

Mapping Objet-Relationnel

Java EE 6 - Sommaire

- Introduction
- Exemple simple
- Contraintes des entités
- Personnalisation simples
- Personnalisation par XML
- Relations
- Héritage

Mapping : Introduction

- Les applications ont une couche modèle, on parle aussi d'objets du domaine. Cette couche exploite toutes les possibilités de l'orienté objet
- Les bases de données relationnelles ont des façons différentes de représenter les données et les relations entre objets. Elles n'ont pas non plus les concepts d'héritage
- Il faut donc arriver à transformer les données d'un monde objet à un monde relationnel et inversement

Mapping : Exemple simple

- Une entité est une simple classe Java (POJO) annotée `@Entity` avec des attributs qui définissent son état (les données à sauvegarder) et des accesseurs pour les manipuler (getters et setters)
- Chaque attribut est stocké dans la colonne d'une table portant le nom de l'entité

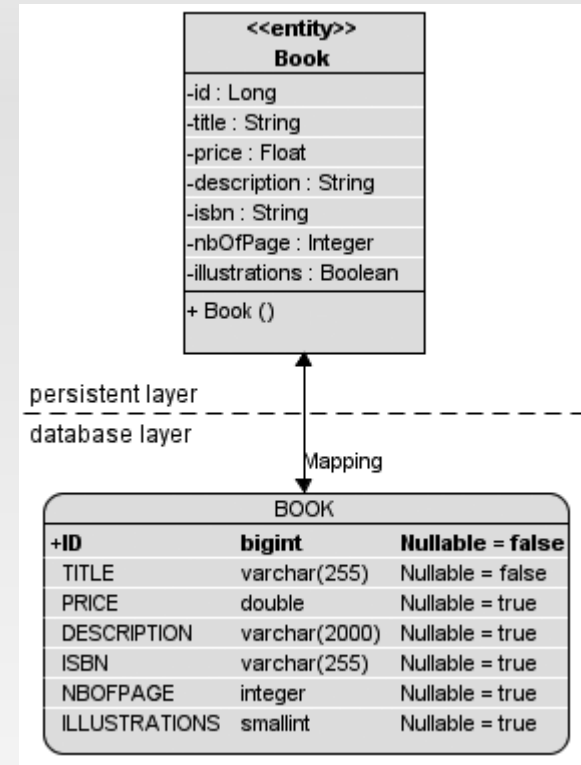
Mapping : Exemple simple

```
@Entity
public class Book {

    @Id
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;

    public Book() {
    }

    // Getters, setters
}
```



Mapping : Contraintes des entités

- Pour être une entité, une classe doit :
 - Être annotée avec `@javax.persistence.Entity` (ou être listé dans le descripteur XML)
 - Avoir un attribut annoté avec `@javax.persistence.Id` pour définir la clé primaire
 - Avoir au moins un constructeur sans argument et public ou protected
 - Être une classe, pas un enum ou une interface
 - Ne pas être finale. Aucune méthode ou variable d'instance persistante ne doit être finale non plus
 - Si l'entité doit être passée par valeur (ex via l'interface Remote), implémenter Serializable

Mapping : Configuration par l'exception

- Règles de configuration par l'exception :
 - Le nom de l'entité est mappé avec une classe du même nom en majuscule : Book → BOOK
Pour changer le nom, utiliser l'annotation @Table
 - Le nom des attributs est mappé avec une colonne du même nom en majuscule : id → ID
Pour changer le nom, utiliser l'annotation @Column
 - Les types primitifs Java suivent les conversion JDBC : String → VARCHAR(255), Long → BIGINT, Boolean → SMALLINT, ... Mais cela varie en fonction de la base String → VARCHAR2 sur Oracle

Mapping : Personnalisations simples

- Tables

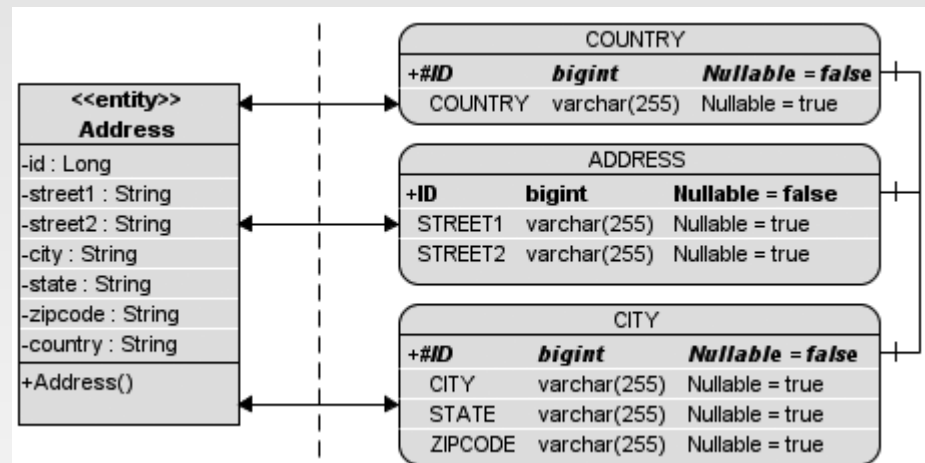
- Par défaut le nom de la table est celui de l'entité
- `@javax.persistence.Table` permet de changer la valeur par défaut

```
@Entity
@Table(name = "t_book")
public class Book {
```

