UNIVERSITÀ DELLA CALABRIA DIPARTIMENTO DI **MATEMATICA** **E INFORMATICA**

# AGILE SOFTWARE DEVELOPMENT

**Dr. Francesco Scalzo** - francesco.scalzo@unical.it
Enterprise Solution Engineer at **Revelis s.r.l.**

Office hours & info: send a mail
**Code:** https://github.com/mascalzonef/asd_21-22.git

## ORM - JPA

- **Object-Relational Mapping** is the process of persisting Java object directly to a database table.
- The name of the class becomes the name of the table, and each field becomes a column.
- Each row corresponds to a record in the application.

- **The Java Persistence API, or JPA**, is a specification that defines the management of relational data in a Java application. The API maps out a set of concepts that defines which objects within the application should be persisted, and how it should persist them.
- It's important to note here that **JPA is only a specification** and that it needs an implementation to work

- **Hibernate** is an object-relational mapping tool that provides an implementation of JPA. **Hibernate is one of the most mature JPA implementations** around, with a huge community backing the project.

## ORM - JPA

### Some Concepts

- **POJO**: or Plain Old Java Object, is a normal Java object class (that is, not a JavaBean, EntityBean etc.) and does not serve any other special role nor does it implement any special interfaces of any of the Java framework. It is used for any simple object with no extra stuff.

- **DTO**: Data Transfer Object it is basically POJO that is used to transfer data between several layers. It transfers data between classes and modules of the application

- **Entity**: defines which objects should be persisted to the database. For each persisted entity, JPA creates a new table within the chosen database.

- **DAO**: provides an abstract interface to data operations without exposing database details. Provides CRUD operations (Create, Read, Update, Delete)

## JPA Concepts

- **Entities in JPA** are POJOs that can be persisted to the database. An entity represents a table stored in a database. Every instance of an entity represents a row in the table.
- Each JPA entity must have a primary key which uniquely identifies it. The @Id annotation defines the primary key. We can generate the identifiers in different ways which are specified by the @GeneratedValue annotation.
- The @Column annotation has many elements such as name, length, nullable, and unique.
- @Transient: If there's a field that we don't want to persist to the database, we can declare the field transient with the @Transient annotation.
- Sometimes, we may want to persist a Java enum type. We can use the @Enumerated annotation to specify whether the enum should be persisted by name or by ordinal (default).
- Relationships: JPA specifies how we should manage relationships among different database tables through**: @OneToOne** - **@OneToMany/@ManyToOne** - **@ManyToMany**

## JPA

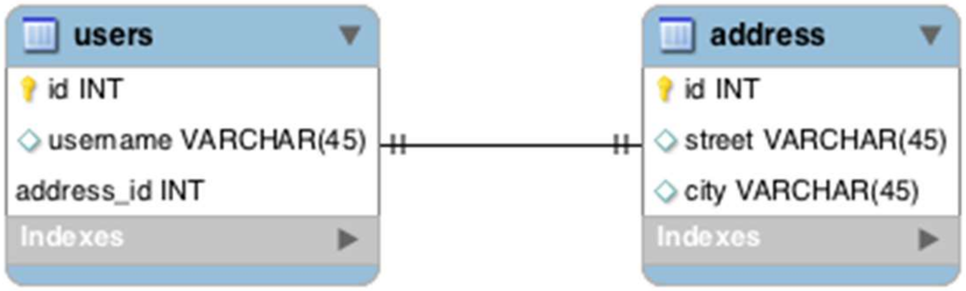
### Relationships

- Relationships: JPA specifies how we should manage relationships among different database tables through:
  1. @OneToOne
  2. @OneToMany
  3. @ManyToOne
  4. @ManyToMany

- The @JoinColumn annotation defines the physical mapping on the owner side.
  - When using a @OneToMany mapping we can use the mappedBy parameter to indicate that the given column is owned by another entity
  - In a One-to-Many/Many-to-One relationship, the owning side is usually defined on the ‘many' side of the relationship. It's usually the side which owns the foreign key.

