JPA RELATIONAL MAPPING

[Mapping Directions](#_l3uz19d98p3u)

[Types of Mapping](#_qbbspekvzte8)

[ONE-TO-ONE:](#_db60e8regr16)

[ONE-TO-MANY:](#_vv1mzyp14gau)

[MANY-TO-ONE:](#_tme1yzrx0k0n)

[MANY-TO-MANY:](#_u5f3lm2xm33h)

[SOURCE CODE:](#_5lt6dcq4abfw)

# Mapping Directions

Mapping Directions are divided into two parts: -

* **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.
* **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

# Types of Mapping

Following are the various ORM mappings: -

* **One-to-one -** This association is represented by @OneToOne annotation. Here, instance of each entity is related to a single instance of another entity.
* **One-to-many -** This association is represented by @OneToMany annotation. In this relationship, an instance of one entity can be related to more than one instance of another entity.
* **Many-to-one -** This mapping is defined by @ManyToOne annotation. In this relationship, multiple instances of an entity can be related to single instance of another entity.
* **Many-to-many -** This association is represented by @ManyToMany annotation. Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side.

## ONE-TO-ONE:

Employee.java:

@Entity

@Table(name="employee")

public class Employee {

@Id

@GeneratedValue(strategy = javax.persistence.GenerationType.IDENTITY )

private int id;

@Column(name="firstname")

private String firstName;

@OneToOne(cascade=CascadeType.REMOVE)

@JoinColumn(name="address\_id")

private Address address;

}

Address.java :

@Entity

@Table(name="address")

public class Address {

@Id

@GeneratedValue(strategy = javax.persistence.GenerationType.IDENTITY )

private int id;

@Column(name="street")

private String street;

@Column(name="city")

private String city;

@OneToOne(cascade=CascadeType.ALL,mappedBy="address")

private Employee employee;

}

## ONE-TO-MANY:

@Entity

@Table(name="department")

public class Department {

@Id

@GeneratedValue(strategy = javax.persistence.GenerationType.IDENTITY )

private int id;

@Column(name="name")

private String name;

@OneToMany(mappedBy="department",fetch=FetchType.EAGER,cascade=CascadeType.REMOVE)

private List<Employee> employees;

}

## MANY-TO-ONE:

@Entity

@Table(name="employee")

public class Employee {

@Id

@GeneratedValue(strategy = javax.persistence.GenerationType.IDENTITY )

private int id;

@Column(name="firstname")

private String firstName;

@ManyToOne

@JoinColumn(name="department\_id")

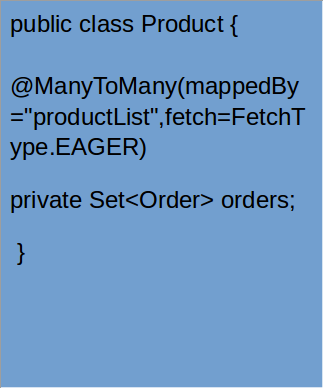
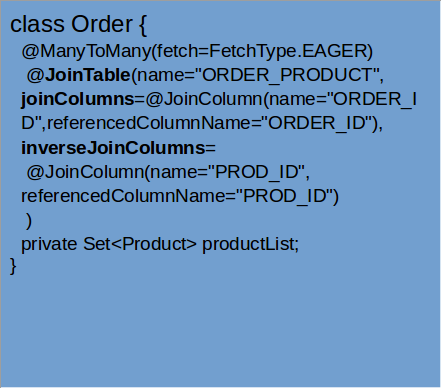
private Department department;

}

## MANY-TO-MANY:

This has two types:

* Many to Many with single primary Key



* Many to Many with composite primary Key

Consider a situation when employee\_id from employee table and speciality\_id from speciality table are together mapped into employee\_speciality table.

Here both employee\_id and speciality\_id are composite primary key for table employee\_speciality.

In Employee class:

Create a list of type EmployeeSpeciality Mapper class and map it to employeeId.

@JsonIgnore

@OneToMany(mappedBy="employeeId",cascade=CascadeType.REMOVE)

private List<EmployeeSpeciality> specialityList;

In EmployeeSpeciality Mapper class define two ids and map them to columns in table.

Two ways of defining composite key in jpa:

@IdClass

[@EmbeddedId](https://www.objectdb.com/api/java/jpa/EmbeddedId)

[@Embeddable](https://www.objectdb.com/api/java/jpa/Embeddable)

Using @IdClass:

@Entity

@Table(name="employee\_specialty")

@IdClass(EmployeeSplKey.class)

public class EmployeeSpeciality {

@Id

@Column(name="employee\_id")

private int employeeId;

@Id

@Column(name="specialty\_id")

private int specialityId;

}

EmployeeSplKey.java

public class EmployeeSplKey implements Serializable {

private static final long serialVersionUID = 6843977865924912122L;

private int employeeId;

private int specialityId;

}

If the inverse side needs access, then mapping can be mentioned using annotation in speciality class also.

EXERCISE:

Implement application with JPA relational mappings :

create table address(id int auto\_increment primary key, street varchar(50), city varchar(50), state varchar(30), country varchar(30), zipcode varchar(10));

create table department(id int auto\_increment primary key, name varchar(50));

create table specialty(id int auto\_increment primary key, name varchar(30));

create table employee(id int auto\_increment primary key, firstname varchar(40), lastname varchar(40), email varchar(50) unique, phone\_number bigint, salary bigint, department\_id int, address\_id int, foreign key(department\_id) references department(id),

foreign key(address\_id) references address(id));

create table employee\_specialty(employee\_id int, specialty\_id int, primary key(employee\_id, specialty\_id), foreign key(employee\_id) references employee(id), foreign key(specialty\_id) references specialty(id));

* One employee - one address (One to One)
* Employee - specialty many-to-many
* Many employees - 1 department (Many to one)
* One department - many employees

Use Spring boot annotations and JPA ( with relational mappings ) to implement the above schema.

### SOURCE CODE:

<https://github.com/sruthiviswanathan/Zterns-19-Sruthi/tree/master/EmployeeRelationalMapping>