- `@javax.persistence.SecondaryTable` permet d'avoir des tables secondaires associées à l'entité, par exemple pour grouper dans des tables dédiées certains attributs ou groupes d'attributs

Mapping : Personnalisations simples

```
@Entity
@SecondaryTables({
    @SecondaryTable(name = "city"),
    @SecondaryTable(name = "country")
})
public class Address {
    @Id
    private Long id;
    private String street1;
    private String street2;
    @Column(table = "city")
    private String city;
    @Column(table = "city")
    private String state;
    @Column(table = "city")
    private String zipcode;
    @Column(table = "country")
    private String country;
    // Constructors, getters, setters
}
```



Mapping : Personnalisations simples

- Clés primaires
 - Une clé primaire identifie de façon unique une ligne d'une table
 - Elle peut être constituée d'une ou plusieurs colonnes
 - Elle doit être unique
 - JPA oblige les entités à définir une clé primaire
 - On utilise `@Id` pour les clés primaires simples
 - On utilise `@EmbeddedId` ou `@IdClass` pour les clés composite

Mapping : Personnalisations simples

- Clé primaire simple :

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    // Constructors, getters, setters
}
```

Mapping : Personnalisations simples

- On peut placer l'annotation `@Id` sur l'un des types suivants :
 - Types primitifs : `byte`, `int`, `short`, `long`, `char`
 - Wrapper de types primitifs : `Byte`, `Integer`, ...
 - Tableaux de types primitifs ou wrappers : `int[]`, ...
 - String, nombres et dates : `java.lang.String`, `java.math.BigInteger`, `java.util.Date`, `java.sql.Date`

Mapping : Personnalisations simples

- Attributs
 - Presque tous les types Java peuvent être mappés et persistés
 - Types primitifs
 - Tableaux de types primitifs
 - String, nombres et types temporels
 - Types énumérés et types personnalisés qui implémentent Serializable
 - Collections de types simples ou embeddable
 - On utilise principalement @Basic, @Column, @Temporal, @Transient, @Enumerated pour les personnaliser

Mapping : Personnalisations simples

- @javax.persistence.Basic

```
@Entity
public class Track {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float duration;
    @Basic(fetch = FetchType.LAZY)
    @Lob
    private byte[] wav;
    @Basic(optional = true)
    private String description;
    // Constructors, getters, setters
}
```

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Basic {
    FetchType fetch() default EAGER;
    boolean optional() default true;
}
```

Mapping : Personnalisations simples

■ @Column

@Entity

```
public class Book {  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
    @Column(name = "book_title", nullable = false,  
            updatable = false)  
    private String title;  
    private Float price;  
    @Column(length = 2000)  
    private String description;  
    private String isbn;  
    @Column(name = "nb_of_page", nullable = false)  
    private Integer nbOfPage;  
    private Boolean illustrations;  
    // Constructors, getters, setters  
}
```

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)  
public @interface Column {  
    String name() default "";  
    boolean unique() default false;  
    boolean nullable() default true;  
    boolean insertable() default true;  
    boolean updatable() default true;  
    String columnDefinition() default "";  
    String table() default "";  
    int length() default 255;  
    int precision() default 0;    // decimal precision  
    int scale() default 0;       // decimal scale  
}
```

Mapping : Personnalisations simples

- @Temporal

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    // Constructors, getters, setters
}
```


Mapping : Personnalisations simples

- **@Transient**

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    // Constructors, getters, setters
}
```

Mapping : Personnalisations simples

■ @Enumerated

```
public enum CreditCardType {  
    VISA,  
    MASTER_CARD,  
    AMERICAN_EXPRESS  
}
```

```
@Entity  
@Table(name = "credit_card")  
public class CreditCard {  
    @Id  
    private String number;  
    private String expiryDate;  
    private Integer controlNumber;  
    private CreditCardType creditCardType;  
    // Constructors, getters, setters  
}
```

↓
INT

