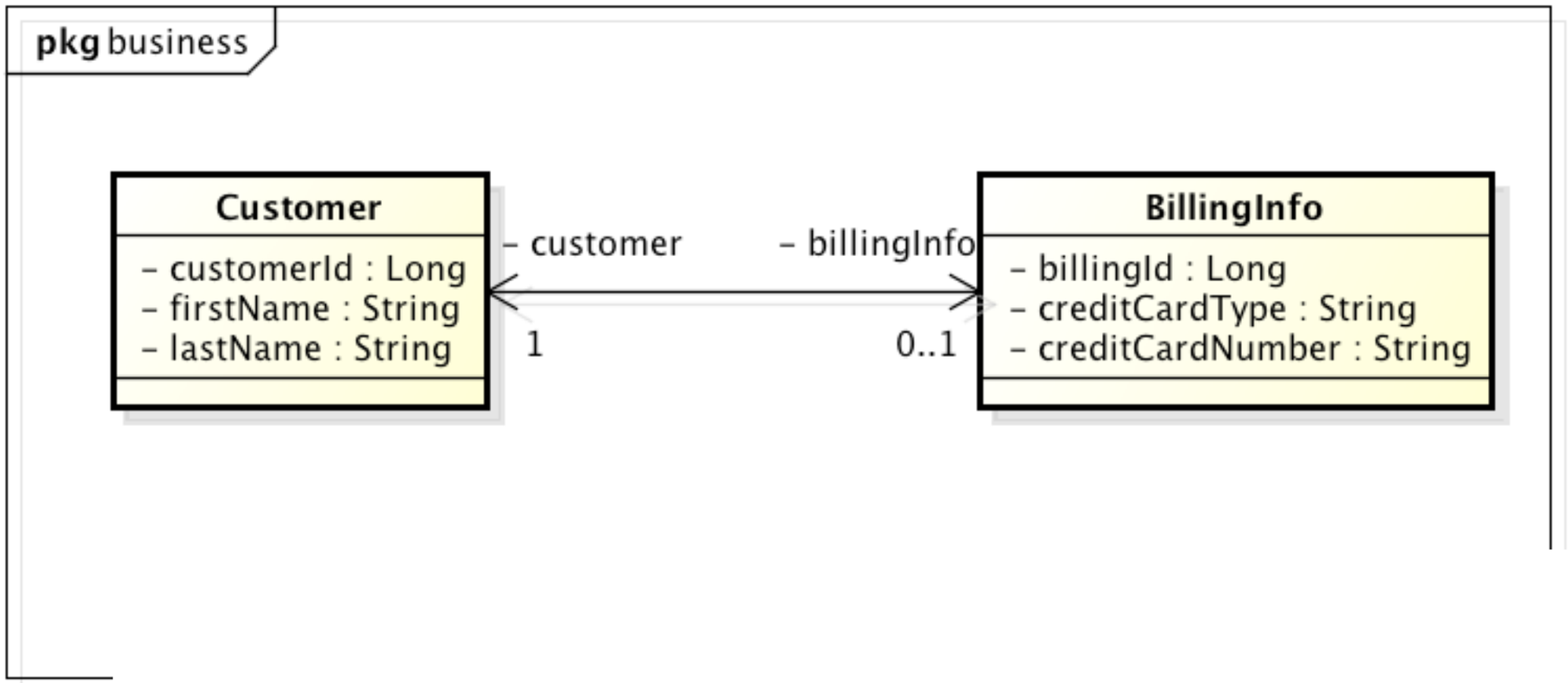
```java
@Entity
public class Customer {
    @Id
    private Long customerId;
    ...
    @OneToOne
    private BillingInfo billingInfo;
...
}
```



tout attribut de type entité dans une entité est considéré comme représentant une association one-to-one

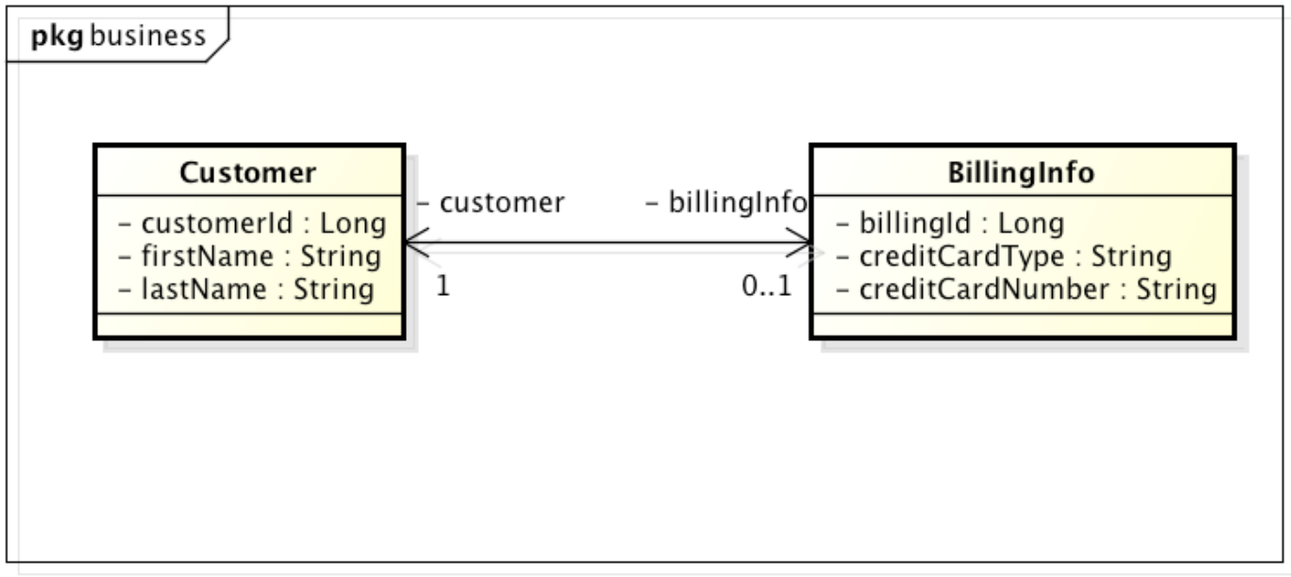
```java
@Entity
public class BillingInfo {
    @Id
    private Long billingId;
    private String creditCardType;
    private String creditCardNumber;
...
}
```

