Taller de Persistencia con Java

Encuentro 3:

"Mapeo Objeto Relacional"

La anotación @Column

- Describe como un campo particular es mapeado a una columna especifica en una tabla.

```
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;
    int     scale( ) default 0;
}
```

La anotación @Column

- *name()* especifica el nombre de la columna, Por defecto es el nombre de la propiedad de la clase.
- *table()* es utilizado cuando se realiza mapeo de una clase, en más de una tabla.
- *unique()* y *nullable()* define las restricciones que se implementaran sobre la columna, *unique* agrega una clave única sobre la columna, y *nullable* permite asignar valores nulos.
- Mediante *insertable() y updatable()* se indica si la columna debe ser incluida en los SQL de INSERT o UPDATE generados por el *Entity Manager*.

Ejemplo

```
1.  @Entity
2.  public class Book {
3.  @Id @GeneratedValue(strategy = GenerationType.AUTO)
4.  private Long id;
5.  @Column(name = "book_title", nullable = false, updatable =
    false)
6.  private String title;
7.  private Float price;
8.  @Column(length = 2000)
9.  private String description;
10. private String isbn;
11. @Column(name = "nb_of_page", nullable = false)
12. private Integer nbOfPage;
13. private Boolean illustrations;
14. // Constructors, getters, setters
15. }
```

Mapeos de columnas

- **@Transient** permite especificar que la propiedad de un entidad que no deben persistirse. A estos atributos, el EntityManager simplemente los ignora .
- La anotación ***@Basic*** es el mapeo por defecto de los tipos de datos de los atributos persistentes y se puede utilizar sobre los tipo de datos Java primitivos y sus clases Wrapper (*java.lang.String, byte[], Byte[], char[], Character[], java.math.BigInteger, java.math.BigDecimal,java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time , and java.sql.Timestamp*)

Anotación Basic y Carga de atributos

- Habitualmente no se necesita utilizar esta anotación, aunque a veces, se requiere especificar el atributo *fetch(),* el cual permite especificar si una propiedad en particular debe ser cargado de determinada manera .

```
1. public @interface Basic {
2. FetchType fetch( ) default EAGER;
3. boolean optional( ) default true;
4. }
5. public enum FetchType{
6.    LAZY, EAGER
7. }
```

Tipos de Fetch – LAZY / EAGER

- Cuando una entidad es cargada de la base de datos por el entity manager, todos los campos son cargados con el valor que poseen en el respectivo registro.
- Aquí se puede modificar la estrategia por defecto y:
  - Cargar datos que son usados frecuentemente → siempre que se cargue la entidad (EAGER).
  - Cargar atributos que se usan esporádicamente → cuando se necesite dicho atributo (LAZY)
- La anotación "basic" posee un atributo denominado "*fetch type*" que permite configurar el tipo de carga deseada para cada atributo.
- *Los atributos marcados como LAZY, cuando se carga la entidad mantienen un valor NULL.*
- *Usarse con cuidado* **Mejor Performance si hay muchas columnas o campos muy grandes**

Mapeo para objetos grandes

- JDBC provee un tipo de datos especial para trabajar con objetos de gran tamaño→java.sql.Blob se utiliza para representar datos binarios y el tipo java.sql.Clob para representar caracteres.
- La anotación javax.persistence.Lob es utilizada para marcar estos tipos de datos.
- De acuerdo al tipo de datos del atributo se persisten en:
  - Blob si el tipo de datos es byte[], Byte[], or Serializable
  - Clob si el tipo de dato Java es char[], Character[], or String.

Mapeo para objetos grandes

- Habitualmente se utiliza @Lob en combinación con @Basic para indicar que el atributo sea cargado en forma tardía (LAZY)

```
1. @Entity
2. public class Employee {
3. @Id
4. private int id;
5. @Basic(fetch=FetchType.LAZY)
6. @Lob @Column(name="PIC")
7. private byte[] picture;
8. // ...
9. }
```

Tipos Temporales

- Los tipos temporales son un conjunto de clases que pueden ser empleadas para manejar fechas y horas en soporte persistente.
- Las clases java soportadas son:
  - java.sql types java.sql.Date
  - java.sql.Time
  - java.sql.Timestamp
  - java.util.Date
  - java.util.Calendar
- Los tres tipos que pertenecen al paquete sql, no presentan ninguna particularidad respecto a cualquier otro tipo de datos, pero no son los que habitualmente se emplean para el desarrollo.
- Las clases de la librería java.util por su parte necesitan algún tipo de adaptación para indicar a que tipo de SQL se deben convertir.