```
@Entity  
@Table(name = "credit_card")  
public class CreditCard {  
    @Id  
    private String number;  
    private String expiryDate;  
    private Integer controlNumber;  
    @Enumerated(EnumType.STRING)  
    private CreditCardType creditCardType;  
    // Constructors, getters, setters  
}
```

↓
VARCHAR

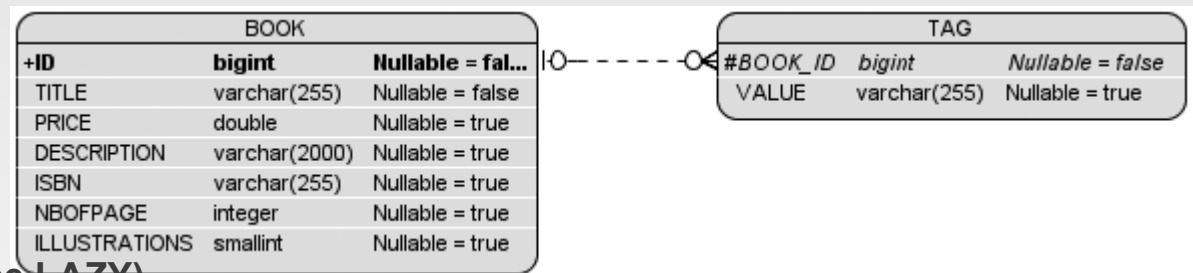
Mapping : Personnalisations simples

- `@ElementCollection` et `@CollectionTable`
 - `@ElementCollection` précise qu'un attribut est une collection de types Java
 - `@CollectionTable` permet de configurer la table créée, par exemple son nom
 - On peut les utiliser à la fois pour les List et les Map. Pour les Map on pourra ajouter l'annotation `@MapKeyColumn`

Mapping : Personnalisations simples

- List : @ElementCollection et @CollectionTable

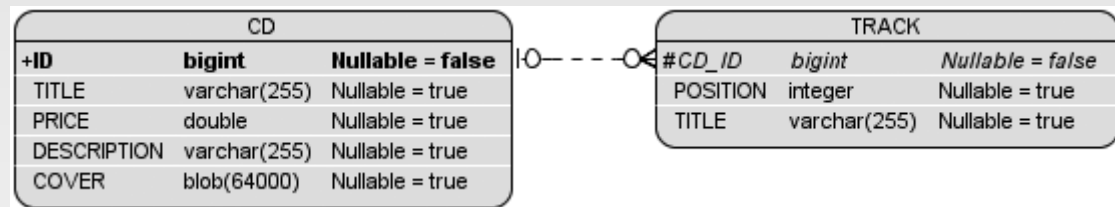
```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "Tag")
    @Column(name = "Value")
    private ArrayList<String> tags;
    // Constructors, getters, setters
}
```



Mapping : Personnalisations simples

- Map : @ElementCollection et @CollectionTable

```
@Entity
public class CD {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    @Lob
    private byte[] cover;
    @ElementCollection
    @CollectionTable(name="track")
    @MapKeyColumn(name="position")
    @Column(name="title")
    private Map<Integer, String> tracks;
    // Constructors, getters, setters
}
```



Mapping : Personnalisations simples

- @Embeddable et @Embedded

@Entity

```
public class Customer {  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    private String email;  
    private String phoneNumber;  
    @Embedded  
    private Address address;  
    // Constructors, getters, setters  
}
```

@Embeddable

```
public class Address {  
    private String street1;  
    private String street2;  
    private String city;  
    private String state;  
    private String zipcode;  
    private String country;  
    // Constructors, getters, setters  
}
```

```
create table CUSTOMER (  
    ID BIGINT not null,  
    LASTNAME VARCHAR(255),  
    PHONENUMBER VARCHAR(255),  
    EMAIL VARCHAR(255),  
    FIRSTNAME VARCHAR(255),  
    STREET2 VARCHAR(255),  
    STREET1 VARCHAR(255),  
    ZIPCODE VARCHAR(255),  
    STATE VARCHAR(255),  
    COUNTRY VARCHAR(255),  
    CITY VARCHAR(255),  
    primary key (ID)  
);
```

Mapping : personnalisation par XML

- On peut définir le mapping avec un fichier de description XML comme avec les annotations
- Si le même mapping est décrit en annotations et en XML, c'est le XML qui détermine le mapping
- Généralement on utilise les annotations
- XML est intéressant si les paramètres dépendent de l'environnement et qu'on les fait changer automatiquement par exemple grâce aux profils Maven


Mapping : personnalisation par XML

book_mapping.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                  version="2.0">
  <entity class=" com.apress.javaee6.chapter05.Book">
    <table name="book_xml_mapping"/>
    <attributes>
      <basic name="title">
        <column name="book_title" nullable="false" updatable="false"/>
      </basic>
      <basic name="description">
        <column length="2000"/>
      </basic>
      <basic name="nbOfPage">
        <column name="nb_of_page" nullable="false"/>
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

Book.java

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy =
        GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    @Column(length = 500)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    // Constructors, getters, setters
}
```



```
create table BOOK_XML_MAPPING (
  ID BIGINT not null,
  BOOK_TITLE VARCHAR(255) not null,
  DESCRIPTION VARCHAR(2000),
  NB_OF_PAGE INTEGER not null,
  PRICE DOUBLE(52, 0),
  ISBN VARCHAR(255),
  ILLUSTRATIONS SMALLINT,
  primary key (ID)
);
```