# JPA

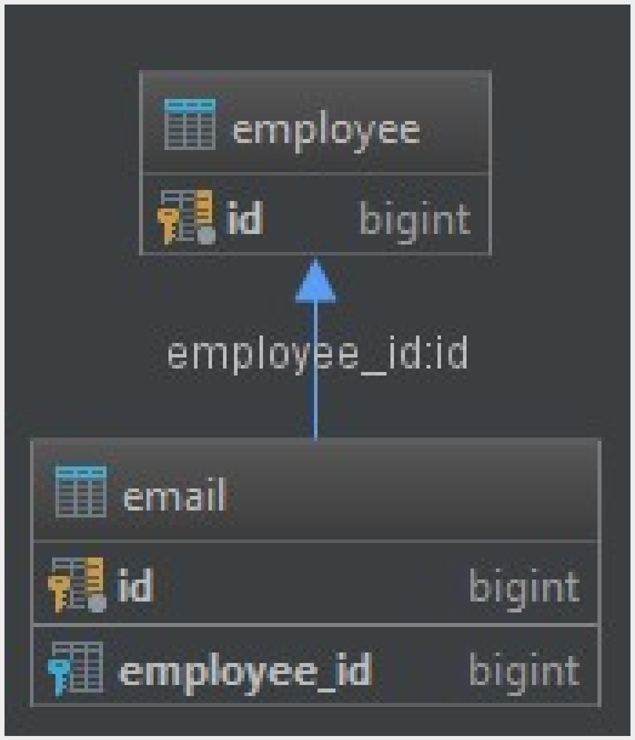
## Relations: OneToOne



In this example, the *address_id* column in *users* is the foreign key to *address*.