One-to-one bidirectionnelle



- Exemple : pour avertir le Customer si la carte expire

```java
@Entity
public class Customer {
   @Id
   private Long customerId;
   ...
   @OneToOne
   private BillingInfo billingInfo;
...
}
```




```java
@Entity
public class BillingInfo {
   @Id
   private Long billingId;
   private String creditCardType;
   @OneToOne(mappedBy="billingInfo", optional="false");
   private Customer customer;
...
}
```

Customer est le *propriétaire* de la relation

BillingInfo ne peut pas exister sans un Customer

Interface

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
}
```

contrôle ce qui se passe avec les données quand la relation est altérée ou effacée

spécifie quand et comment le champs correspondants sont remplis

```java
@Entity
public class Customer {
   @Id
   private Long customerId;
   ...
   @OneToOne (cascade = CascadeType.ALL)
   private BillingInfo billingInfo;
...
}
```

toutes les opérations sont propagées

```java
@Entity
public class BillingInfo {
   @Id
   private Long billingId;
   private String creditCardType;
   @OneToOne(mappedBy="billingInfo",optional="false");
   private Customer customer;
...
}
```

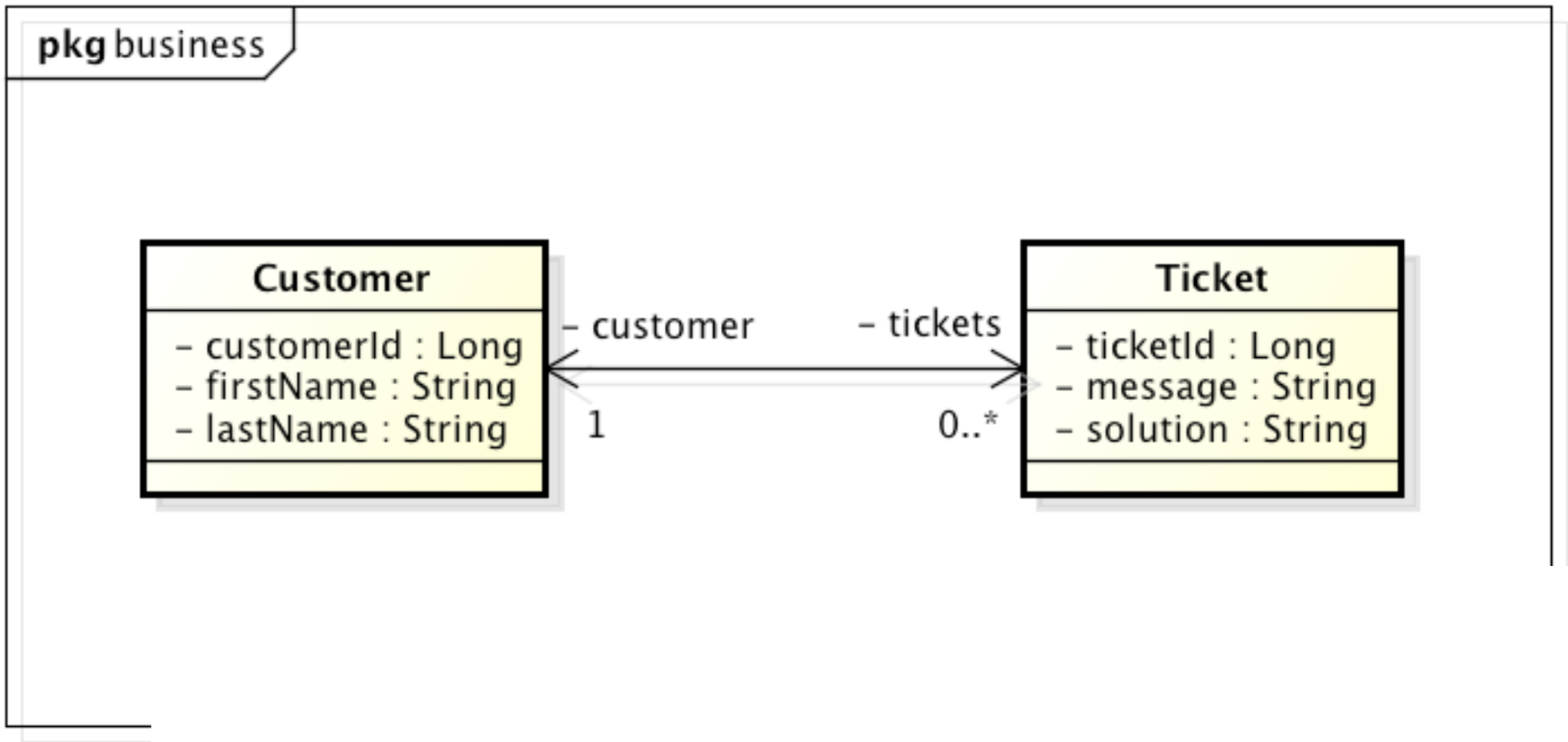
CascadeType

- `CascadeType.MERGE` → opération merge
- `CascadeType.PERSIST` → opération persist
- `CascadeType.REFRESH` → opération refresh
- `CascadeType.REMOVE` → opération merge
- `CascadeType.ALL` → toute opération

One-to-many/Many-to-one

- relations représentées par des collections ou maps : `Collection`, `Set`, `List`, `Map`
- les variantes génériques sont souhaitable(exemple `Collection<Employe>`)
- pas un type concret (e.g. `ArrayList`)
- attribut initialisé avec une classe concrète (générique)

One-to-many/Many-to-one

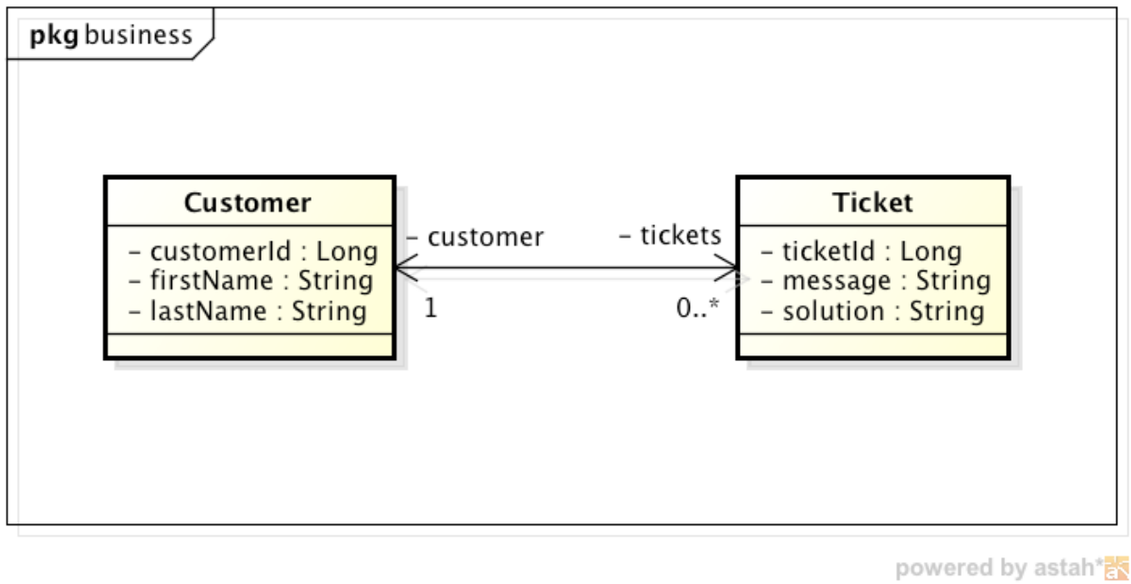


- relation bidirectionnelle : si un bout est `@OneToMany` alors l'autre bout est `@ManyToOne`