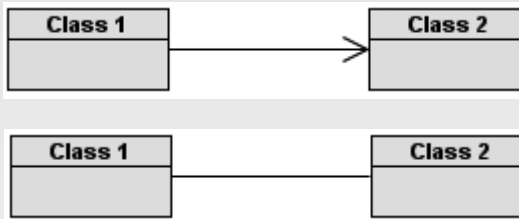

Mapping : personnalisation par XML

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
  <persistence-unit name="javaee6PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>com.apress.javaee6.chapter05.Book</class>
    <mapping-file>META-INF/book_mapping.xml</mapping-file>
    <properties>
      <!--Persistence provider properties-->
    </properties>
  </persistence-unit>
</persistence>
```

Mapping : Relations

- Dans le monde objet, les objets ont des relations définies par :
 - Une direction : unidirectionnelle ou bidirectionnelle



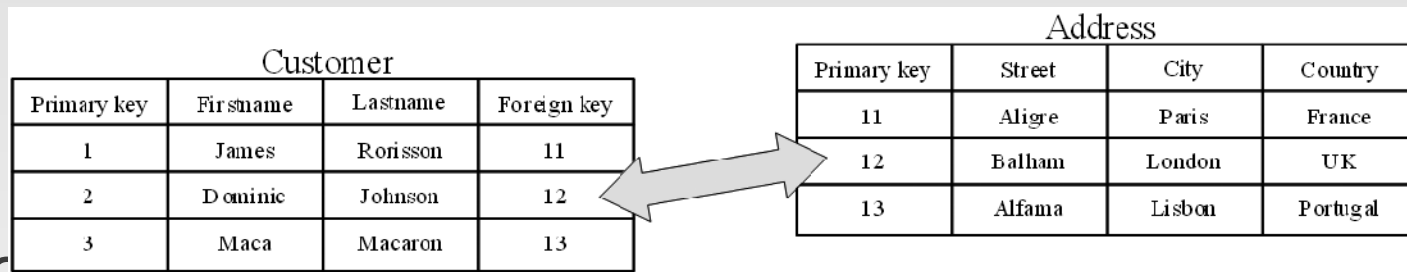
- Une cardinalité



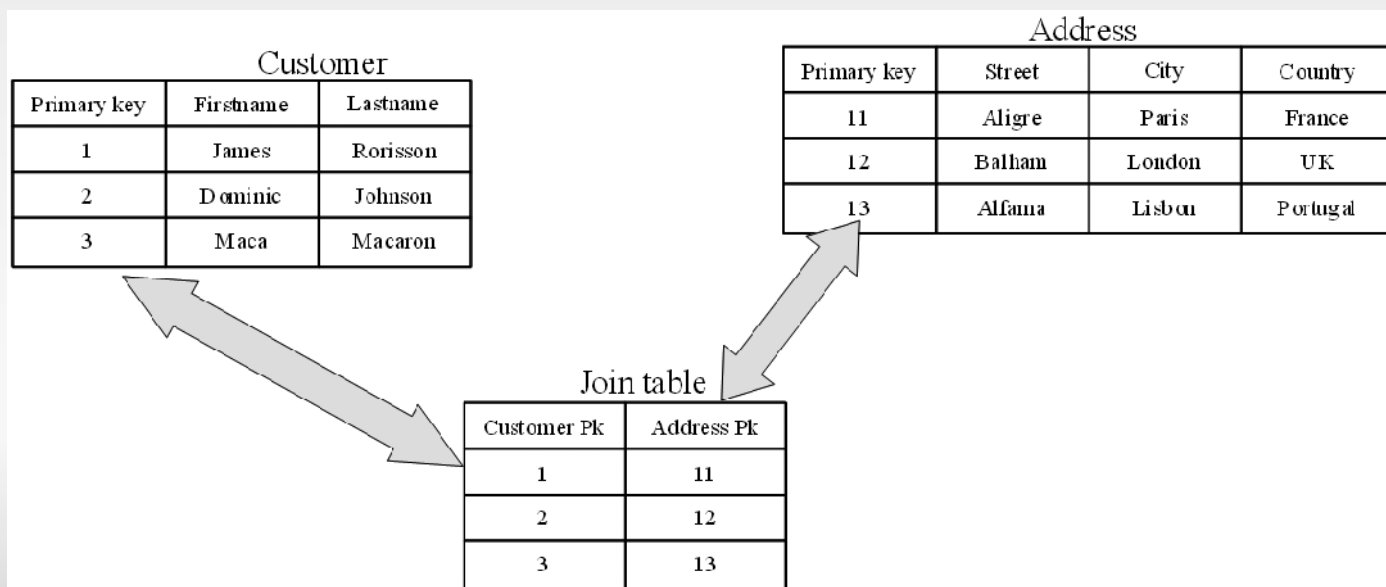
- Un responsable : implicite pour unidirectionnel

Mapping : Relations

- Dans le monde relationnel, les relations se traduisent par :
 - Des clés étrangères (foreign key ou join column)



- Des tables de jointure (join table)



Mapping : Relations

- JPA permet de traduire les notions de cardinalité et de direction, si bien qu'on a toutes les possibilités suivantes :

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

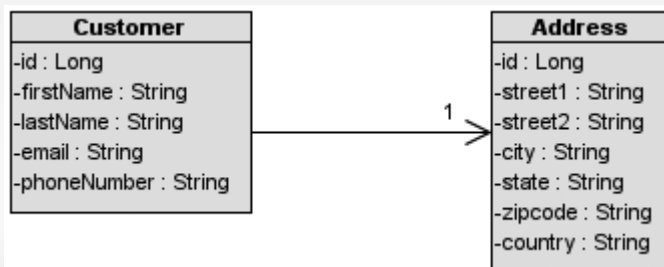
Mapping : Relations

- @OneToOne unidirectionnelle (par défaut)

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    private Address address;
    // Constructors, getters, setters
}
```

```
@Entity
public class Address {
    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    // Constructors, getters, setters
}
```

```
create table CUSTOMER (
    ID BIGINT not null,
    FIRSTNAME VARCHAR(255),
    LASTNAME VARCHAR(255),
    EMAIL VARCHAR(255),
    PHONENUMBER VARCHAR(255),
    ADDRESS_ID BIGINT,
    primary key (ID),
    foreign key (ADDRESS_ID)
        references ADDRESS(ID)
);
```



