Persistance
- ORM -
- JPQL -

Retrouver les entités avec JPQL

Java Persistence Query Language

Interroger la base de données

- Il est possible de rechercher des données sur des critères plus complexes que la simple identité
- Etapes
  - décrire ce qui est recherché
  - créer une instance de type Query
  - initialiser la requête (paramètres, pagination)
  - lancer l'exécution de la requête

JDBC vs. JPA

JDBC	JPA/JPQL
Obtenir une connexion à la BD	Obtenir une instance d'un EM
Définir une requête	Créer une requête
Exécuter la requête	Exécuter la requête
Récupérer les résultats (enregistrements)	Récupérer les résultats (entités)

Requêtes

- nommées
  - stockées et réutilisées
- dynamiques
  - créé à la volée
- même format pour les deux types
- typées ou non

Requête dynamique

```
@PersistenceContext
private EntityManager em;

public List<Customer> findAllCustomers() {
  Query query =
        em.createQuery("SELECT c FROM Customer c");
  return (List<Customer>) query.getResultList();
}
```

résultat de type `List`

Requête dynamique typée

```
@PersistenceContext
private EntityManager em;

public List<Customer> findAllCustomers() {
  TypedQuery<Customer> query =
       em.createQuery("SELECT c FROM Customer c",
                      Customer.class);
  return query.getResultList();
}
```

résultat de type `List<Customer>`

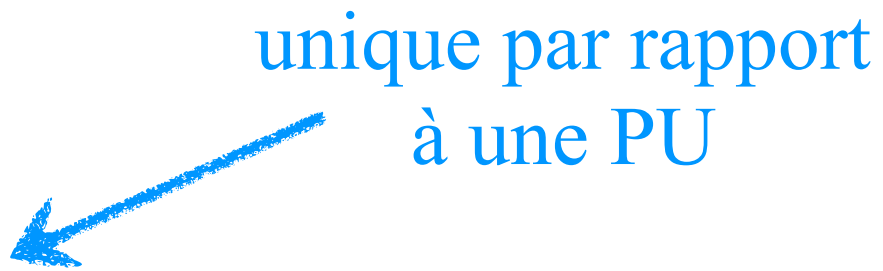
- pour simplifier, on utilisera souvent des requêtes non-typées mais les versions typées sont souvent préférables

Requête nommée

- réutilisables
- pas disséminées dans la logique métier
- peuvent améliorer l'efficacité (préparées une fois et réutilisées efficacement)

Requête nommée

```
@Entity
@NamedQuery(name = "Client.findAll",
            query = "SELECT c FROM Customer c")
public class Customer implements Serializable {
  ...
}
```



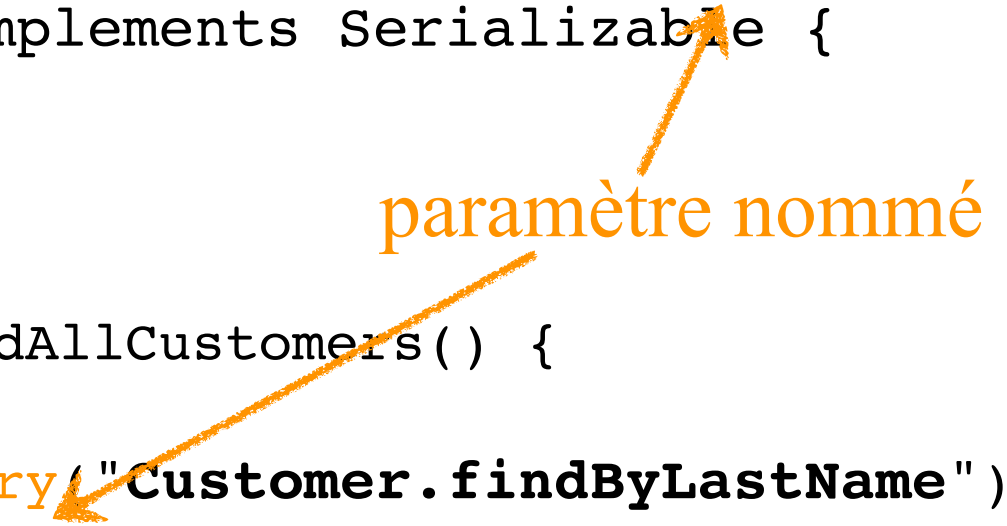

```
@Entity
@NamedQueries({
    @NamedQuery(name = "Customer.findAll",
                query = "SELECT c FROM Customer c"),
    @NamedQuery(name = "Customer.findByLastName",
                query = "SELECT c FROM Customer c
                         WHERE c.lastName = :name")
})
public class Customer implements Serializable {
  ...
}


public List<Customer> findAllCustomers() {
  Query query =
        em.createNamedQuery("Customer.findAll");
  return query.getResultList();
}
```

Création requêtes

- **public Query createQuery(String qlString);**
  - requête JPQL dynamique
- **public Query createNamedQuery(String name);**
  - requête JPQL nommée
- **public Query createNativeQuery(String sqlString);**
  - requête native dynamique avec UPDATE/DELETE
- **public Query createNativeQuery(String sql,Class rc);**
  - requête native dynamique
- ...

Paramètres

```
@Entity
@NamedQuery(name = "Customer.findByLastName",
            query = "SELECT c FROM Customer c
                     WHERE c.lastName = :name")
public class Customer implements Serializable {
  ...
}

public List<Customer> findAllCustomers() {
  Query query =
        em.createNamedQuery("Customer.findByLastName");
  query.setParameter("name", "Bond");
  return query.getResultList();
}
```



Paramètres

```
@Entity
@NamedQuery(name = "Customer.findByLastName",
            query = "SELECT c FROM Customer c
                     WHERE c.lastName = ?1")
public class Customer implements Serializable {
  ...
}
```

paramètre positionnel

```
public List<Customer> findAllCustomers() {
  Query query =
        em.createNamedQuery("Customer.findByLastName");
  query.setParameter(1, “Bond”);
  return query.getResultList();
}
```

Récupérer le(s) résultat(s)

```
public List<Customer> findAllCustomers() {
    Query query =
          em.createNamedQuery("Customer.findByLastName");
    query.setParameter(“name”, “James”);
    return query.getResultList();
}
```

liste vide si pas de résultat correspondant

collection d’entités

- efficacité → exécuter les requêtes de type read-only en dehors d’une transaction
  - résultats attachés si exécutée dans une transactions