```
@Entity
public class Customer {
   @Id private Long customerId;
   ...
   @OneToMany (targetEntity= Ticket.class,
               mappedBy="customer")
   private Set<Ticket> tickets;
...
}
```

nécessaire si la collection n'est pas générique

indique le ***propriétaire***



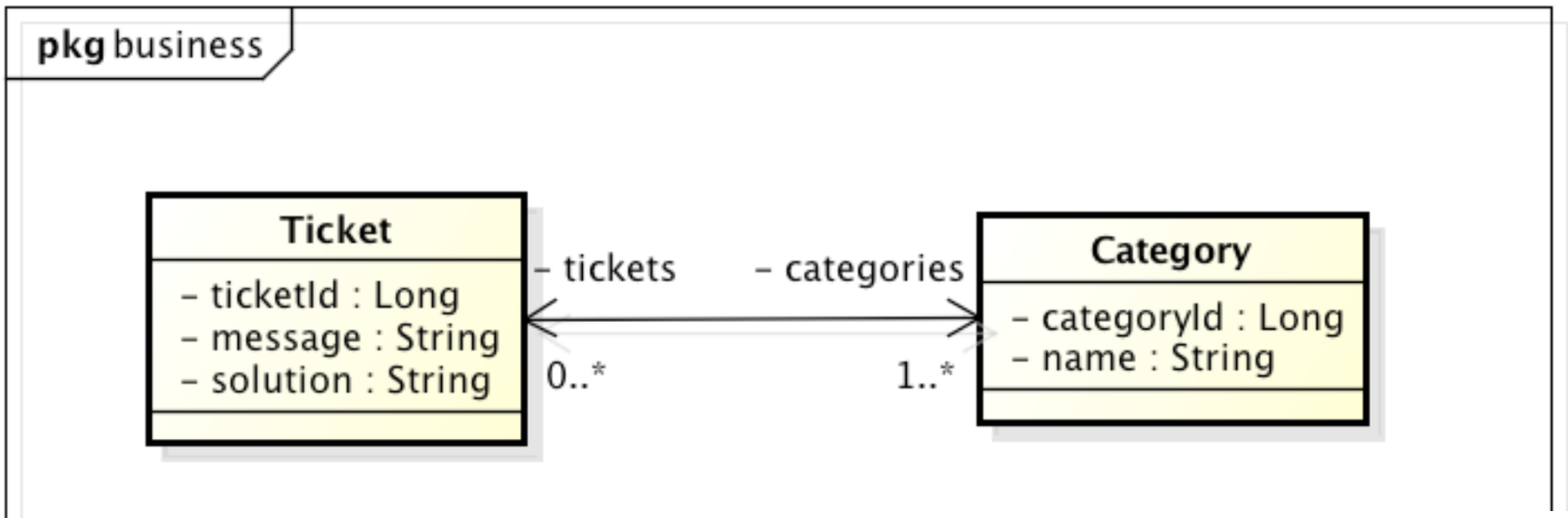

```
@Entity
public class Ticket {
   @Id
   private Long ticketId;
   private String message;

   @ManyToOne
   private Customer customer;
...
}
```

toujours le ***propriétaire*** de la relation (bidirectionnelle)

Many-to-many

- `@ManyToMany`
- représentée par 1 ou 2 collections (uni- ou bi-directionnelle)
- souvent bidirectionnelle

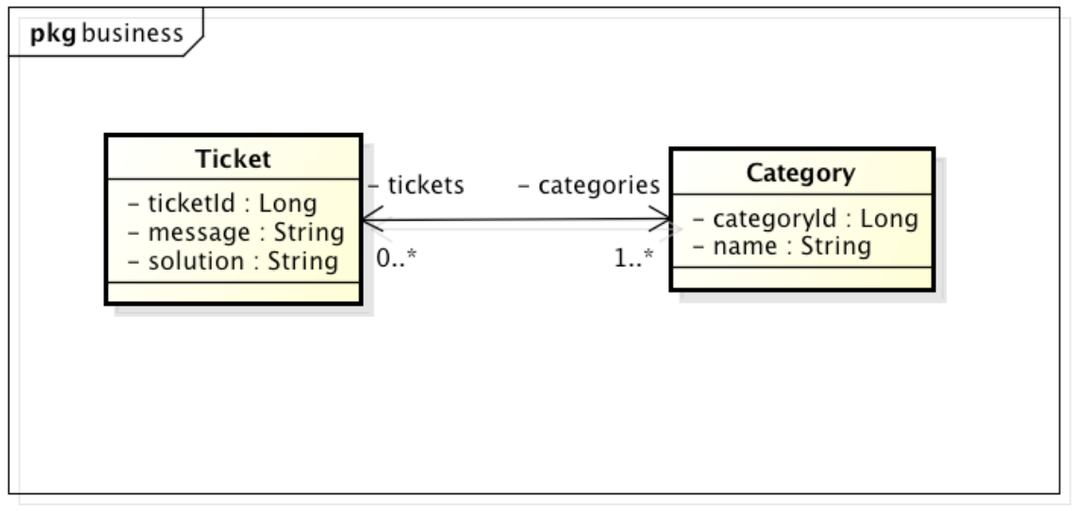
Many-to-many



- on peut choisir le côté propriétaire avec l'attribut `mappedBy`

```
@Entity
public class Category {
   @Id private Long categoryId;
   private String name;
   ...
   @ManyToMany
   private Set<Ticket> tickets;
...
}
```

optional = “true”




```
@Entity
public class Ticket {
   @Id
   private Long ticketId;
   private String message;

   @ManyToMany (mappedBy="tickets")
   private Set<Category> categories;
...
}
```

indique le *propriétaire*

Les entités

- **peuvent être**
  - passées en paramètre des méthodes d'un **`EntityManager`** ou d'un **`Query`**
  - renvoyées par une requête (**`Query`**)
  - le but d'une association
  - référencées dans une requête JPQL

	@OneToOne	@OneToMany	@ManyToOne	@ManyToMany
targetEntity	Yes	Yes	Yes	Yes
cascade	Yes	Yes	Yes	Yes
fetch	Yes	Yes	Yes	Yes
optional	Yes	No	Yes	No
mappedBy	Yes	Yes	No	Yes

Gestion des entités avec le EntityManager

Le gestionnaire d'entités

`javax.persistence.EntityManager`

- le pont entre les mondes OO et relationnel
- fournit les méthodes pour gérer les entités : les rendre persistantes, les supprimer de la base de données, retrouver leurs valeurs dans la base, les mettre à jour, ...
- appel explicite à ses méthodes pour rendre persistante une entité
- implémentation fournie par le fournisseur de persistance

Types gestionnaire d'entités

- géré par le container (uniquement disponible dans un serveur d'applications) → le contexte de persistance n'existe souvent que le temps d'une seule transaction
- **géré par l'application (seul type disponible en dehors d'un serveur d'applications)** → le contexte de persistance reste attaché au GE pendant toute son existence

Interface