Mapping : Relations

- @OneToOne unidirectionnelle
 - Pour personnaliser la relation, on peut utiliser les annotations @OneToOne et @JoinColumn

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne (fetch = FetchType.LAZY)
    @JoinColumn(name = "add_fk", nullable = false)
    private Address address;
    // Constructors, getters, setters
}
```

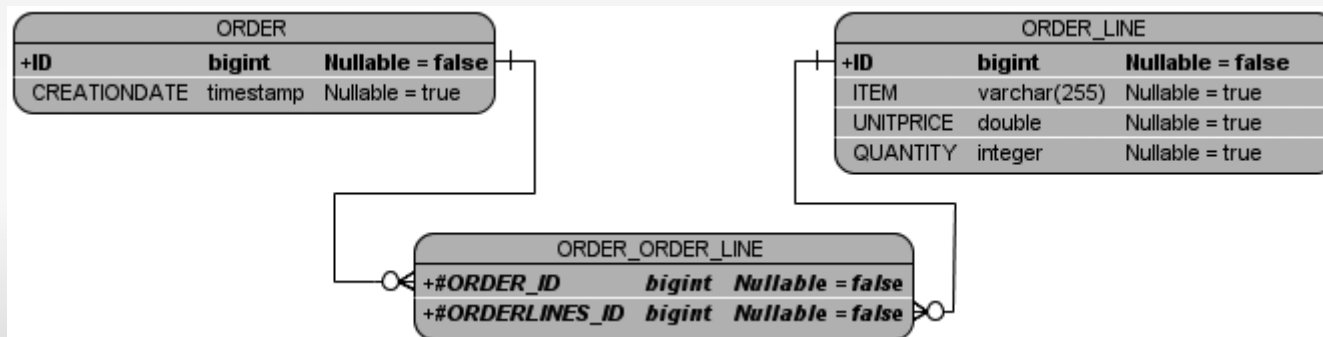
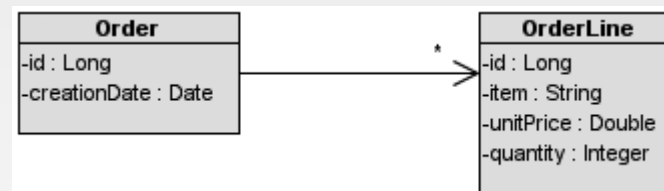
```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
    boolean orphanRemoval() default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

Mapping : Relations

- @OneToMany unidirectionnelle (par défaut)

```
@Entity
public class Order {
    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    private List<OrderLine> orderLines;
    // Constructors, getters, setters
}
```

```
@Entity
@Table(name = "order_line")
public class OrderLine {
    @Id @GeneratedValue
    private Long id;
    private String item;
    private Double unitPrice;
    private Integer quantity;
    // Constructors, getters, setters
}
```