Récupérer le(s) résultat(s)

```
public Customer findCustomers(String name) {
  Query query =
          em.createNamedQuery("Customer.findByLastName");

  try{
    query.setParameter(“name”, name);
    Customer customer = (Customer) query.getSingleResult();
  }catch (NonUniqueResultException ex) {
     handleException(ex);
  }catch (NoResultException ex) {
     handleException(ex);
  }
  ...
}
```

entité unique

résultat de type `Object`

Récupérer le(s) résultat(s)

```
public Customer findCustomers(String name) {
  TypedQuery<Customer> query =
          em.createNamedQuery("Customer.findByLastName",
                              Customer.class);
  try{
    query.setParameter("name", name);
    Customer customer = query.getSingleResult();
  }catch (NonUniqueResultException ex) {
     handleException(ex);
  }catch (NoResultException ex) {
     handleException(ex);
  }
  ...
}
```

entité unique

résultat de type `Customer`

Pagination

```
List<Customer> findAllCustomersPaginated(int pageIndex,
                                         int pageSize) {
    Query query =
        em.createNamedQuery("Customer.findAll");

    query.setMaxResults(pageSize) ;
    query.setFirstResult(pageIndex) ;

    return query.getResultList();
}
```

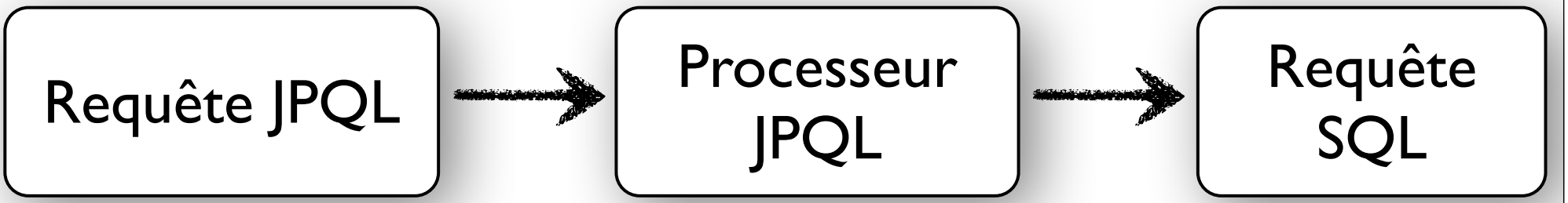
nombre maximum d'entités à retrouver

position du premier résultat retourné

FlushModeType

- au niveau EM ou `Query` (prioritaire)
- en général, `FlushModeType.AUTO convient`
  - permet de garantir l'intégrité des requêtes
- toutes les requêtes dans une transaction n'ont pas forcement besoin d'un flush
  - `query.setFlushMode(FlushModeType.COMMIT)`

Introduction JPQL



- 3 types d'instructions
  - `SELECT`
  - `UPDATE`
  - `DELETE`

http://docs.oracle.com/javaee/6/tutorial/doc/

Construction SELECT

- SELECT → les entités/valeurs/objets à retrouver
- FROM → déclarations entités utilisées dans SELECT
- WHERE → filtrer les résultats
- ORDER BY → ordoner les résultats
- GROUP BY → réaliser une agrégation
- HAVING → filtrer en conjonction avec agrégation

FROM et WHERE

optionnel

```
SELECT c
FROM Customer AS c
```

single-value
path expression

```
SELECT c
FROM Customer c
WHERE c.lastName = "Bond"
```

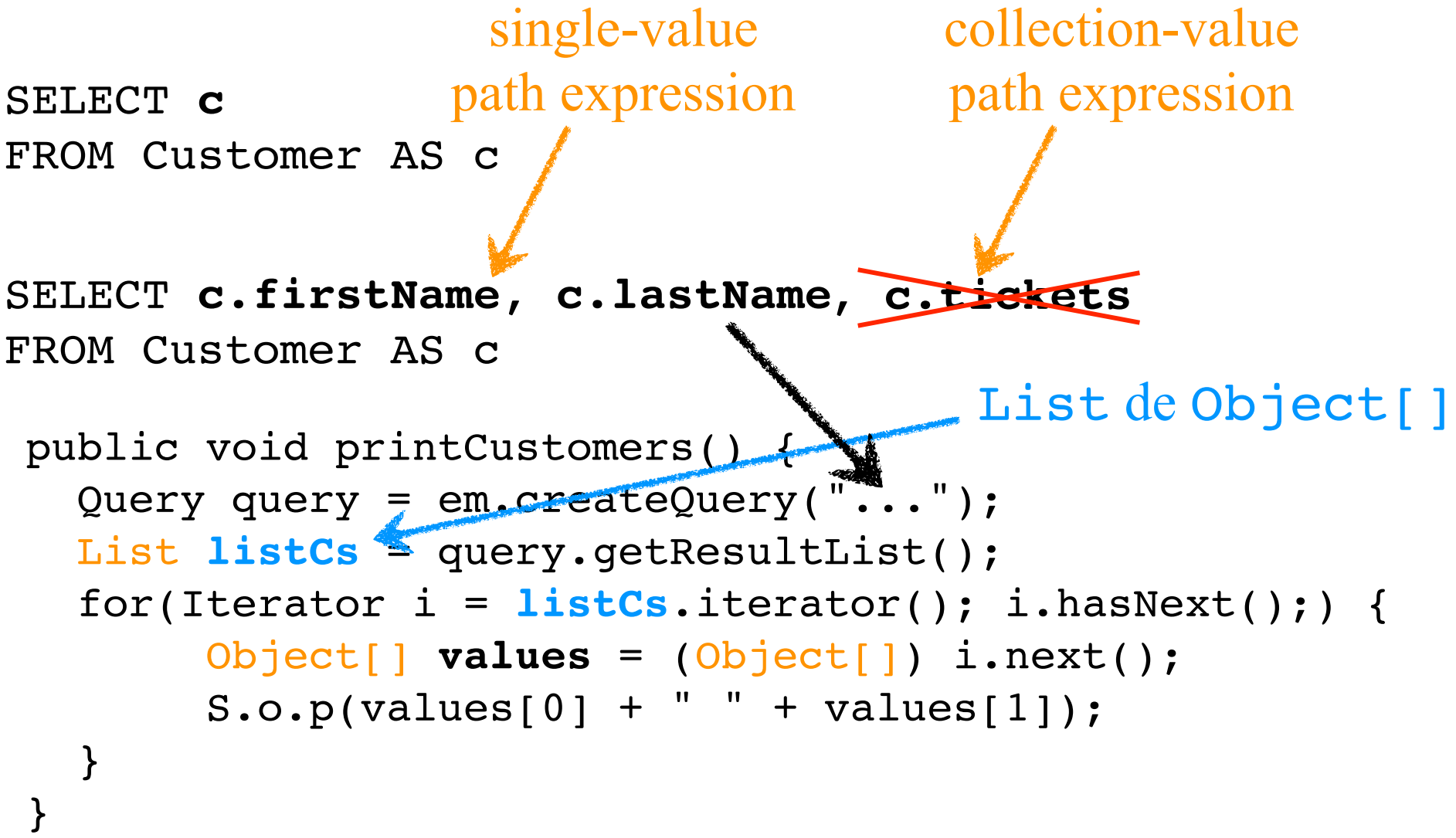
collection-value
path expression

```
SELECT c
FROM Customer c
WHERE c.tickets is NOT EMPTY
```