```
public void persist(Object entity);
```

- sauvegarde (persiste) une entité dans la BD, rend l'entité *managed* (gérée)

```
public <T> T merge(T entity);
```

- "*merge*" une entité dans le contexte de persistance du EntityManager et retourne l'entité gérée

```
public void remove(Object entity);
```

- enlève une entité de la BD

Interface

```
public <T> T find(Class<T> entityClass,
                  Object primaryKey);
```

- recherche une entité par la clé primaire

```
public void refresh(Object entity);
```

- rafraichit une entité depuis la BD

```
public void flush();
```

- synchronise les entités dans le contexte de persistance du EntityManager avec la BD

...

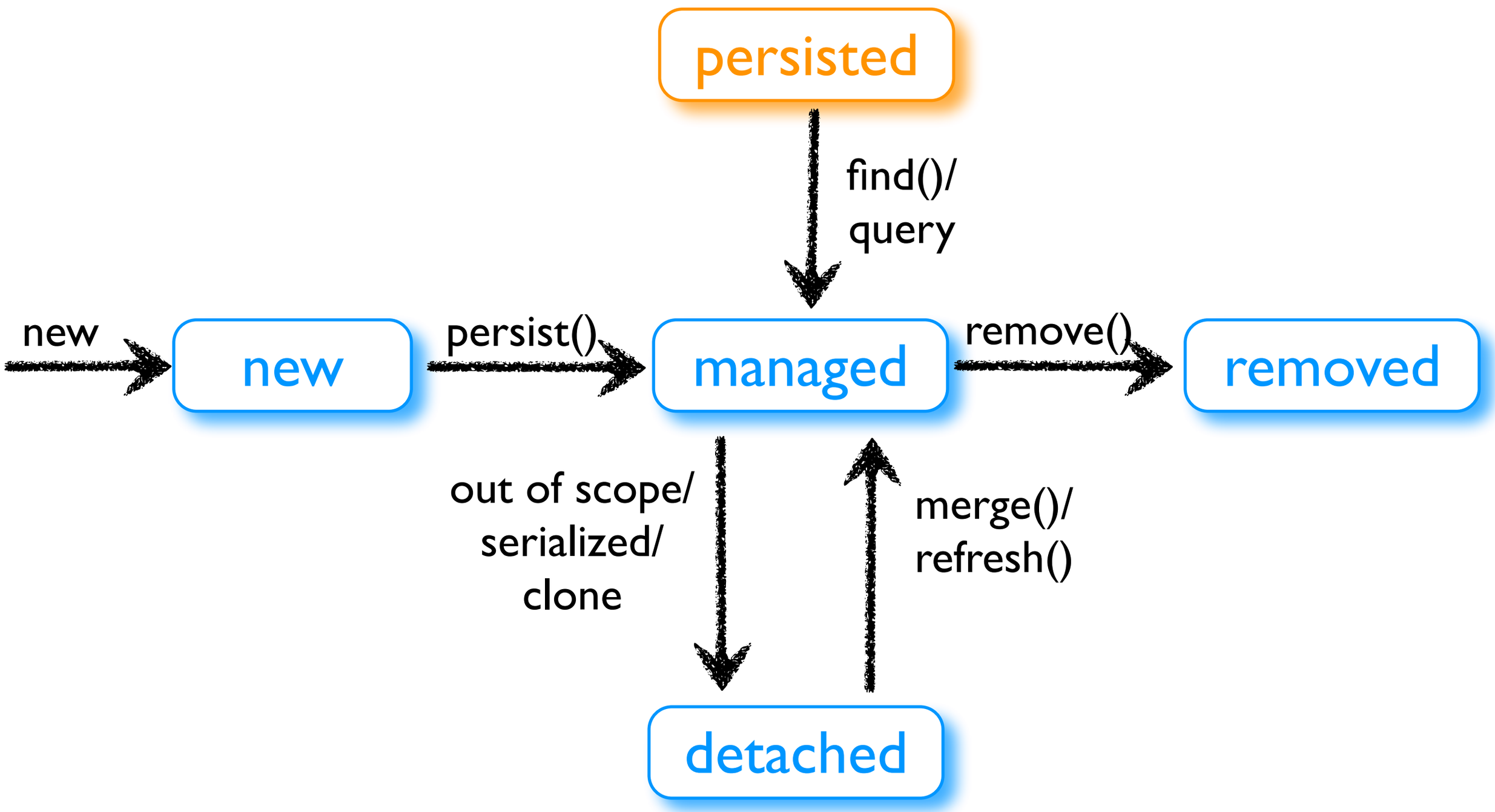
Cycle de vie entité

- new (nouvelle) : créée mais pas associée à un contexte de persistance
- managed (gérée par un gestionnaire de persistance) : **a une identité dans la base de données**
  - **`persist`**(), ou **`merge`**() d'une entité détachée, ou requête faite par un gestionnaire d'entités (dans un contexte transactionnel), ou navigation à partir d'un objet géré

Cycle de vie entité

- detached (détachée) : a une identité dans la base mais n'est plus associée à un contexte de persistance
  - le contexte de persistance est vidé ou entité envoyée dans une autre JVM par RMI
- removed (supprimée) : a une identité dans la base ; associée à un contexte de persistance et ce contexte doit la supprimer de la BD
  - `remove()`

Cycle de vie entité



Contexte de persistance

- collection d'entités gérées par un `EntityManager` pendant un scope de persistance
  - **transaction-scoped** `EntityManager`
  - **extended** `EntityManager`
- toute modification apportée à un objet du contexte de persistance sera enregistrée dans la base de données
- il ne contient pas 2 entités différentes qui représentent des données identiques dans la base

Contexte de persistance

- **transaction-scoped** `EntityManager`
  - entités **attached** pendant une transaction sont automatiquement **détached** à la fin de la transaction
    - avant de détacher il synchronise avec la base
- **extended** `EntityManager`
  - dure pendant plusieurs transactions
  - utilisable seulement avec STSB

Container-managed EM