# JPA

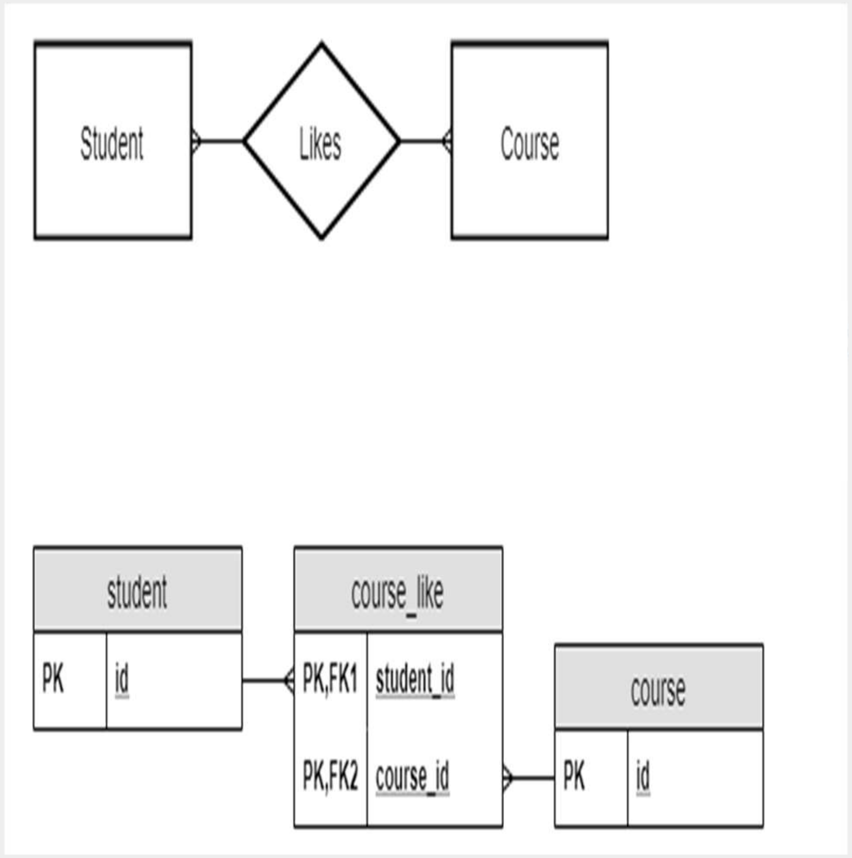
## Relations: OneToMany / ManyToOne

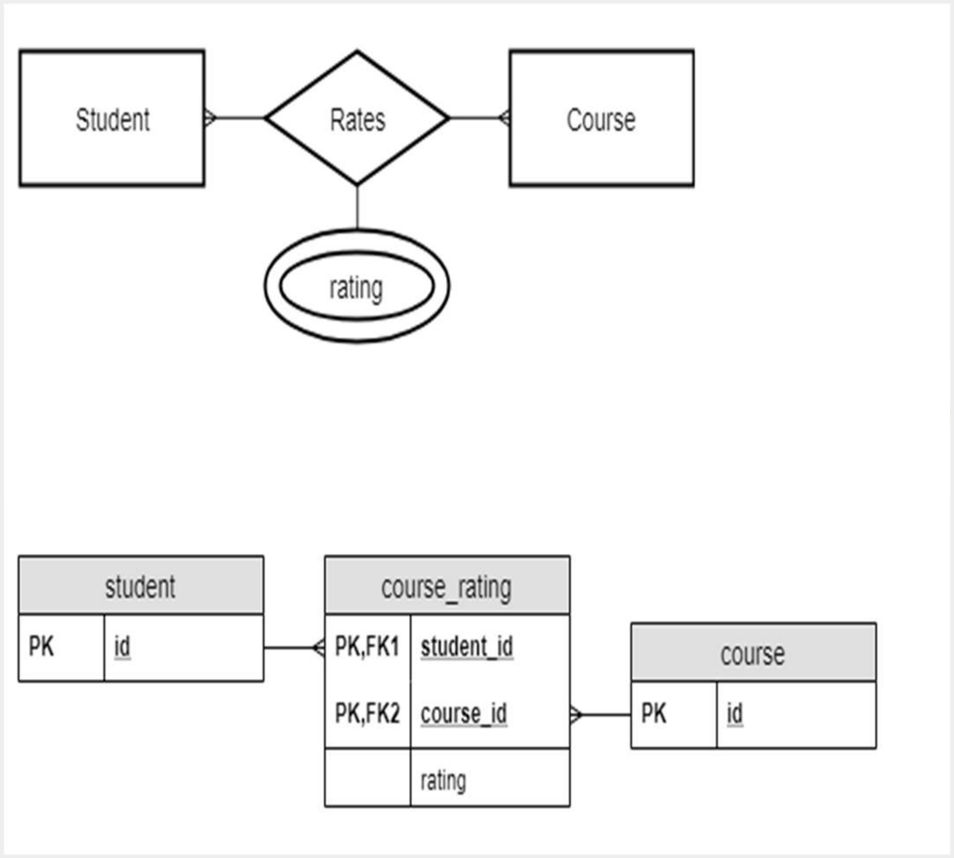


- Once we have defined the owning side of the relationship, Hibernate already has all the information it needs to map that relationship in our database. To make this association bidirectional, all we'll have to do is to define the referencing side. The inverse or the referencing side simply maps to the owning side. We can easily use the **mappedBy** attribute of **@OneToMany** annotation to do so.

# JPA

## Relations: ManyToMany





## JPA

### CascadeType

To establish a dependency between related entities, JPA provides javax.persistence.CascadeType.
When we perform some action on the target entity, the same action will be applied to the associated entity. (By default no operations are cascaded.)

1. CascadeType.PERSIST: means that save() or persist() operations cascade to related entities.
2. CascadeType.MERGE: means that related entities are merged when the owning entity is merged.
3. CascadeType.REFRESH: does the same thing for the refresh() operation.
4. CascadeType.REMOVE: removes all related entities association with this setting when the owning entity is deleted.
5. CascadeType.DETACH: detaches all related entities if a “manual detach” occurs.
6. CascadeType.ALL: is a shortcut for all of the above cascade operations.

The orphanRemoval attribute is going to instruct JPA to remove the child entity no longer referenced by its parent entity.

## JPA

### Embeddable/Embedded

JPA provides the @Embeddable annotation to declare that a class will be embedded by other entities.

The JPA annotation @Embedded is used to embed a type into another entity.

We can use @AttributeOverrides and @AttibuteOverride to override the column properties of our embedded type.