Mapping : Relations

- @OneToMany unidirectionnelle
 - Pour personnaliser la relation, on peut utiliser les annotations @OneToMany et @JoinTable ou @JoinColumn

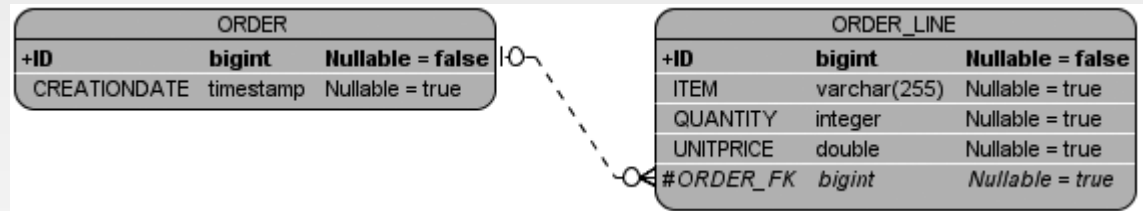
```
@Entity
public class Order {
    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany
    @JoinTable(name = "jnd_ord_line",
        joinColumns = @JoinColumn(name = "order_fk"),
        inverseJoinColumns = @JoinColumn(name = "order_line_fk") )
    private List<OrderLine> orderLines;
    // Constructors, getters, setters
}
```

```
create table JND_ORD_LINE (
    ORDER_FK BIGINT not null,
    ORDER_LINE_FK BIGINT not null,
    primary key (ORDER_FK, ORDER_LINE_FK),
    foreign key (ORDER_LINE_FK)
        references ORDER_LINE(ID),
    foreign key (ORDER_FK)
        references ORDER(ID)
);
```


Mapping : Relations

- **@OneToMany** unidirectionnelle
 - Pour ne pas utiliser une table de jointure (comportement par défaut), utiliser **@JoinColumn**

```
@Entity
public class Order {
    @Id @GeneratedValue
    private Long id;
    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;
    @OneToMany(fetch = FetchType.EAGER)
    @JoinColumn(name = "order_fk")
    private List<OrderLine> orderLines;
    // Constructors, getters, setters
}
```