```
@Stateless(name = "TicketManager")
public class TicketManagerBean implements TicketManager {

    @PersistenceContext(unitName = "glaPU",
                        type = PersistenceContextType.TRANSACTION)
    private EntityManager em;

    public Ticket addTicket(Ticket ticket, long cid) {
        Customer c = em.find(Customer.class, cid);
        ticket.setCustomer(c);
        em.persist(ticket);
        return ticket;
    }
...
```

default

EXTENDED
(possible si
Stateful)

Container-managed EM

```
...
    public Ticket updateTicket(Ticket ticket) {
        em.merge(ticket);
        return ticket;
    }
    public void deleteTicket(Ticket ticket) {
        em.remove(em.merge(ticket));
    }
    public Ticket undoChange(Ticket ticket) {
        Ticket attachedTicket = em.merge(ticket);
        em.refresh(attachedTicket);
        return attachedTicket;
    }
...
}
```

Application-managed EM (dans container JEE)

```
@Stateless(name = "TicketManager")
public class TicketManagerBean implements TicketManager {

   @PersistenceUnit
   private EntityManagerFactory entityManagerFactory;
   EntityManager entityManager;

   @PostConstruct
   public void initialize() {
      entityManager=entityManagerFactory.createEntityManager();
   }
   @PreDestroy
   public void cleanup() {
      if (entityManager.isOpen()) {
         entityManager.close();
      }
   }
```

JTA Application-managed EM

```
@Stateless(name = "TicketManager")
public class TicketManagerBean implements TicketManager {
   @PersistenceUnit
   private EntityManagerFactory entityManagerFactory;
   EntityManager entityManager;
   ...
   public Ticket updateTicket(Ticket ticket) {
       entityManager.joinTransaction();
       entityManager.merge(Ticket);
       return Ticket;
   }
}
...
```

rejoindre une transaction dans le container

- le `joinTransaction` est nécessaire si EM a été créé en dehors d'une transaction JTA active (sinon automatique)
  - souvent avec STSB et Extended Context

Application-managed EM (RESOURCE_LOCAL)

```
@Stateless(name = "TicketManager")
public class TicketManagerBean implements TicketManager {

   @PersistenceUnit
   private EntityManagerFactory entityManagerFactory;
   EntityManager entityManager;

   ...
   public Ticket updateTicket(Ticket ticket) {
        entityManager.getTransaction().begin();
        entityManager.merge(Ticket);
        entityManager.getTransaction().commit();
        return Ticket;
   }
}
...
```

si pas de transaction existante

SE Application-managed EM

```
EntityManagerFactory entityManagerFactory =
    Persistence.createEntityManagerFactory("glaPU");
EntityManager entityManager =
    entityManagerFactory.createEntityManager();
try {
    EntityTransaction entityTransaction =
                entityManager.getTransaction();
    entityTransaction.begin();
    entityManager.persist(ticket);
    entityTransaction.commit();
} finally {
    entityManager.close();
    entityManagerFactory.close();
}
```

persistance

Opérations

- persister
- rechercher
- mettre à jour

Persister les entités

```
INSERT INTO TICKETS (MESSAGE, SOLUTION, CUSTOMER_ID, ..
VALUES ("Big problem","Smart solution", 3, ...)
```

```
public Ticket addTicket(String mes, String sol, long cid){
        Ticket ticket = new Ticket();
        ticket.setMessage(mes);
        ticket.setSolution(sol);
        Customer c = em.find(Customer.class, cid);
        ticket.setCustomer(c);
        em.persist(ticket);
        return ticket;
}
```

exécuté quand la transaction correspondante est en train de commiter

- fin méthode
- EM closed
- EM flushed

si la clé existe →