Opérateurs

Type	Opérateur
Navigational	.
Unary sign	+,-
Arithmetic	*,/ +,-
Relational	>, >=, < ,<=, <>, [NOT] BETWEEN, [NOT] LIKE,[NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
Logical	NOT AND OR

SELECT




```
SELECT c
FROM Customer AS c


SELECT c.firstName, c.lastName, c.tickets
FROM Customer AS c

 public void printCustomers() {
   Query query = em.createQuery("...");
   List listCs = query.getResultList();
   for(Iterator i = listCs.iterator(); i.hasNext();) {
        Object[] values = (Object[]) i.next();
        S.o.p(values[0] + " " + values[1]);
   }
 }
```

fully-qualified name

SELECT

```
SELECT new entity.BasicClient(c.firstName,c.lastName)
FROM Customer AS c
```

```
public class BasicClient {
    private String firstName;
    private String lastName;
...
```

```
public void printCustomers() {
  Query query = em.createQuery("...");
  List<BasicClient> listCs = query.getResultList();
  for(BasicClient cl: listCs) {
       S.o.p(cl.firstName + " " + cl.lastName);
  }
}
```

JOINs

- Theta-joins
- INNER
- OUTER
- FETCH

- **Theta** → en général, en absence d'une relation entre les entités

```
SELECT p.number
FROM Customer c, Phone p
WHERE p.customer = c AND p.type = "Home"
```

- **INNER** → en termes d'association avec l'entité correspondante


```
SELECT p.number
FROM Customer c JOIN c.phones p
WHERE p.type = "Home"
```

- path-navigation correspond à des INNER JOIN

```
SELECT c.department
FROM Customer c

        SELECT d
        FROM Customer c JOIN c.department d
```

- peuvent conduire à des JOIN multiples

```
SELECT c.department
FROM Customer c
WHERE c.address.code = 54000
```

```
SELECT DISTINCT d
FROM Customer c JOIN c.department d JOIN c.address a
WHERE a.code = 54000
```

OUTER JOIN

- seulement une partie de la relation doit être complète

tous les clients avec un département assigné

```
SELECT c, d
FROM Customer c INNER JOIN c.department d
```

tous les clients avec **éventuellement** leur département

```
SELECT c, d
FROM Customer c LEFT JOIN c.department d
```

LEFT/RIGHT [OUTER] JOIN

FETCH JOIN

- permet la spécification de relations qui doivent être naviguées et préchargées par le moteur de requêtes (pour ne pas être chargées de manière lazy au runtime)

```
SELECT c
FROM Customer c JOIN FETCH c.address
```

l'adresse est déjà présente dans les entités `Customer` obtenues comme résultat

Aggregate queries

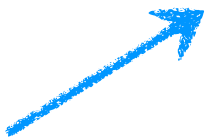
- fonctions d'agrégation généralement pour les requêtes qui permettent la générations de rapports:
  - `AVG, COUNT, MAX, MIN, SUM`

```
SELECT d.name, AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d.name
HAVING COUNT(e) > 2
```

Updates

```
@Entity
@NamedQuery(name = "Customer.updateStatus",
            query = "UPDATE Customer c
                     SET c.status = 'Hard worker'
                     WHERE c.numTickets >= ?1")
public class Customer implements Serializable { ...
```

```
public int updateCustomers() {
 Query query = em.createNamedQuery("Customer.updateStatus");
 query.setParameter(1, 1);
 return query.executeUpdate();
}
```



return: nombre de items modifiés

invoqué dans transaction active

Updates

```
@Entity
@NamedQuery(name = "Customer.deleteSome",
            query = "DELETE FROM Customer c
                     WHERE c.status = :status")
public class Customer implements Serializable { ...
```

```
public int deleteLazyCustomers() {
 Query query =  em.createNamedQuery("Customer.deleteSome");
 query.setParameter("status", "Lazy");
 return query.executeUpdate();
}
```

Updates

- garantit l'exécution des opération (update/remove) dans la BDD
- Persistence Context n’est pas mis à jour pour refléter les résultats des opérations
  - isoler les opérations dans une transaction séparée: `@TransactionAttributeTransactionAttributeType.REQUIRES_NEW`
  - opérations update exécutées en premier dans une transaction

Pas vu

- subqueries
- hints
- requêtes natives
- ...

Best practices

- utiliser les requêtes nommées (si possible)
  - précompilation au déploiement : efficacité, sécurité
- exécuter les requêtes rapport (qui ne modifient pas les résultats) en dehors de la transaction (dans un EM transaction-scoped)
  - éviter le cout pour attacher les entités
- sélectionner seulement les propriétés nécessaires si une entité est couteuse à la construction (à cause des relations “eager”)

Best practices

- **utiliser les stateless session beans pour l'exécution de requêtes**
  - utilisation de noms (liés à la logique métier) de méthodes à la place de nom de requêtes
  - optimiser les transactions en fonction de la nécessité ou non d'avoir des entités attachées
  - un transaction-scoped EM garantit que les entités ne restent pas attachées plus que nécessaire

Object-relational mapping (ORM)

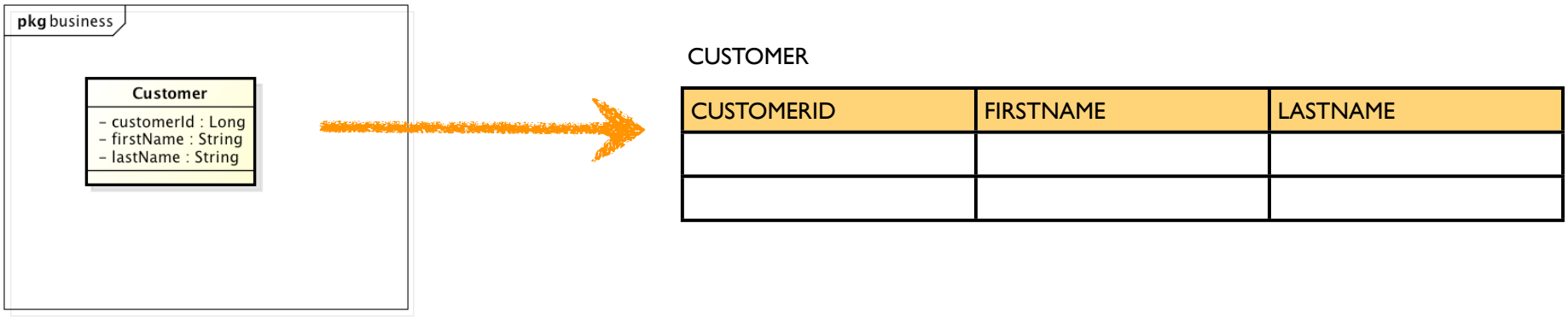
- **processus de sauvegarde d'un objet dans une BD relationnelle**
- **persistance automatique** → meta-données pour indiquer à un API haut-niveau les tables où les objets sont stockés
- **impedance mismatch =** différence entre les paradigmes OO et relationnel
- **avantages :** éviter le boilerplate, portabilité