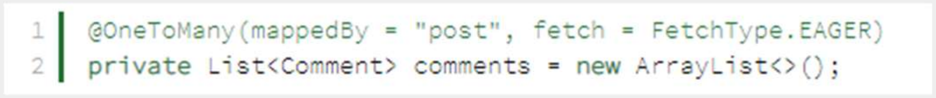
## JPA

### FetchType

The FetchType method defines two strategies for fetching data from the database:

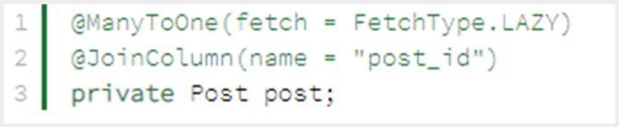
1. FetchType.EAGER: The persistence provider must load the related annotated field or property. This is the default behavior for @Basic, @ManyToOne, and @OneToOne annotated fields.
2. FetchType.LAZY: The persistence provider should load data when it's first accessed, but can be loaded eagerly. This is the default behavior for @OneToMany, @ManyToMany and @ElementCollection-annotated fields.

For example, when we load a *Post* entity, the related *Comment* entities are not loaded as the default *FetchType* since *@OneToMany* is *LAZY.* We can override this behavior by changing the *FetchType* to *EAGER:*



```
@OneToMany(mappedBy = "post", fetch = FetchType.EAGER)
private List<Comment> comments = new ArrayList<>();
```

*By comparison, when we load a Comment entity, his Post parent entity is loaded as the default mode for @ManyToOne, which is EAGER. We can also choose to not load the Post entity by changing this annotation to LAZY:*



```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "post_id")
private Post post;
```

# JPA

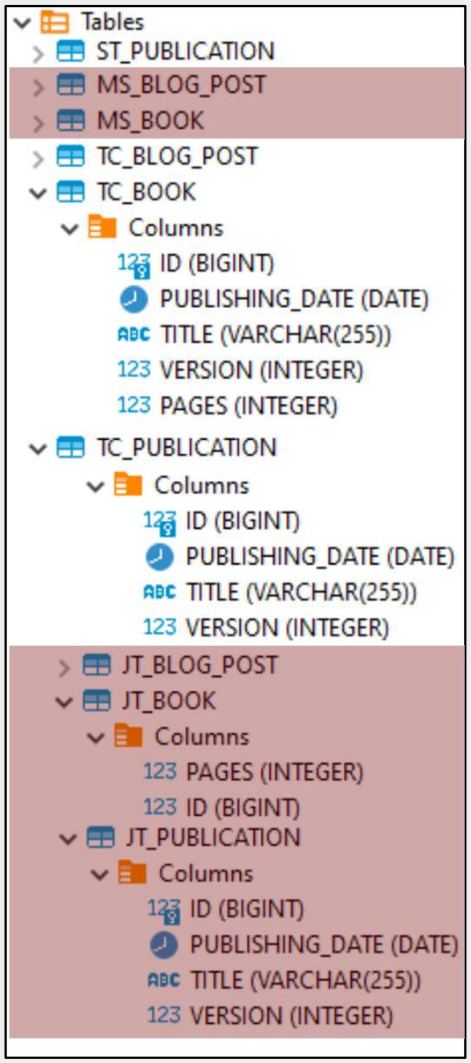
## Inheritance

- **Single Table** – It maps all entities of the inheritance structure in a single database table. This approach makes polymorphic queries very efficient and provides the best performance. To differentiate the tables, we use @DiscriminatorColumn and @DiscriminatorValue.

- **Mapped Superclass** – It maps only concrete class to its own table. The parent classes can't be entities: inheritance is only evident in the class, but not the entity model.