```
javax.persistence.PersistenceException
```

Persister les relations

```
public Customer addCustomer(String fname, String lname,
                            String creditCardType){
  BillingInfo bi = new BillingInfo();
  bi.setCreditCardType(creditCardType);
  em.persist(bi);
  Customer customer = new Customer();
  customer.setFirstName(fname);
  customer.setLastName(lname);
  customer.setBillingInfo(bi);

  em.persist(customer);
  return customer;
}
```

persister explicitement

par défaut, ne persiste pas le **BillingInfo** du client

Spécifier le type de cascading

```
@Entity public class Customer {
  @Id private Long customerId;
  ...
  @OneToOne (cascade = CascadeType.PERSIST)
  private BillingInfo billingInfo;
...
}

@Entity public class BillingInfo {
  @Id
  private Long billingId;
  private String creditCardType;
...
}
```

Rappel `CascadeType`

- `CascadeType.MERGE` → opération merge
- `CascadeType.PERSIST` → opération persist
- `CascadeType.REFRESH` → opération refresh
- `CascadeType.REMOVE` → opération merge
- `CascadeType.ALL` → toute opération

Rechercher les entités

```
public Customer findCustomer(long cid){
        Customer customer = em.find(Customer.class, cid);
        return customer;
}
```

- si inexistant pas d'exc
- utilise EM caching
- modes `fetch`
  - `EAGER` → chargement immédiat
  - `LAZY` → chargement au premier accès

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
    @Lob
    @Basic(fetch=FetchType.LAZY)
    private byte[] picture;
      ...
```

Charger les entités reliées

- utiliser `fetch` sur le type de relation

	Défaut	No entités chargées
@OneToOne	`EAGER`	1
@OneToMany	`LAZY`	*
@ManyToOne	`EAGER`	1
@ManyToMany	`LAZY`	*

Mettre à jour les entités

```
public Customer addCustomer(String fname, String lname,
                            String creditCardType){
   ....
   em.persist(customer);
   customer.setLastName(lname + " the best");
   return customer;
}
```

managed

synchronisé automatiquement avec le contexte persistance (et la BDD)

- **r(é)attacher une entité à un EM**
  - seulement pour une entité qui existe dans la BDD (sinon `IllegalArgumentException`)

```
public Ticket updateTicket(Ticket ticket) {
        em.merge(ticket);
        return ticket;
}
```

rappel:
la requête correspondante n'est pas exécutée immédiatement

Mettre à jour les relations

- par défaut, les entités associées ne sont pas mises à jour
- il faut utiliser le `CascadeType.MERGE` (ou `CascadeType.ALL`)
- comme **`persist`** (et la plupart des autres opérations), **`merge`** doit être appelé dans un contexte transactionnel
  - sinon, `TransactionRequiredException`

Effacer des entités

```
public void deleteTicket(Ticket ticket) {
    em.remove(em.merge(ticket));
}
```

- on peut effacer (avec **`remove`**) seulement des entités attachées
  - sinon, `IllegalArgumentException`
- la requête correspondante n'est pas exécutée immédiatement mais l'entité est marquée "removed" (changements ultérieurs pas synchronisés)

Effacer les relations

- par défaut, les entités associées ne sont pas effacées
- **il faut utiliser le `CascadeType.REMOVE` (ou `CascadeType.ALL`)**
  - justifié pour **`@OneToOne, @OneToMany`**
  - attention aux relations des entités liées

Contrôle des update

- opérations `persist`, `merge`, `remove` → modifications dans la BDD quand EM "flushed"
- `FlushModeType.AUTO`
  - commit, persistence-context fermé, requête liée
- **`em.setFlushMode(FlushModeType.COMMIT)`**
  - **attention :** flush avant certaines requêtes peut être nécessaire
- **`em.flush()`**

Rafraîchir les entités

version attachée de `ticket`

```
public Ticket undoChange(Ticket ticket) {
    Ticket attachedTicket = em.merge(ticket);
    em.refresh(attachedTicket);
    return attachedTicket;
}
```

rappel:
la requête correspondante n'est pas exécutée immédiatement

- **`refresh`** utilisé pour
  - annuler les modifications faites sur une entité
  - récupérer les valeurs par défaut (dans la BDD)

Listeners

- déclarer des callbacks pour les événements relatifs au cycle de vie des entités
- généralement utilisés pour la journalisation, la validation (de données), l'envoie de notifications
- il est préférable de définir les callbacks dans un listener différent de l'entité concernée

Callbacks

- **PrePersist** → avant que le EM persiste l'entité
- **PostPersist** → après avoir persisté l'entité
- **PostLoad** → après chargement entité avec requête, find ou refresh
- **PreUpdate** → avant update d'une BDD pour synchroniser une entité
- **PostUpdate** → après update d'une BDD
- **PreRemove** → avant que le EM efface l'entité
- **PostRemove** → après avoir effacé l'entité