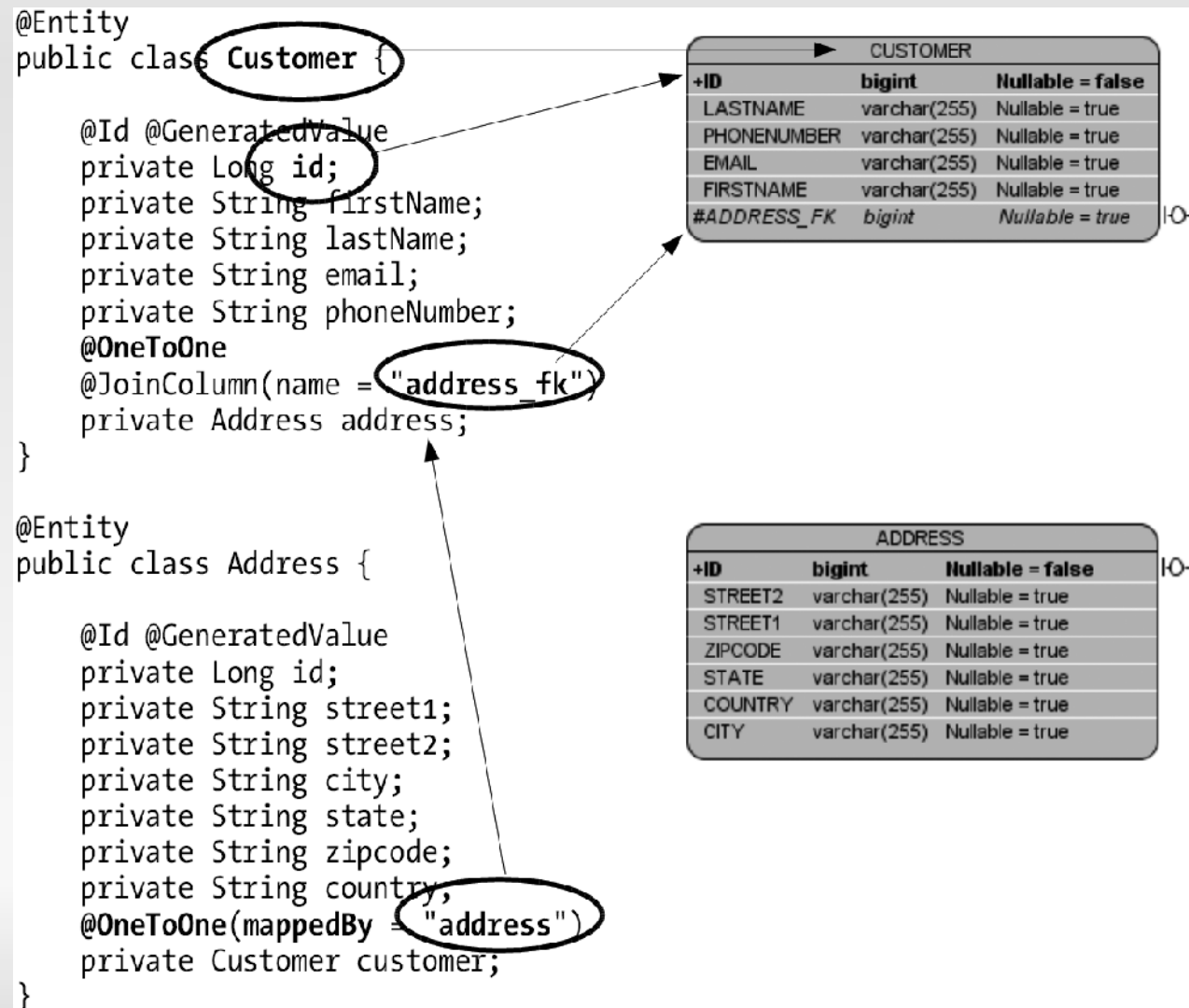


Mapping : Relations

- Dans une relation bidirectionnelle, il faut définir un responsable de la relation
 - Le responsable utilise `@JoinColumn` ou `@JoinTable` pour définir la jointure
 - L'autre utilise l'attribut `mappedBy` pour renvoyer à l'autre bout de la relation pour la définition de la relation. Il y a un attribut `mappedBy` pour les annotation `@OneToOne`, `@OneToMany` et `@ManyToMany` (mais pas sur `@ManyToOne`)

Mapping : Relations

- @OneToOne bidirectionnelle



Mapping : Relations

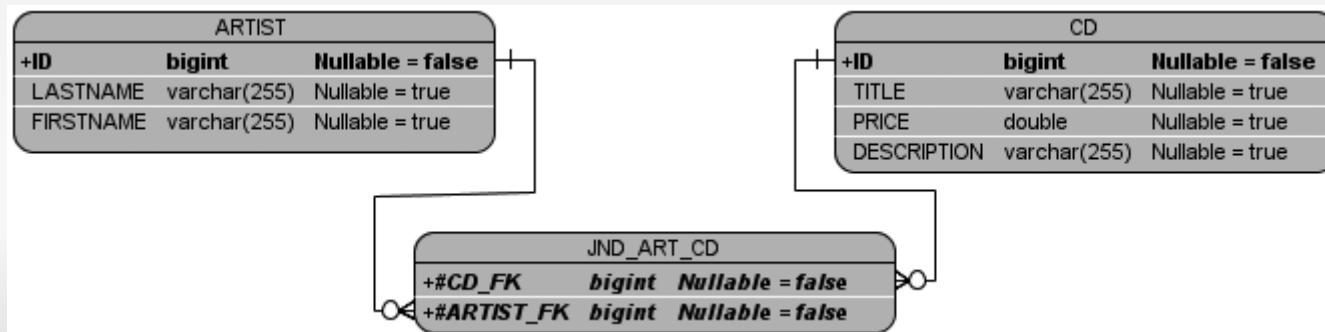
- @ManyToMany bidirectionnelle

@Entity

```
public class Artist {  
    @Id @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    @ManyToMany  
    @JoinTable(name = "jnd_art_cd", ↵  
        joinColumns = @JoinColumn(name = "artist_fk"), ↵  
        inverseJoinColumns = @JoinColumn(name = "cd_fk"))  
    private List<CD> appearsOnCDs;  
    // Constructors, getters, setters  
}
```

@Entity

```
public class CD {  
    @Id @GeneratedValue  
    private Long id;  
    private String title;  
    private Float price;  
    private String description;  
    @ManyToMany(mappedBy = "appearsOnCDs")  
    private List<Artist> createdByArtists;  
    // Constructors, getters, setters  
}
```