Tipos Temporales

- Anotación @Temporal y el atributo TemporalType
  - DATE
  - TIME
  - TIMESTAMP
- Por defecto, el proveedor de persistencia asume timestamp como tipo de datos para la columna

```
@Entity
public class Employee {
@Id
private int id;
@Temporal(TemporalType.DATE)
private Calendar dob;
@Temporal(TemporalType.DATE)
@Column(name="S_DATE")
private Date startDate;
// ...
}
```

Mapeo para colecciones de tipos básicos

- En general las colecciones se utilizan para modelar asociaciones entre objetos de cardinalidad mayor a 1.
  - Cuando las asociaciones son con tipos del modelo como entidades tienen un mapeo específico.
  - Si la asoicación es con un tipo primitivo, una fecha, un string o algun otro tipo del API de java que sea persistido en una DB el mapeo es un tanto diferente.
  - ¿Si para una persona quiero guardar una lista de correos electrónicos, necesito crear una entidad correo electrónico?
- Desde JPA 2.0 existen algunas anotaciones que permiten separar estos conceptos que no sufrieron alteracioens en JPA 2.1
  - @ElementCollection
  - @CollectionTable.

@ElementCollection

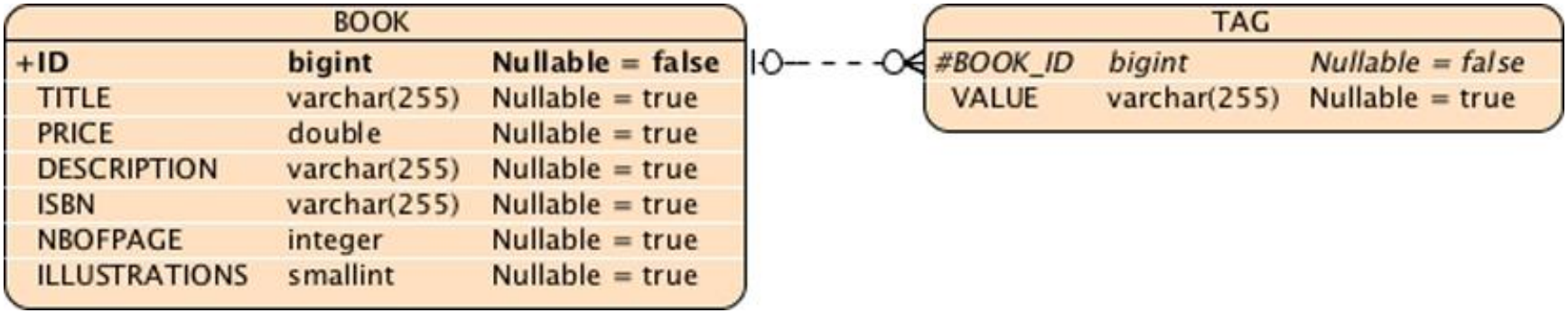
- Indica que un atributo del tipo java.util.Collection o cualaquiera de sub subclases, poseen un conjunto de intancias “no entidades”.
- Los atributos que lleven esta anotación pueden ser
  - java.util.Collection - java.util.Set - java.util.List.
- La idea es que una lista de entidades vinculadas se guarde en una segunda tabla y que esta relación sea manejada automáticamente por JPA.

@CollectionTable

- La anotación @CollectionTable permite personalizar los detalles de la tabla secundaria en donde se almacenará la colección.
  - Podemos definir entre otras cosas el nommbre.
  - Si esta anotación no está presente, el nombre automáticamente lo dará el proveedor de persistencia y consistirá en la concatenación del nombre de la entidad que tiene la colección y el atributo de tipo colección, separados por un guión bajo, lo cual puede dar lugar a nombres poco claros.
- Mostraremos un ejemplo donde a una entidad libro le asignaremos una serie de etiquetas.

Ejemplo

```
1.  @Entity
2.  public class Book {
3.  @Id @GeneratedValue
4.  private Long id;
5.  private String title;
6.  . . . .
7.  @ElementCollection(fetch = FetchType.LAZY)
8.  @CollectionTable(name = "Tag")
9.  @Column(name = "Value")
10. private List<String> tags = new ArrayList<>();
11. // Constructors, getters, setters
12. }
```



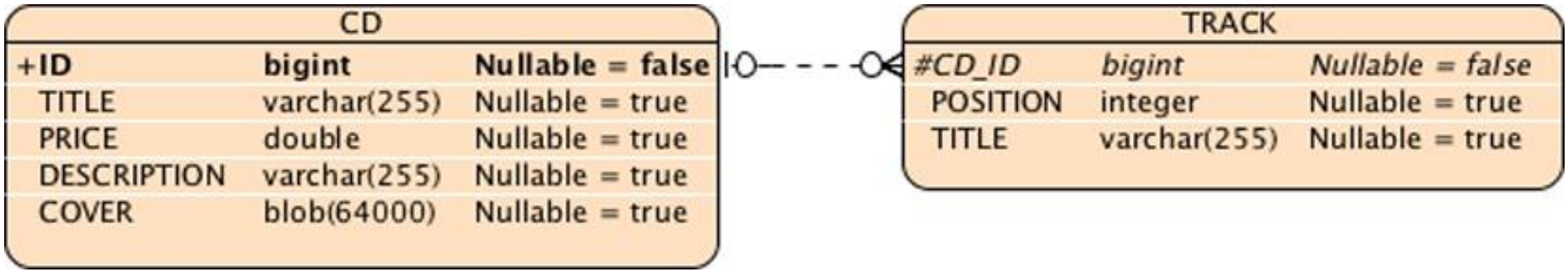
- En el ejemplo vemos que usamos la anotación @ElementCollection para informar al proveedor de persistencia que el atributo "tags" es una lista de Strings que debe ser recuperada de forma "lazy" de la base de datos (es decir a demanda).
- Si la anotación @CollectionTable no estuviese, el nombre por defecto de la tabla sería BOOK_TAGS, pero se ha configurado como "TAG". Además se la complementó con una anotación secundaria @Column, para indicar cual es el nombre de la columna en la tabla (TAGS).