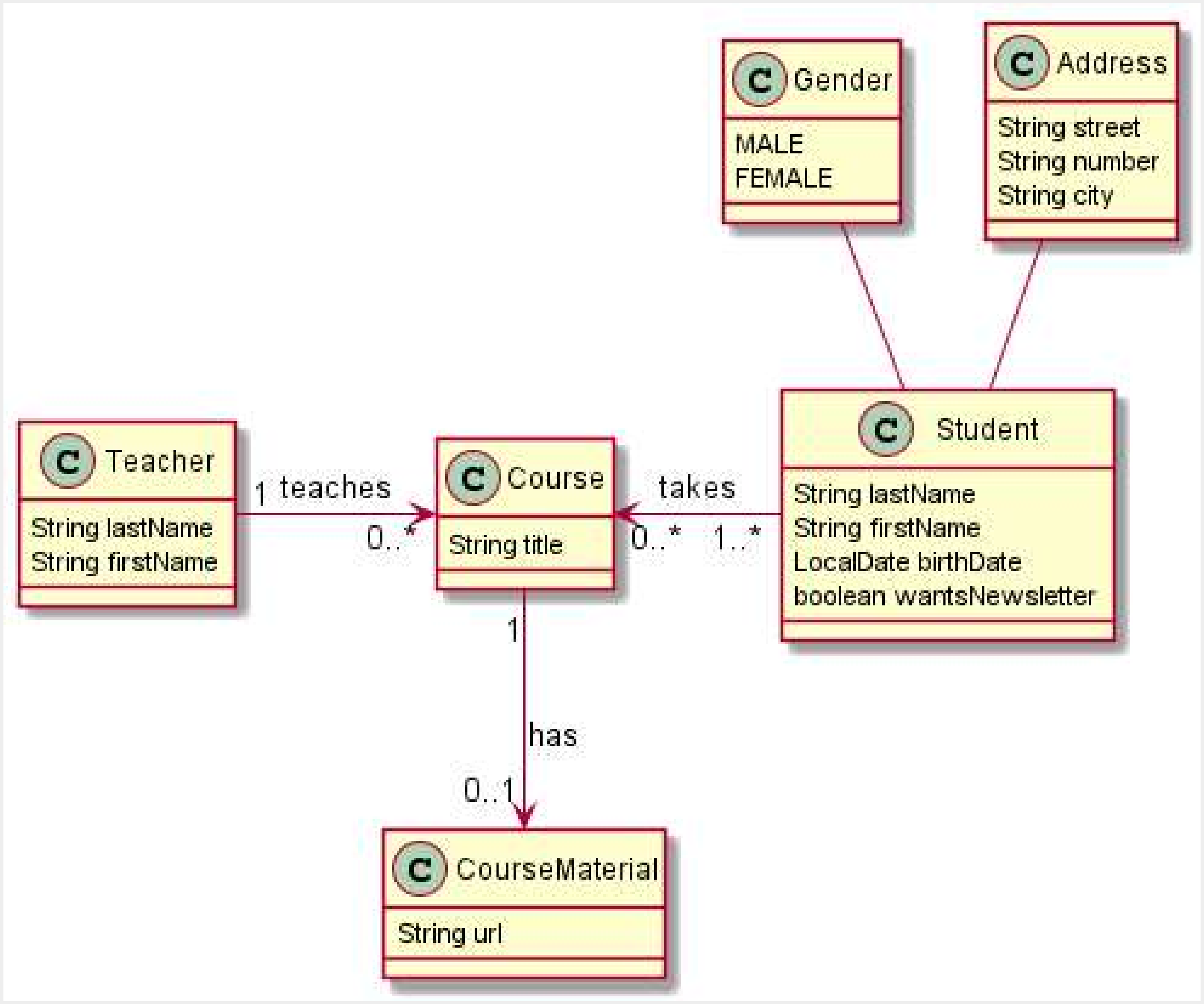
- **Table Per Class** – It maps each class to its own table. All the properties of a class, are in its table, so no join is required: the Table Per Class strategy maps each entity to its table which contains all the properties of the entity, including the ones inherited.

- **Joined Table** – It maps each class to its own table. each class has its table and querying a subclass entity requires joining the tables: the only column which repeatedly appears in all the tables is the identifier, which will be used for joining them when needed.