Impedance mismatch

Modèle OO	Modèle relationnel
Classes, objets	Tables, lignes
Attributs, propriétés	Colonnes
Identité	Clé primaire
Relation/référence entités	Clé etrangere
Héritage, polymorphsme	Pas supporté
Méthodes	(indirect) Procédures stockées, ...

Mapping des objets dans la BD

- **cas simples → une table correspond à une classe**
  - nom table → nom classe
  - noms colonnes → noms attributs persistants



Mapping entités

- valeurs par défaut pour le mapping des classes entité avec les tables de la base de données
- ajouter des informations de configuration que si ces valeurs par défaut ne conviennent pas
  - la base de données est créée par l'application JPA → valeurs par défaut conviennent souvent
  - la base de données existe déjà → les valeurs par défaut ne conviennent souvent pas

Spécifier les tables

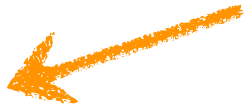

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
...
```

- la plupart des providers permettent une génération automatique des schémas
- possibilité de spécifier des contraintes pour l'auto-création
- la plupart des entités sont mappées dans une seule table

Spécifier les colonnes

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
    @Column(name = "CUSTOMER_ID")
    private Long customerId;
    ...
```

par défaut `CUSTOMERID`



- possibilité de spécifier des contraintes pour l'auto-création : `unique`, `nullable`, `insertable`, `updatable`,...

Attributs énumération

- pour les attributs de type énumération

```
public enum CustomerType {STUDENT,PROF,ADMIN};
```

  - ordinal

    “STUDENT”,“PROF”,“ADMIN”

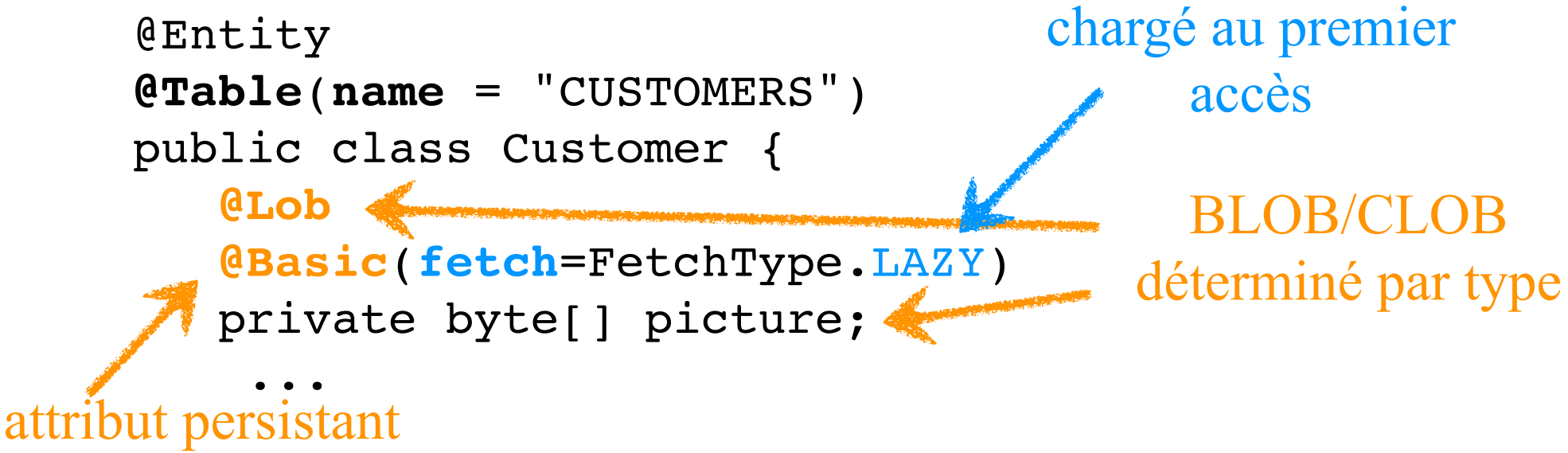
```
@Enumerated(EnumType.ORDINAL)
private CustomerType customerType;
```

  - String

```
@Enumerated(EnumType.STRING)
private CustomerType customerType;
```

Attributs CLOB/BLOB

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



- indiquer qu'un attribut est un Character ou Binary LOB (Large OBject)
- *lazy loading* est optionnel pour le provider

Attributs temporels

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
    @Temporal(TemporalType.DATE)
    private Date birthday;
...
```

- indiquer quel type temporel utiliser :

  `DATE, TIME, TIMESTAMP`

Tables multiples

clé secondaire

```
@Entity
@Table(name = "CUSTOMERS")
@SecondaryTable(name="CUSTOMER_PICTURES",
 pkJoinColumns=@PrimaryKeyJoinColumn(name="CUSTOMER_ID"))
public class Customer {
    @Id @Column(name = "CUSTOMER_ID")
    private Long customerId;
    ...
    @Lob @Basic(fetch=FetchType.LAZY)
    @Column(name="PICTURE", table="CUSTOMER_PICTURES")
    private byte[] picture;
...
```

- utile pour les cas où la base de données existe déjà et ne correspond pas au modèle objet

Génération automatique de clé

- `@GeneratedValue` → générer automatiquement les clés de type numérique entier
- attribut `strategy` → comment la clé sera générée
  - identités, séquences, tables
- si la clé est générée automatiquement, le code Java ne doit pas y faire référence

Types de génération

- **`AUTO`** : type de génération choisi par le fournisseur de persistance, selon le SGBD
- **`SEQUENCE`** : séquence dans la BD
- **`IDENTITY`** : colonne de type IDENTITY
- **`TABLE`** : table qui contient la prochaine valeur de l'identificateur

IDENTITY

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "CUSTOMER_ID")
    private Long customerId;
    ...
```