Maps

- Otro elemento que no es una colección específicamente pero si forma parte del API de Collection, es la estructura de datos Map.
- En JPA 1.0 era posible mapearlos, pero de manera muy básica: las claves del mapa solo podían ser de un tipo básico (es decir no podían ser una entidad) y los valores solo podían ser entidades (no tipos básicos).
- En JPA 2.0 se introdujo una mejora y se permitió que cualquier combinación sea posible lo cual aumentó enormemente la flexibilidad.
- Cuando un map usa tipos básicos podemos usar las 2 colecciones vistas anteriormente @ElementCollection y @CollectionTable. Y se usará una "collection table" para guardar el map

Ejemplo

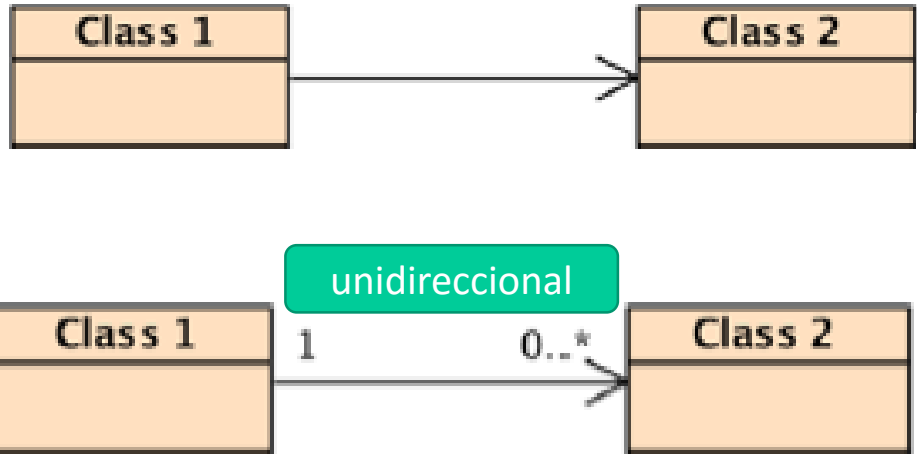

```
1.  @Entity
2.  public class CD {
3.  @Id @GeneratedValue
4.  private Long id;
5.  private String title;
6.  private Float price;
7.  private String description;
8.  @Lob
9.  private byte[] cover;
10. @ElementCollection
11. @CollectionTable(name="track")
12. @MapKeyColumn (name = "position")
13. @Column(name = "title")
14. private Map<Integer, String> tracks = new HashMap<>();
15. // Constructors, getters, setters
```

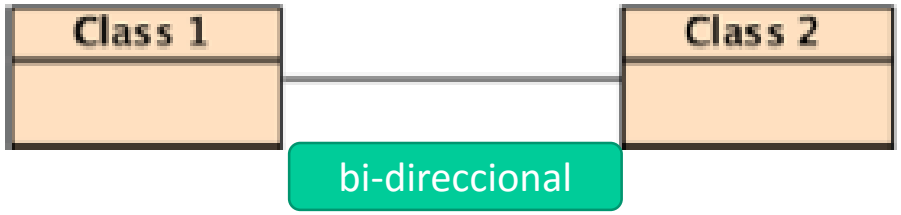


- En el ejemplo se muestra como se utiliza @ElementCollection para indicar que los objetos en el mapa son almacenados en una tabla de colección.
- La anotación @CollectionTable cambia el nombre por defecto de la tabla.
- Hasta aquí no hay variantes con las colecciones.
- La diferencia surge cuando se agrega la anotación @MapKeyColumn, la cual especifica el nombre de la columna que se corresponde con la clave del mapa.

Mapeo Objeto Relacional.

- Hasta aquí hemos analizado como vincular los atributos de un objeto con anotaciones, para que sean persistentes en la base de datos. ¿y las relaciones entre objetos?
- Las relaciones entre objetos son un elemento clave en la POO.
  - Son estructurales: un objeto “apunta” a otro.
- Las asociaciones se pueden caracterizar por:
  - Cardinalidad: cantidad de objetos que pueden estar relacionados.
  - Dirección: quien guarda un puntero de la relación (un lado o ambos).