Utilisation listener

```
public class TicketMonitor { ...

   @PrePersist
   @PreUpdate
   public void monitorTicket(Ticket ticket) {
      // logging
      // envoyer notifications
   }
 }


@Entity
@EntityListeners(persistence.TicketMonitor.class)
public class Ticket implements Serializable {
   ...
}
```

Utilisation listener

pattern : `void METHOD(Object)`

entité correspondant au callback

```
public class TicketMonitor { ...

   @PrePersist
   @PreUpdate
   public void checkTicket(Ticket ticket) {
      if(ticket.getMessage().getSize() > MAX_SIZE){
          throw new TicketException(“Ticket too big”);
      }
   }
}
```

si une exception est levée alors l’opération est abandonnée

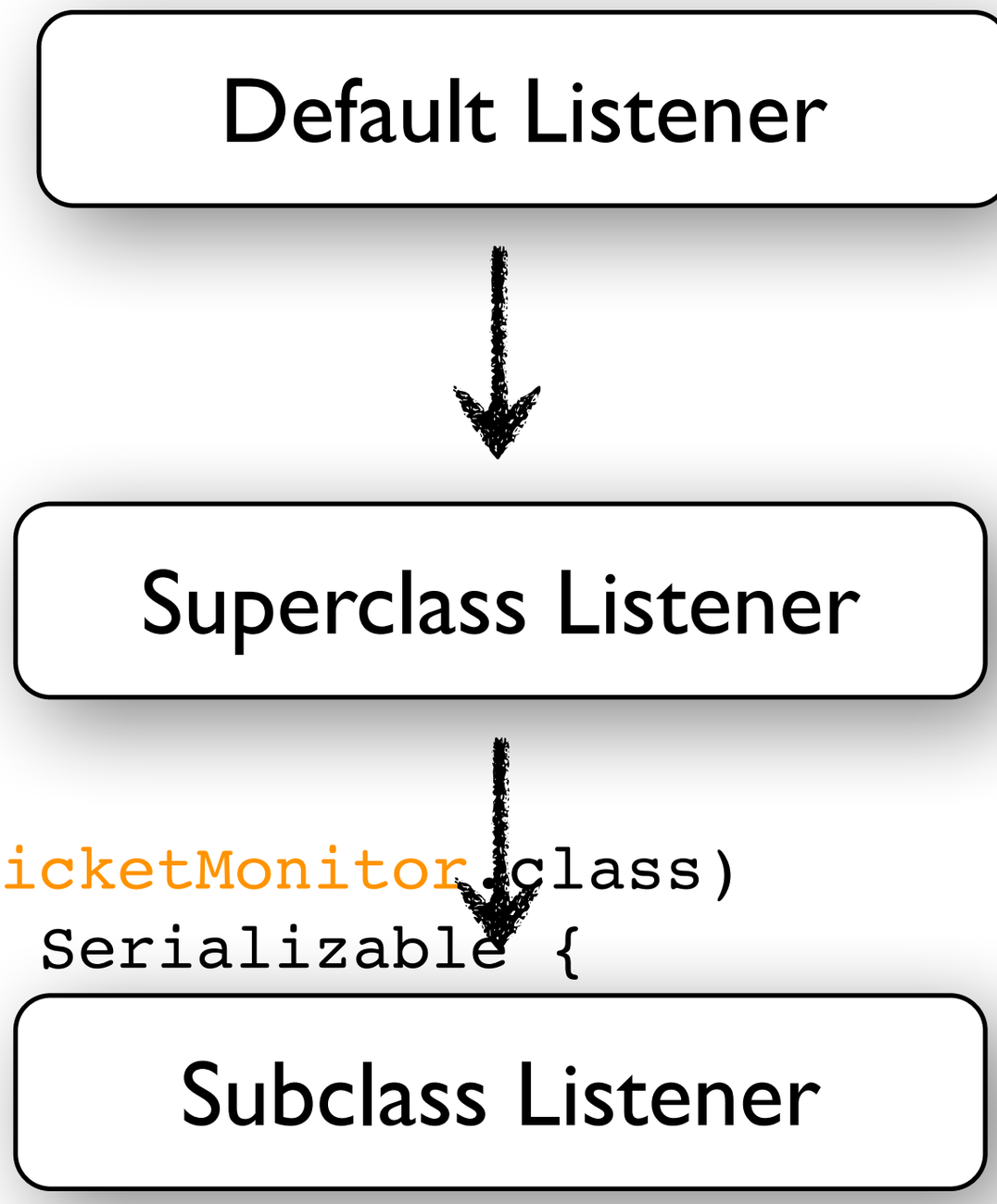
Ordre d'application

dans `persistence.xml`

- **plusieurs intercepteurs à un niveau** → appliquer dans l'ordre de spécification

- **exclure des intercepteurs**

```
@Entity
@ExcludeDefaultListeners
@ExcludeSuperClassListeners
@EntityListeners(persistence.TicketMonitor.class)
public class Ticket implements Serializable {
   ...
}
```



Best practices

- **utiliser Container-managed Entity Manager**
  - focus on business logic
- **éviter l'injection de EM dans tier web**
  - JNDI lookup ou des facades
- **séparer les listeners des entités**
  - ne pas polluer le modèle domaine

Questions ?