- suppose une contrainte identité sur `CUSTOMERS.CUSTOMER_ID`

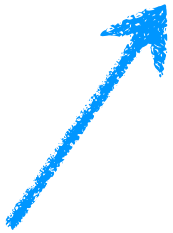
Séquence

```
CREATE SEQUENCE CUSTOMER_SEQUENCE
START WITH 1 INCREMENT BY 10;
```

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
  @SequenceGenerator(name="CUSTOMER_SEQUENCE_GENERATOR",
      sequenceName="CUSTOMER_SEQUENCE",
      initialValue=1, allocationSize=10)
  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE,
                  generator="CUSTOMER_SEQUENCE_GENERATOR")
  @Column(name = "CUSTOMER_ID")
  private Long customerId;
    ...
```

- les générateurs d'identificateurs peuvent être utilisés dans toute l'unité de persistance

```
CREATE TABLE SEQUENCE_GENERATOR_TABLE
(SEQUENCE_NAME VARCHAR2(80) NOT NULL,
 SEQUENCE_VALUE NUMBER(15) NOT NULL,
 PRIMARY KEY (SEQUENCE_NAME));
```



```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
  @TableGenerator (name="CUSTOMER_TABLE_GENERATOR",
                   table="SEQUENCE_GENERATOR_TABLE",
                   pkColumnName="SEQUENCE_NAME",
                   valueColumnName="SEQUENCE_VALUE",
                   pkColumnValue="CUSTOMER_SEQUENCE")
  @Id
  @GeneratedValue(strategy=GenerationType.TABLE,
                  generator="CUSTOMER_TABLE_GENERATOR")
  @Column(name = "CUSTOMER_ID")
  private Long customerId;
    ...
```

```
INSERT INTO SEQUENCE_GENERATOR_TABLE
(SEQUENCE_NAME, SEQUENCE_VALUE) VALUES ('CUSTOMER_SEQUENCE', 1);
```

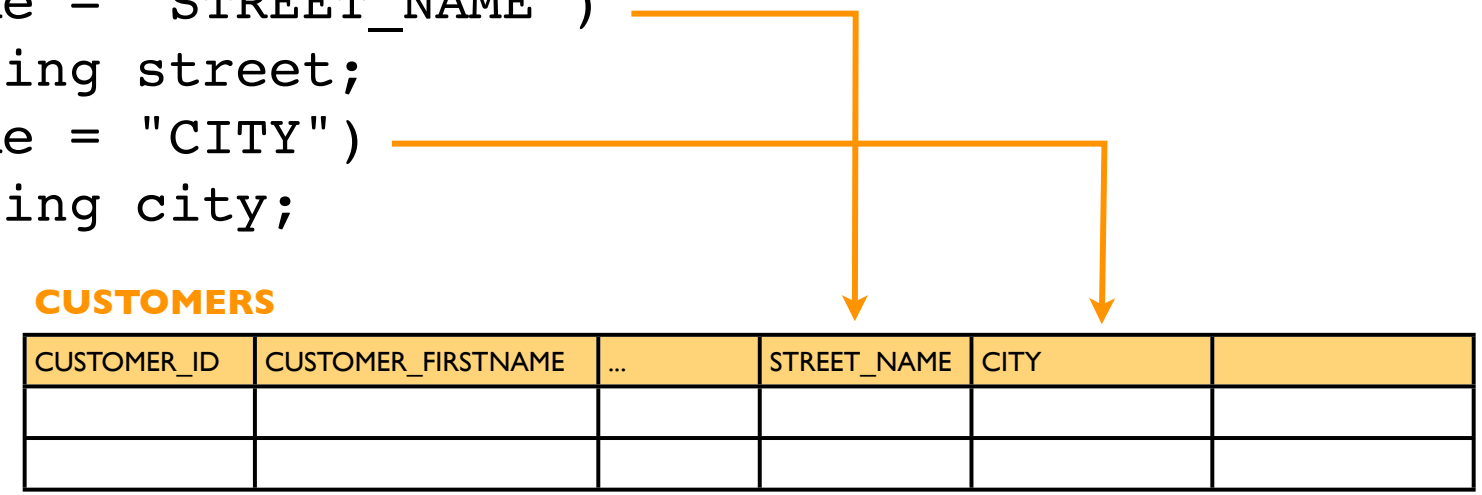
Classes *Embeddable*

- **rappel**
  - objets persistants mais qui ne peuvent pas avoir une identité (dans la BD)
  - leurs attributs sont sauvegardés dans la table associée à l'entité englobante (`@Embedded`)
- **mêmes** annotations/contraintes/types pour les attributs **que les entités**
- **même type d'accès que l'entité englobante**

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable {
    @Id @Column(name = "CUSTOMER_ID")
    private Long customerId;
    @Embedded
    private Address address;
...
}
```

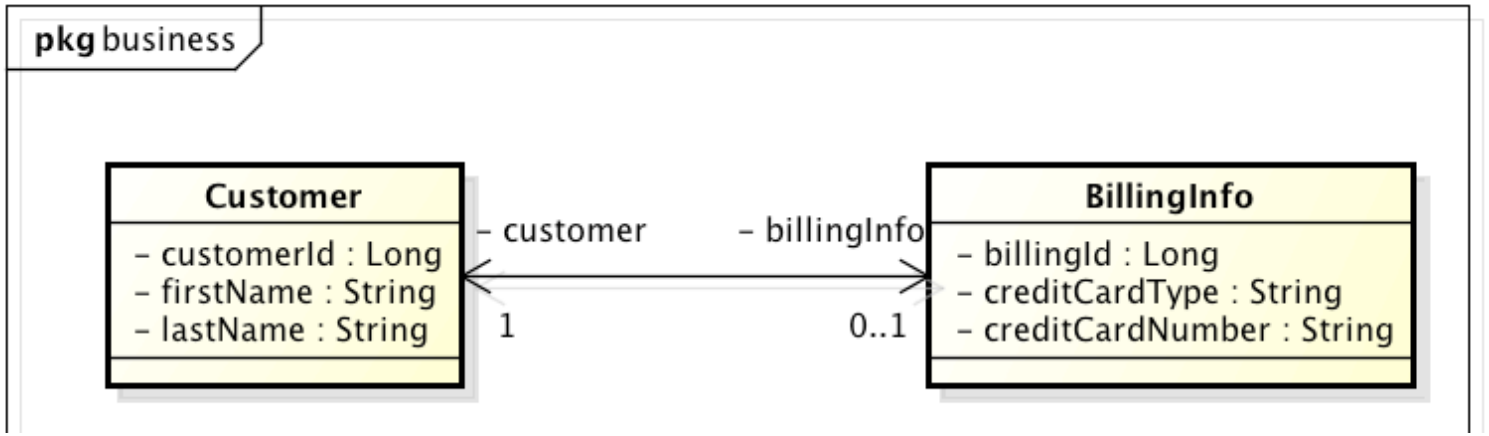
```
@AttributeOverrides(
{@AttributeOverride(
name="street",
column=@Column(name="STREET")
)})
```

```
@Embeddable
public class Address implements Serializable{
  @Column(name = "STREET_NAME")
  private String street;
  @Column(name = "CITY")
  private String city;
...
}
```