Relaciones en una base de datos

1:1

Customer

Primary key	Firstname	Lastname	Foreign key
1	James	Rorisson	11
2	Dominic	Johnson	12
3	Maca	Macaron	13

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

1:N

Customer

Primary key	Firstname	Lastname
1	James	Rorisson
2	Dominic	Johnson
3	Maca	Macaron

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Join table

Customer PK	Address PK
1	11
2	12
3	13

Si un cliente puede tener muchas direcciones y una dirección puede ser la de muchos cliente la estructura se mantiene pero se le agrega una columna a la tabla de unión

En una base de datos, por definición todas las relaciones son “bidireccionales” por lo que no existe el concepto de visibilidad.

Combinaciones posibles

Cardinalidad	Direction
One-to-one	**Unidirectional**
One-to-one	**Bidirectional**
One-to-many	Unidirectional
One-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-one	Bidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

@OneToOne Unidirectional

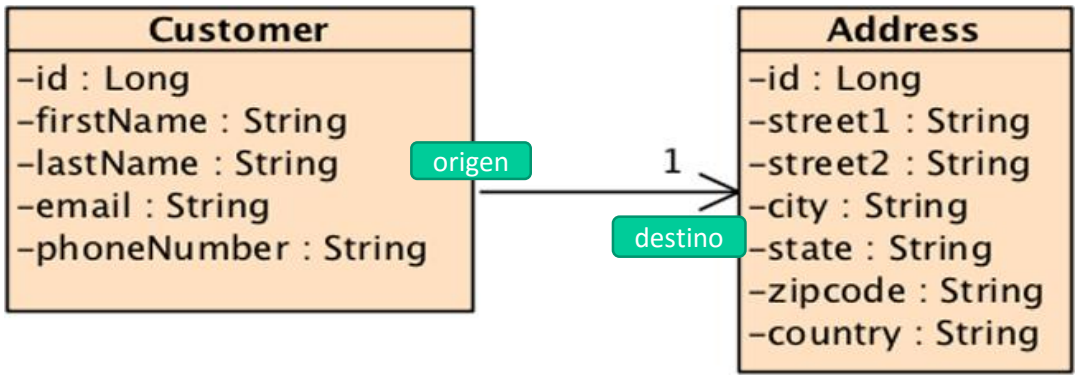
- Supongamos que un cliente tiene una dirección y solo una dirección y que cada dirección es única para un cliente.
- A nivel de base de datos..

```
1. create table ADDRESS (
2. ID BIGINT not null,
3. STREET1 VARCHAR(255),. . . . ,
4. primary key (ID)
5. );
```

```
1. create table CUSTOMER (
2. ID BIGINT not null,
3. FIRSTNAME VARCHAR(255), . . . . . .
4. ADDRESS_ID BIGINT,
5. primary key (ID),
6. foreign key (ADDRESS_ID) references ADDRESS(ID)
7. );
```

@OneToOne Unidirectional

- A nivel de código java, “Customer” tendrá un atributo de tipo “Address” pero “Address” no tendrá atributos de tipo “Customer”.




```
1.  @Entity
2.  public class Customer {
3.  @Id @GeneratedValue
4.  private Long id;
5.  private String firstName;
6.  . . . . . .
7.  private Address address;
```

```
1.  @Entity
2.  public class Address {
3.  @Id @GeneratedValue
4.  private Long id;
5.  private String street1;
```

@OneToOne Unidirectional

- Es obligatorio que todas las relaciones tengan asignado un mapeo.
- Para representar esta relación empleamos la anotación @OneToOne.
  - Si no especificamos nada mediante la anotación @JoinColumn, la clave foránea la tomará como [ENTIDAD]_[CLAVE].
  - Tambien podemos definir si puede ser nula o no la columna (cardinalidad [0..1] o estrictamente 1)

```
1. @Entity
2. public class Customer {
3. @Id @GeneratedValue
4. private Long id;
5. private String firstName;
6. . . . . .
7. @OneToOne
8. @JoinColumn(name = "add_fk", nullable = false)
9. private Address address;
10. }
```