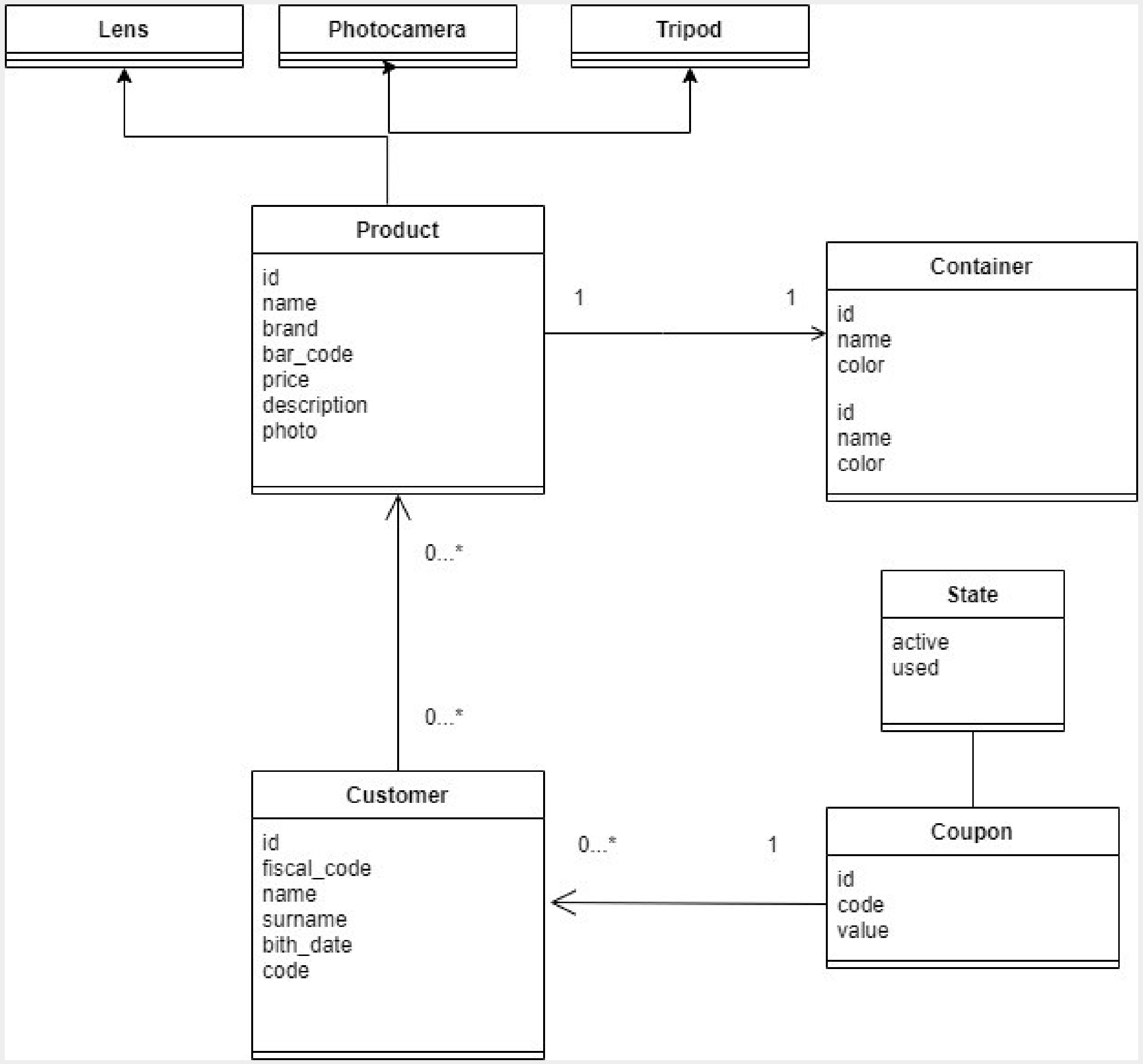


```
Tables
  ST_PUBLICATION
  MS_BLOG_POST
  MS_BOOK
  TC_BLOG_POST
  TC_BOOK
    Columns
      ID (BIGINT)
      PUBLISHING_DATE (DATE)
      TITLE (VARCHAR(255))
      VERSION (INTEGER)
      PAGES (INTEGER)
  TC_PUBLICATION
    Columns
      ID (BIGINT)
      PUBLISHING_DATE (DATE)
      TITLE (VARCHAR(255))
      VERSION (INTEGER)
  JT_BLOG_POST
  JT_BOOK
    Columns
      PAGES (INTEGER)
      ID (BIGINT)
  JT_PUBLICATION
    Columns
      ID (BIGINT)
      PUBLISHING_DATE (DATE)
      TITLE (VARCHAR(255))
      VERSION (INTEGER)
```

## Exercise


Gender
MALE
FEMALE
Address
String street
String number
String city
Teacher
String lastName
String firstName
1 teaches
0..*
Course
String title
takes
0..*
1..*
Student
String lastName
String firstName
LocalDate birthDate
boolean wantsNewsletter
1
has
0..1
CourseMaterial
String url

# Exercise



Lens
Photocamera
Tripod
Product
id
name
brand
bar_code
price
description
photo
Container
id
name
color
id
name
color
1
1
0...*
0...*
State
active
used
Customer
id
fiscal_code
name
surname
bith_date
code
Coupon
id
code
value
0...*
1