Mapping relations

Mapping one-to-one



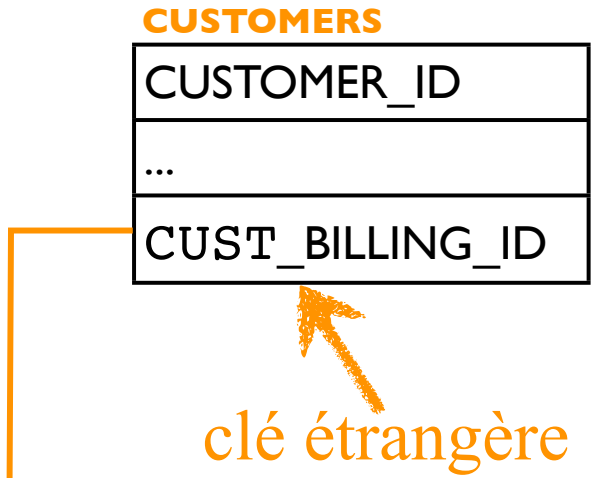
- représentée typiquement par une clé étrangère (ajoutée dans la table côté propriétaire)
- 2 possibilités en fonction de l'emplacement de la clé étrangère

```
@JoinColumn

@PrimaryKeyJoinColumn
```

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
   @Id @Column(name = "CUSTOMER_ID")
   private Long customerId;
   @OneToOne
   @JoinColumn(name="CUST_BILLING_ID",
      referencedColumnName="BILLING_ID")
   private BillingInfo billingInfo;
...
}
```

default : BILLINGINFO_BILLING_ID

default : BILLING_ID

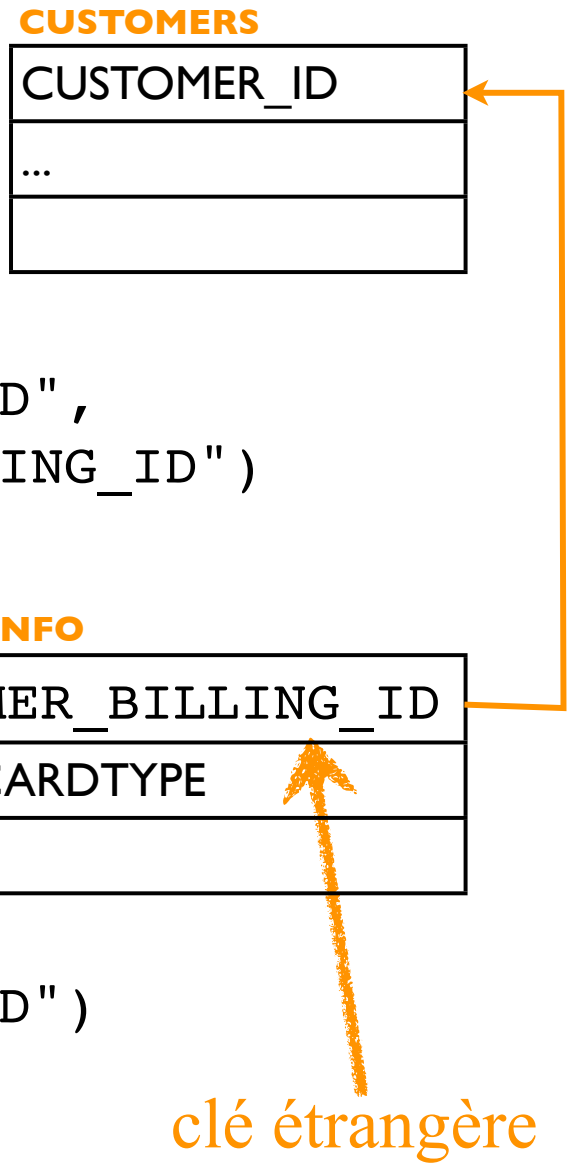



```
@Entity
@Table(name = "BILLING_INFO")
public class BillingInfo {
   @Id @Column(name = "BILLING_ID")
   private Long billingId;
   @OneToOne(mappedBy="billingInfo",optional="false");
   private User user;
...
}
```

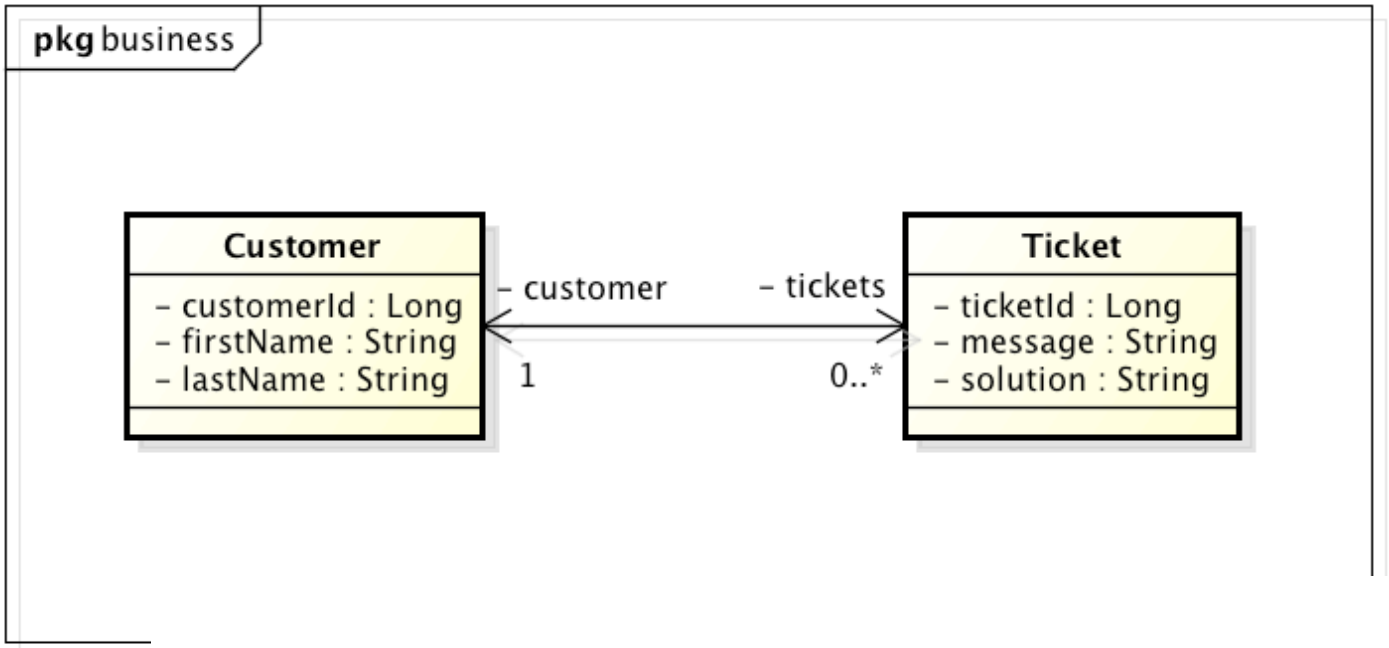
bidirectionnelle

Relation sur les clés

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer {
   @Id @Column(name = "CUSTOMER_ID")
   private Long customerId;
   @OneToOne
   @PrimaryKeyJoinColumn(name="CUSTOMER_ID",
      referencedColumnName="CUSTOMER_BILLING_ID")
   private BillingInfo billingInfo;
...
}

@Entity
@Table(name = "BILLING_INFO")
public class BillingInfo {
   @Id @Column(name = "CUSTOMER_BILLING_ID")
   private Long userId;
  ...
}
```