@OneToMany Unidirectional

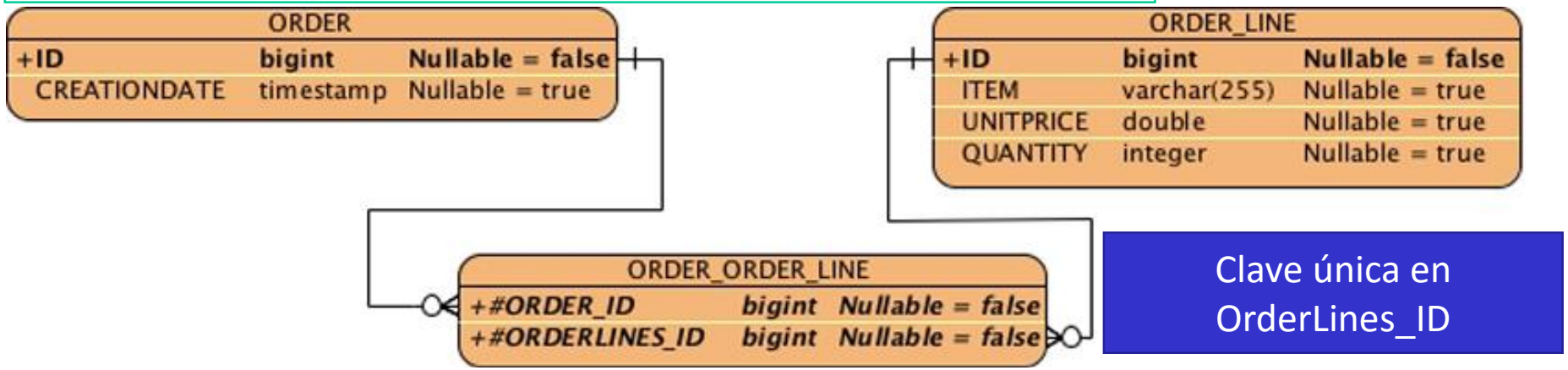
- En estas relaciones un objeto origen puede tener la referencia de más de un objeto destino relacionado.
- Por ejemplo una orden de compra, puede estar compuesta de varias líneas que detallan la compra.
- La anotación que se usa para esta situación es @OneToMany.
- 

@OneToMany Unidirectional

- Por defecto se usa una tabla de unión, con dos claves foráneas.

```
1.  @Entity
2.  public class Order {
3.  @Id @GeneratedValue
4.  private Long id;
5.  @OneToMany
6.  @JoinTable(name = "jnd_ord_line",
7.  joinColumns = @JoinColumn(name = "order_fk"),
8.  inverseJoinColumns = @JoinColumn(name = "order_line_fk") )
9.  private List<OrderLine> orderLines;
10. }
```

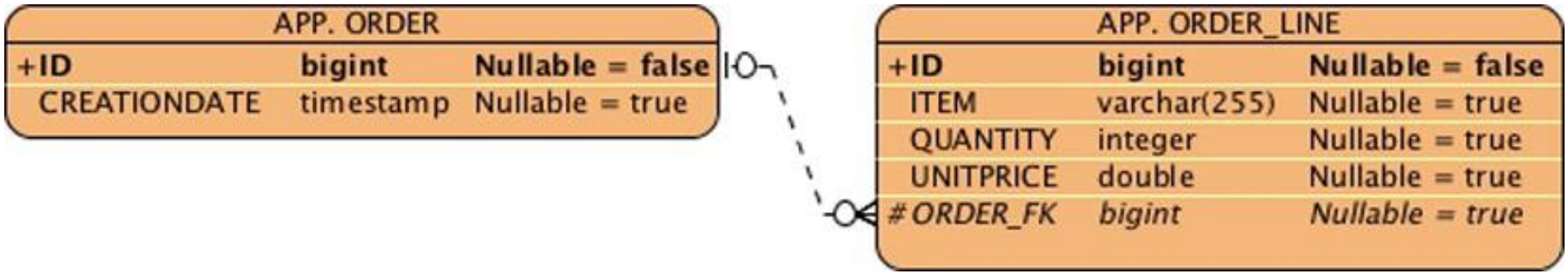
```
1.  @Entity
2.  @Table(name = "order_line")
3.  public class OrderLine {
4.  @Id @GeneratedValue
5.  private Long id;
6.  . . . .
7.  }
```



@OneToMany Unidirectional

- Si queremos evitar usar una tabla de unión solo usamos la anotación “@JoinColumn”

```
1.  @Entity
2.  public class Order {
3.  @Id @GeneratedValue
4.  private Long id;
5.  @Temporal(TemporalType.TIMESTAMP)
6.  private Date creationDate;
7.  @OneToMany(fetch = FetchType.EAGER)
8.  @JoinColumn(name = "order_fk")
9.  private List<OrderLine> orderLines;
10. // Constructors, getters, setters
11. }
```



@ManyToMany Bidirectional

- Una relación de muchos a muchos bidireccional existe cuando un objeto puede apuntar a muchos objetos de otro tipo relacionado, y a su vez cada uno de estos tipos de objetos relacionados, puede apuntar también a muchos objetos originales.
  - Ejemplos: Libro y Autores, Disco y Artistas, Alumnos y Cursos....
- En el modelo relacional la única forma de representar esto es mediante una tabla de unión donde cada entidad exporta su clave foránea.
- En el modelo de objetos cada entidad tendrá un arreglo o colección del tipo relacionado.
- En las relaciones “bidireccionales” se necesita definir en este caso, quien es la entidad responsable de actualizar la relación.
  - El OWNER, y se define con el elemento “mappedBy”