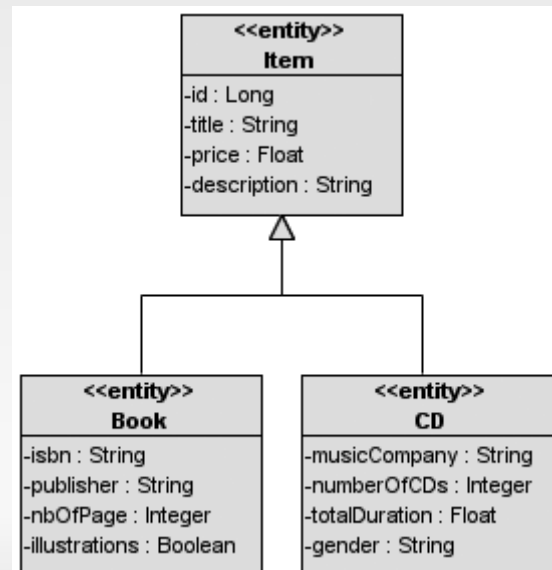


Mapping : Héritage

- Le principe d'héritage est absent du monde relationnel. Pour le simuler, JPA propose trois stratégies :
 - Single-table strategy : une seule table par hiérarchie : stratégie par défaut. Tous les attributs de la hiérarchie dans une seule table
 - Joined strategy : une table par classe concrète ou abstraite
 - Table-per-class strategy : une table par classe concrète et indépendante : stratégie toujours optionnelle dans JPA 2.0. Peut poser des problèmes si besoin de portabilité

Mapping : Héritage

- La classe de base peut utiliser l'annotation `@Inheritance` pour définir la stratégie d'héritage. Si elle ne le fait pas, la stratégie par défaut est utilisée (single-table : une table par hiérarchie)



Mapping : Héritage

- Single-table strategy : une table par hiérarchie

```
@Entity
public class Item {
    @Id @GeneratedValue
    protected Long id;
    @Column(nullable = false)
    protected String title;
    @Column(nullable = false)
    protected Float price;
    protected String description;
    // Constructors, getters, setters
}
```

ITEM		
+ID	bigint	Nullable = false
DTYPE	varchar(31)	Nullable = true
TITLE	varchar(255)	Nullable = false
PRICE	double	Nullable = false
DESCRIPTION	varchar(255)	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDs	integer	Nullable = true
TOTALDURATION	double	Nullable = true
GENDER	varchar(255)	Nullable = true

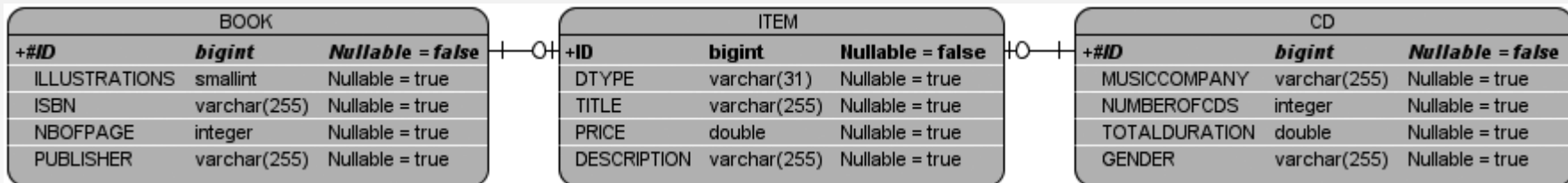
```
@Entity
public class Book extends Item {
    private String isbn;
    private String publisher;
    private Integer nbOfPage;
    private Boolean illustrations;
    // Constructors, getters, setters
}
```

```
@Entity
public class CD extends Item {
    private String musicCompany;
    private Integer numberOfCDs;
    private Float totalDuration;
    private String gender;
    // Constructors, getters, setters
}
```

Mapping : Héritage

- Joined strategy : une table par classe

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {
    @Id @GeneratedValue
    protected Long id;
    protected String title;
    protected Float price;
    protected String description;
    // Constructors, getters, setters
}
```



Mapping : Héritage

- Table-per-class strategy : une table par classe

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Item {
    @Id @GeneratedValue
    protected Long id;
    protected String title;
    protected Float price;
    protected String description;
    // Constructors, getters, setters
}
```

BOOK		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
ILLUSTRATIONS	smallint	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
ISBN	varchar(255)	Nullable = true
NBOFPAGE	integer	Nullable = true
PUBLISHER	varchar(255)	Nullable = true

ITEM		
+ID	bigint	Nullable = false
TITLE	varchar(255)	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true

CD		
+ID	bigint	Nullable = false
MUSICCOMPANY	varchar(255)	Nullable = true
NUMBEROFCDS	integer	Nullable = true
TITLE	varchar(255)	Nullable = true
TOTALDURATION	double	Nullable = true
PRICE	double	Nullable = true
DESCRIPTION	varchar(255)	Nullable = true
GENDER	varchar(255)	Nullable = true

Mapping : Héritage

Stratégie	Avantages	Inconvénients
Single-table	Facile à comprendre Bien pour de petites hiérarchies stables	Difficile d'ajouter des niveaux à la hiérarchie ou des attributs Les colonnes des fils ne peuvent être nullable
Joined-table	Intuitif	Impacts de performances (jointures)
Table-per-class	Fonctionne bien avec des requêtes confinées à une table	Les requêtes sur les hiérarchies sont plus longues (union)