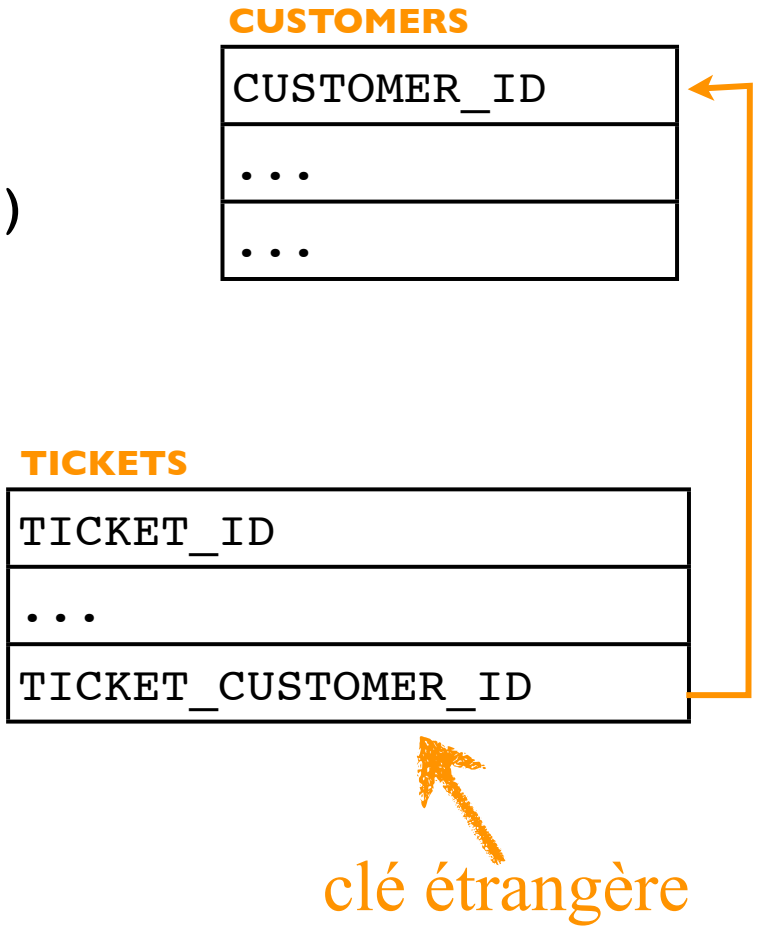


One-to-many/Many-to-one



- **représentée par une clé étrangère dans la table qui correspond au côté propriétaire**
  ➙ obligatoirement le côté Many pour les associations bidirectionnelles

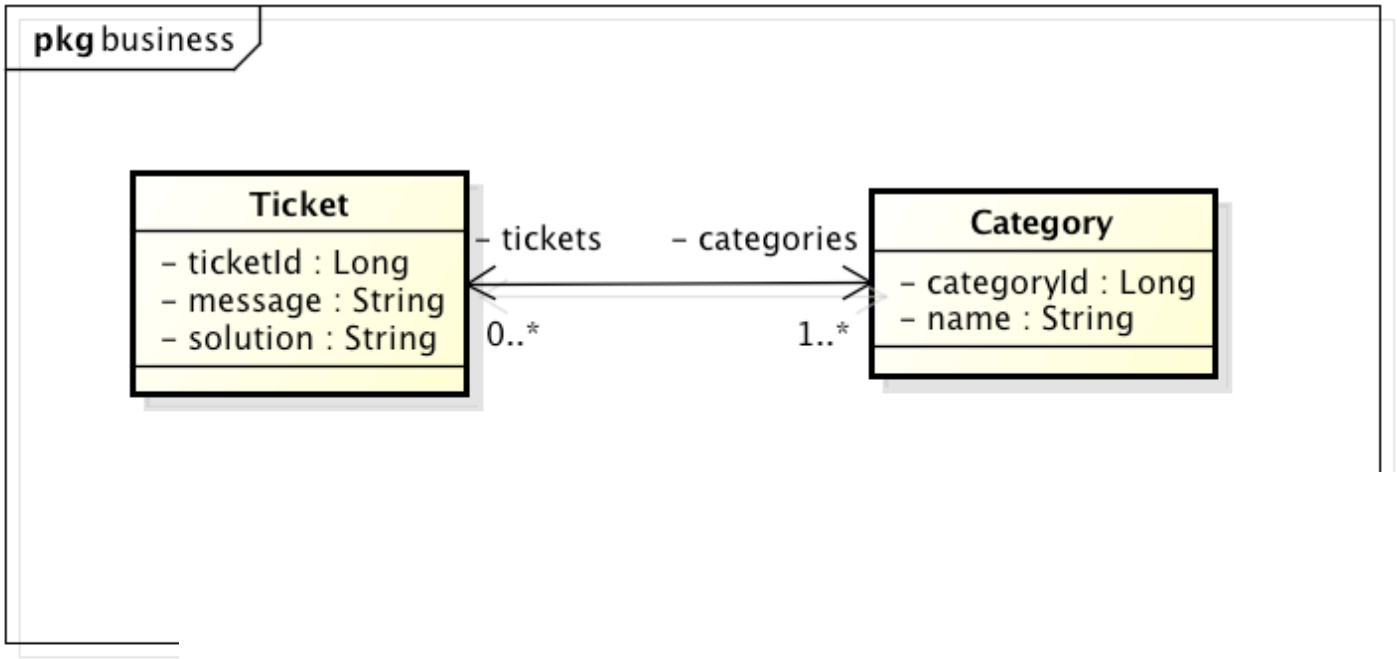
```java
@Entity
@Table(name="CUSTOMERS")
public class Customer {
   @Id @Column(name = "CUSTOMER_ID")
   private Long customerId;
   @OneToMany (mappedBy="customer")
   private Set<Ticket> tickets;
...
}

@Entity
@Table(name="TICKETS")
public class Ticket {
   @Id @Column(name = "TICKET_ID")
   private Long ticketId;
   @ManyToOne
   @JoinColumn(name="TICKET_CUSTOMER_ID",
               referencedColumnName="CUSTOMER_ID")
   private Customer customer;
... }
```