@ManyToMany Bidirectional

- En las relaciones bidireccionales, cada entidad tiene una relación (puntero) hacía la otra.
- Ej JPA deben seguir las siguientes reglas:
  - El lado inverso debe referirse al lado “owner”a través del elemento mappedBy de @OneToOne, @OneToMany, or @ManyToMany (no está presente en **@ManyToOne**)
  - En una relación ManyToOne, bidireccional, el lado muchos, siempre es el dueño de la relación (el dueño en el modelo relacional tiene la clave foránea).
  - En las relaciones “oneToOne” bidireccionales si la estructura de datos nos viene dada, el dueño es el que tiene la clave foránea, sino cualquiera puede serlo.
  - En las relaciones “manyToMany” cualquiera puede ser el dueño.

@ManyToMany Bidirectional

- mappedBy define cual es el nombre del atributo en el dueño de la relación.
  - La entidad que tiene una anotación con este atributo NO ES LA DUEÑA y la que tiene un atributo con el nombre denotado por mappedBy si es la dueña.

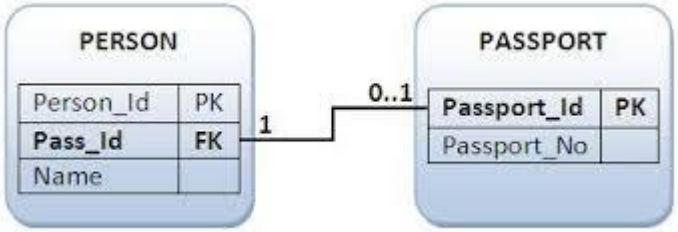
```
1.  @Entity
2.  public class Artist {
3.  @Id @GeneratedValue
4.  private Long id;
5.  private String firstName;
6.  @ManyToMany
7.  @JoinTable(name = "jnd_art_cd", 
8.  joinColumns = @JoinColumn(name = "artist_fk"),
9.  inverseJoinColumns = @JoinColumn(name = "cd_fk"))
10. private List<CD>appearsOnCDs;
11. }
```

DUEÑO

```
1.  @Entity
2.  public class CD {
3.  @Id @GeneratedValue
4.  private Long id;
5.  private String title;
6.  private Float price;
7.  private String description;
8.  @ManyToMany(mappedBy = "appearsOnCDs")
9.  private List<Artist> createdByArtists;
10. // Constructors, getters, setters
11. }
```

@OneToOne bidireccional

- Al igual que en las relaciones @ManyToMany, debemos elegir cual es el lado inverso y cual es el dueño de la relación, y puede ser cualquiera de los dos.
  - El lado que no tenga el atributo “mappedBy” será el owner.
  - A nivel de bases de datos relacional, el dueño de la relación tendrá la clave foránea en su tabla.



- Si no se explicita quien es el dueño JPA puede lo tomará como dos relaciones “oneToMany” unidireccionales y tendremos hasta 4 tablas: Persona y Pasaporte con las tablas de union (Persona_Pasaporte y Pasaporte_Persona)
- **Error en tiempo de despliegue!!!!**.

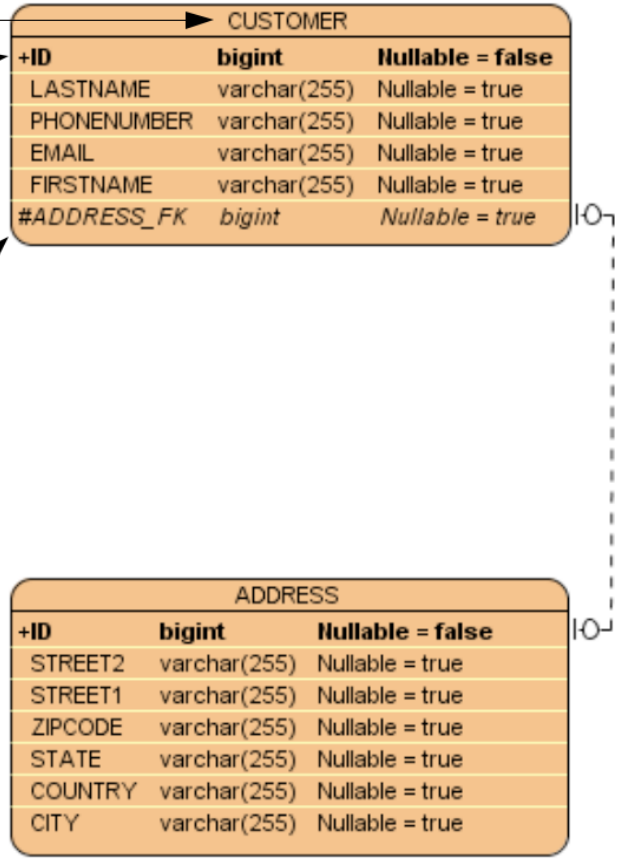
Ejemplo OneToOne Bidireccional

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@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;
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```



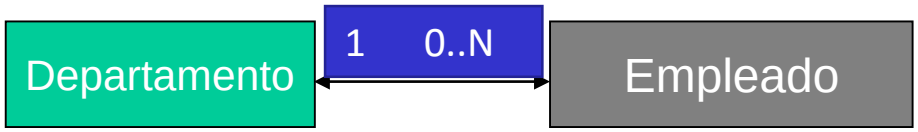
@OneToOne Unidireccional

- Relación **unidireccional** uno a muchos:


```
@Entity
public class Departamento implements Serializable{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    int id;
    @OneToMany
    List<Empleado> empleados = new ArrayList<Empleados>();
. . .
}
```

@OneToOne bidireccional

- Relación **bidireccional** uno a muchos:




```
@Entity
public class Departamento implements Serializable{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    int id;
    @OneToMany
    List<Empleado> empleados = new ArrayList<Empleados>();
. . .
}
```

```
@Entity
public class Empleado implements Serializable{
. . .
    @ManyToOne
    Departamento departamento;
. . .
}
```

Como se cargan las relaciones

- Cada tipo de relación tiene un tipo de carga por defecto.
- En general tendemos a pensar que Lazy es óptima, pero, muchas veces aumenta los acceso a la base de datos, por lo que en realidad, debemos evaluarlo por cada tipo de problema que se intente resolver.

Tipos de Carga por defecto

1. @OneToOne EAGER
2. @ManyToOne EAGER
3. @OneToMany LAZY
4. @ManyToMany LAZY

- Si cada vez que cargamos una orden, **siempre** vamos a acceder al detalle de la orden entonces podría ser apropiado

```
1. @OneToMany(fetch = FetchType.EAGER)
2. private List<OrderLine> orderLines;
```

Ordenamiento de relaciones

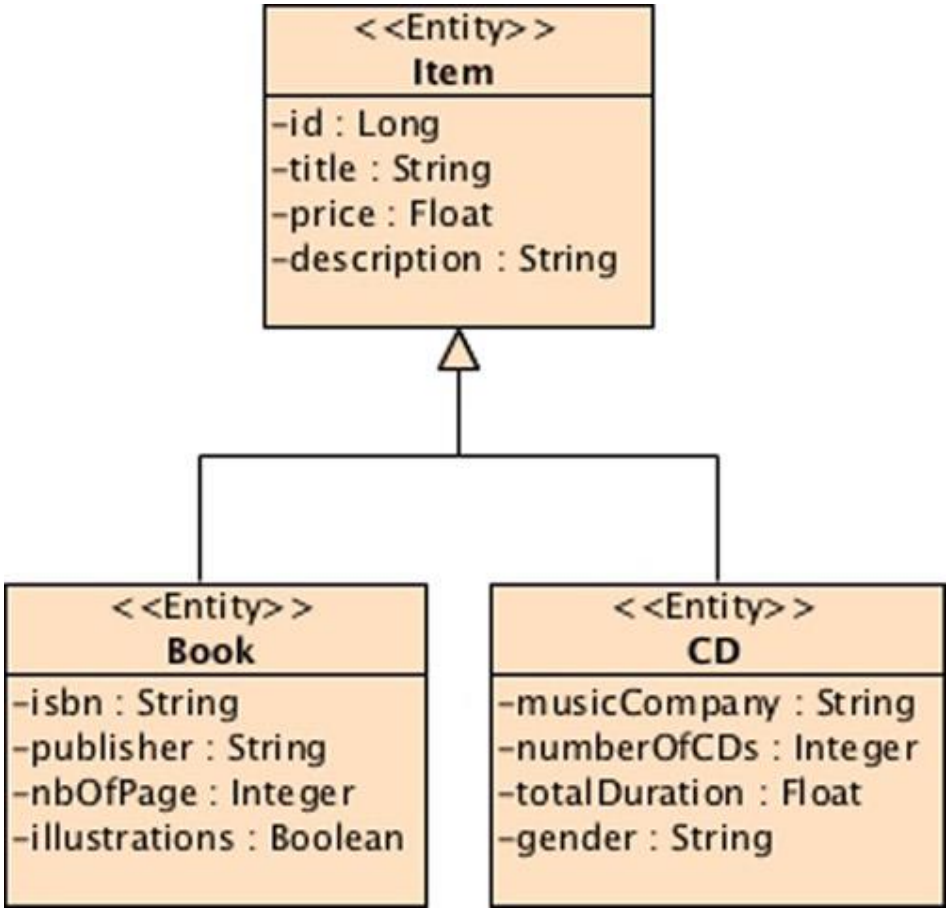
- En las relaciones OneToMany y ManyToMany, las entidades se almacenan en colecciones por lo que aunque por defecto no tienen un criterio de orden, podrían estar ordenadas.
- SI queremos que se cargue de la base de datos la lista ordenada, o bien ordenamos la colección o bien relizamos la consulta SQL con una cláusula de orden.
- Mediante la anotación @OrderBy se puede realizar un ordenamiento **dinámico**.
  - Se ordenan los elementos cuando se recupera la colección
  - Se indica que cláusula OrderBy debe usar en la consulta que recupera la

```
1.  @Entity
2.  public class Comment {
3.  @Id @GeneratedValue
4.  private Long id;
5.  private String nickname;
6.  private Integer note;
7.  @Column(name = "posted_date")
8.  @Temporal(TemporalType.TIMESTAMP)
9.  private Date postedDate;
10. }
```

```
1.  @Entity
2.  public class News {
3.  @Id @GeneratedValue
4.  private Long id;
5.  @Column(nullable = false)
6.  private String content;
7.  @OneToMany(fetch = FetchType.EAGER)
8.  @OrderBy("postedDate DESC")
9.  private List<Comment> comments;
10. }
```