One-to-many unidirectionnelle

- par défaut, une association unidirectionnelle one-to-many est traduite pas une table association (comme pour many-to-many)
- elle n'est pas traduite par une clé étrangère dans la table du côté *many* comme c'est le cas pour une association bidirectionnelle

Many-to-many

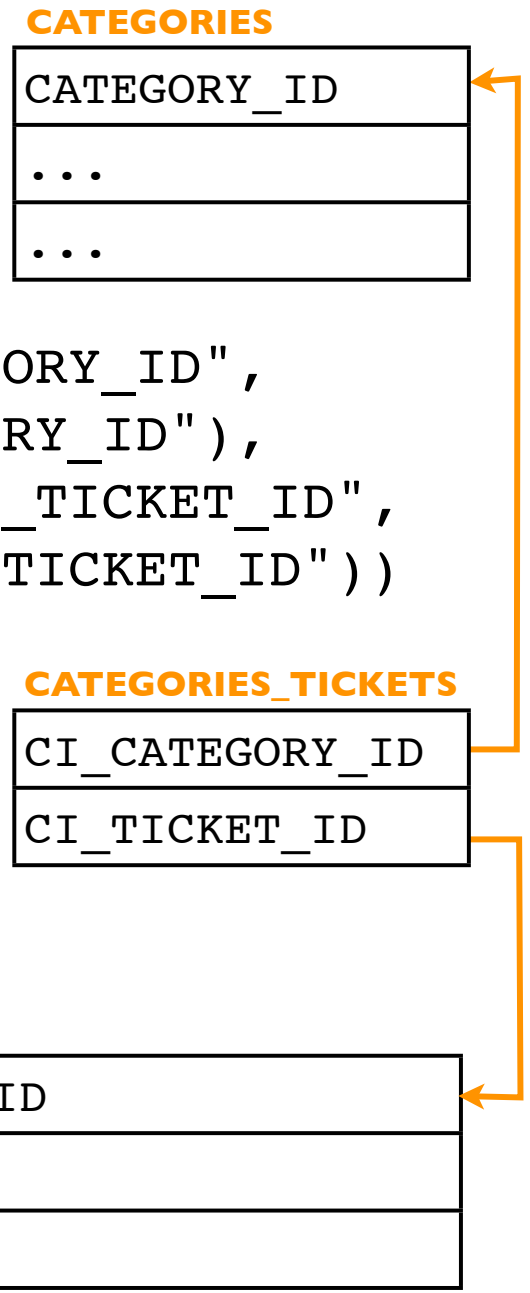


- représentée par une table association dans la base de données relationnelle

Many-to-many

- **valeurs par défaut**
  - nom de la table association = nom de la table de l'entité propriétaire + `_` + nom de la table de l'entité non-propriétaire
  - noms des colonnes clés étrangères = nom de l'attribut qui pointe vers l'autre entité + `_` + la colonne `Id` de la table référencée
- **annotation `@JoinTable` du côté propriétaire pour changer les valeurs par défaut**

```
@Entity
@Table(name="CATEGORIES")
public class Category {
  @Id @Column(name="CATEGORY_ID")
  private Long categoryId;
  @ManyToMany
  @JoinTable(name="CATEGORIES_TICKETS",
    joinColumns= @JoinColumn(name="CI_CATEGORY_ID",
                 referencedColumnName="CATEGORY_ID"),
    inverseJoinColumns=@JoinColumn(name="CI_TICKET_ID",
                       referencedColumnName="TICKET_ID"))
  private Set<Ticket> tickets;
...}

@Entity
@Table(name="TICKETS")
public class Ticket {
  @Id @Column(name = "TICKET_ID")
  private Long ticketId;
  @ManyToMany (mappedBy="tickets")
  private Set<Category> categories;
... }
```



@JoinTable

- `name` → le nom de la table
- `joinColumns` → noms des colonnes de la table qui référencent les clés primaires du côté propriétaire de l'association
- `inverseJoinColumns` → noms des colonnes de la table qui référencent les clés primaires du côté qui n'est pas propriétaire de l'association

Héritage

- Stratégies
  - une seule table pour une hiérarchie d'héritage ➙ `SINGLE_TABLE`
  - une table par classe ; les tables sont jointes pour reconstituer les données ➙ `JOINED`
  - une table distincte par classe concrète ➙ `TABLE_PER_CLASS`

SINGLE_TABLE

- valeur par défaut de la stratégie de traduction de l'héritage
- performante et permet le polymorphisme
- beaucoup de valeurs NULL dans les colonnes si la hiérarchie est complexe
- l'ajout d'un nouveau type d'entité après le démarrage de l'application oblige à modifier la définition de la table qui contient les données

default : DTYPE

```
@Entity
@Table(name="CUSTOMERS")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="CUSTOMER_TYPE",
     discriminatorType=DiscriminatorType.STRING,
     length=1)
public abstract class Customer { ... }
```

default : Company

```
@Entity
@DiscriminatorValue(value="C")
public class Company extends Customer { ... }
```

default : Individual

```
@Entity
@DiscriminatorValue(value="I")
public class Individual extends Customer { ... }
```

JOINED

- utilise des relations one-to-one pour modéliser l'héritage
- l'ajout d'un nouveau type d'entité après le démarrage de l'application réalisé par une nouvelle table
- pertes de performance pas significatives

```
@Entity
@Table(name="CUSTOMERS")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="CUSTOMER_TYPE",
     discriminatorType=DiscriminatorType.STRING,length=1)
public abstract class Customer { ... }


@Entity
@Table(name="COMPANIES")
@DiscriminatorValue(value="C")
@PrimaryKeyJoinColumn(name="CUSTOMER_ID")
public class Company extends Customer { ... }


@Entity
@Table(name="INDIVIDUALS")
@DiscriminatorValue(value="I")
@PrimaryKeyJoinColumn(name="CUSTOMER_ID")
public class Individual extends Customer { ... }
```

Questions ?