Mapeo de relaciones de Herencia

- Para realizar el mapeo de relaciones de herencia existen 3 alternativas:
  - *A single-table-per-class hierarchy strategy*: The sum of the attributes of the entire entity hierarchy is flattened down to a single table (this is the default strategy).
  - *A joined-subclass strategy*: In this approach, each entity in the hierarchy, concrete or abstract, is mapped to its own dedicated table.
  - *A table-per-concrete-class strategy*: This strategy maps each concrete entity hierarchy to its own separate table (Support for the table-per-concrete-class inheritance mapping strategy is still optional in JPA 2.1. Portable applications should avoid using it until officially mandated.)




```
1.  @Entity
2.  public class Item {
3.  @Id @GeneratedValue
4.  protected Long id;
5.  protected String title;
6.  *******
7.  @Entity
8.  public class Book extends Item {
9.  private String isbn;

10. @Entity
11. public class CD extends Item {
12. private String musicCompany;
```

"Single-Table-per-Class Hierarchy"

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
GENRE	varchar(255)	Nullable = true

- Se agrega una columna discriminadora, por defecto de tipo String y el valor es el nombre de la entidad a la que refiere.
- Con la anotación @DiscriminatorColumn la podemos personalizar.
- El valor se puede sobreescribir con la anotación @DiscriminatorValue

ID	DTYPE	TITLE	PRICE	DESCRIPTION	MUSIC COMPANY	ISBN	...
1	Item	Pen	2.10	Beautiful black pen			...
2	CD	Soul Trane	23.50	Fantastic jazz album	Prestige		...
3	CD	Zoot Allures	18	One of the best of Zappa	Warner		...
4	Book	The robots of dawn	22.30	Robots everywhere		0-554-456	...
5	Book	H2G2	17.50	Funny IT book ;o)		1-278-983	...

"Single-Table-per-Class Hierarchy”

```
1. @Entity
2. @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
3. @DiscriminatorColumn (name="disc", discriminatorType =
   DiscriminatorType.CHAR)
4. @DiscriminatorValue("I")
5. public class Item {
6. @Id @GeneratedValue
7. protected Long id;
8. protected String title;
9. protected Float price;
10. protected String description;
11. // Constructors, getters, setters
12. }
```

"Single-Table-per-Class Hierarchy”

```
1. @Entity
2. @DiscriminatorValue("B")
3. public class Book extends Item {
4. private String isbn;
5. private String publisher;
6. private Integer nbOfPage;
7. private Boolean illustrations;
8. // Constructors, getters, setters
9. }
```

```
1. @Entity
2. @DiscriminatorValue("C")
3. public class CD extends Item {
4. private String musicCompany;
5. private Integer numberOfCDs;
6. private Float totalDuration;
7. private String genre;
8. // Constructors, getters, setters
9. }
```

ID	DISC	TITLE	PRICE	DESCRIPTION	MUSIC COMPANY	ISBN	...
1	I	Pen	2.10	Beautiful black pen			...
2	C	Soul Trane	23.50	Fantastic jazz album	Prestige		...
3	C	Zoot Allures	18	One of the best of Zappa	Warner		...
4	B	The robots of dawn	22.30	Robots everywhere		0-554-456	...
5	B	H2G2	17.50	Funny IT book ;o)		1-278-983	...

Joined-Subclass

- Cada entidad de la jerarquía está mapeada a su propia tabla.
- La entidad raíz tiene la clave primaria y la exporta al resto de las entidades.
  - El resto de las entidades tiene una tabla con sus propios atributos.
  - La tabla raíz puede tener una columna discriminador, no es neceasaria en las tablas hijas.
- Se implementa con la antoación @Inheritance=InheritanceType.JOINED.

```
1. @Entity
2. @Inheritance(strategy = InheritanceType.JOINED)
3. public class Item {
4. @Id @GeneratedValue
5. protected Long id;
```

Tareas

- Mapear los tipos de datos
- Mapear las relaciones
- Persistir