# Essential object-relational mapping in JPA

The Java Persistence API is the standard framework for persistence in both Java SE and Java EE platforms. It is the outcome of the collaborative work of the industry's leading vendors in object-relational mapping, EJB and JDO, including a remarkable contribution from the open source community.

In this post we summarize, one-by-one, the essential relationships between entities. We examine each relationship from the prespectives of "Domain Model" and "Relation Model", and provide the associated construction in JPA. Finally, for each relationship we present an example from the real world.

The relationships are divided in two categories. "For daily use" presents the correct way to use common relationships in our application development activities. On the other hand, "Containing pitfalls" identify common mistakes.

For daily use:

1. One-to-One (one direction)
2. One-to-One (both directions)
3. Many-to-One (one direction)
4. Many-to-One (both directions)
5. One-to-Many (one direction)
6. One-to-Many (both directions)
7. Many-to-Many (both directions)

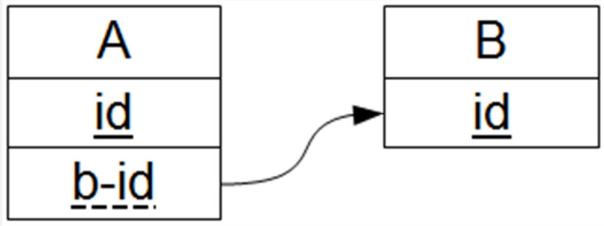
Containing pitfalls:

1. Many-to-Many (one direction)

# One-to-One (one direction)


Domain model of an one-to-one unidirectional relationship

```
@Entity
class A {
   @Id int id;

   @OneToOne
   B b;
}

@Entity
class B {
   @Id int id;
}
```




Relation model of an one-to-one unidirectional relationship

## Example 1, One-to-One unidirectional relationship in JPA

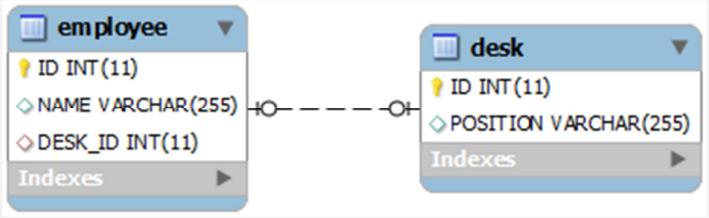
An employee has one desk.

```
@Entity
public class Employee implements Serializable {

   @Id
   private int id;
   private String name;

   @OneToOne
   private Desk desk;
```

```
}

@Entity
public class Desk implements Serializable {

   @Id
   private int id;

   private String position;

}
```

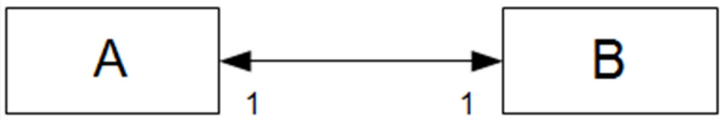
```
mysql> describe employee;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| ID      | int(11)      | NO   | PRI | NULL    |       |
| NAME    | varchar(255) | YES  |     | NULL    |       |
| DESK_ID | int(11)      | YES  | MUL | NULL    |       |
+---------+--------------+------+-----+---------+-------+

mysql> describe desk;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ID       | int(11)      | NO   | PRI | NULL    |       |
| POSITION | varchar(255) | YES  |     | NULL    |       |
+----------+--------------+------+-----+---------+-------+
```

Physical description of an one-to-one unidirectional relationship




ER diagram of an one-to-one unidirectional relationship

# One-to-One (both directions)

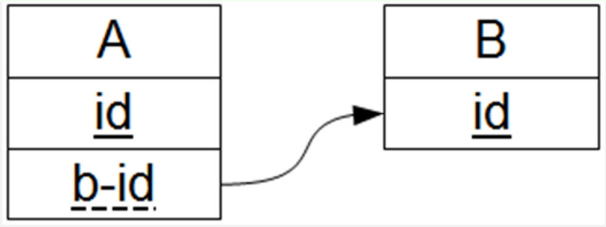



Domain model of an one-to-one birectional relationship

In this case we would like to keep a reference between both sides.

```
@Entity
class A {
   @Id int id;
   @OneToOne
```

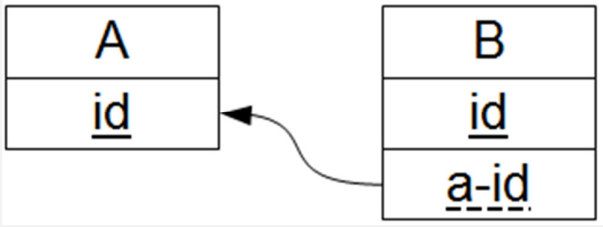
```
    B b;
}

@Entity
class B {
    @Id int id;

    @OneToOne(mappedBy="b")
    A a;
}
```




Relation model of an one-to-one bidirectional relationship

Since the relation concerns both directions, we may change the owning side.

```
@Entity
class A {
    @Id int id;

    @OneToOne(mappedBy="a")
    B b;
}

@Entity
class B {
    @Id int id;

    @OneToOne
    A a;
}
```



Relation model of an one-to-one bidirectional relationship

## Example 2, One-to-One bidirectional relationship in JPA

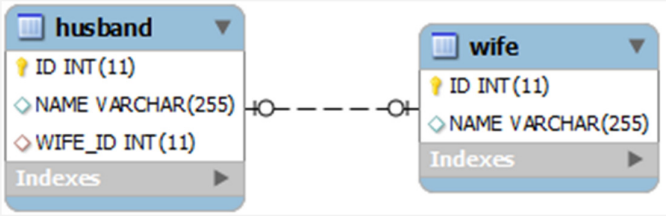
A husband has exactly one wife. A wife has exactly one husband.

```
@Entity
public class Husband implements Serializable {

   @Id
   private int id;

   private String name;

   @OneToOne
   private Wife wife;

}

@Entity
public class Wife implements Serializable {

   @Id
   private int id;

   private String name;

   @OneToOne(mappedBy="wife")
   private Husband husband;

}
```

We assume husband is the owning side, so he will hold the foreign key.

```
mysql> describe husband;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| ID      | int(11)      | NO   | PRI | NULL    |       |
| NAME    | varchar(255) | YES  |     | NULL    |       |
| WIFE_ID | int(11)      | YES  | MUL | NULL    |       |
+---------+--------------+------+-----+---------+-------+

mysql> describe wife;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| ID    | int(11)      | NO   | PRI | NULL    |       |
| NAME  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
```

Physical description of an one-to-one bidirectional relationship



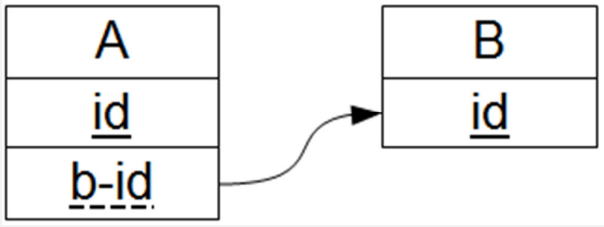

ER diagram of a bidirection one-to-one relationship

Now we can ask the wife "Who is your husband?", as well as, ask the husband "Who is your wife?".

# Many-to-One (one direction)


Domain model of a many-to-one unidirectional relationship

```
@Entity
class A {
   @Id int id;
   @ManyToOne
   B b;
}

@Entity
class B {
   @Id int id;
}
```



Relation model of a many-to-one unidirectional relationship

## Example 3, Many-to-One unidirectional relationship in JPA

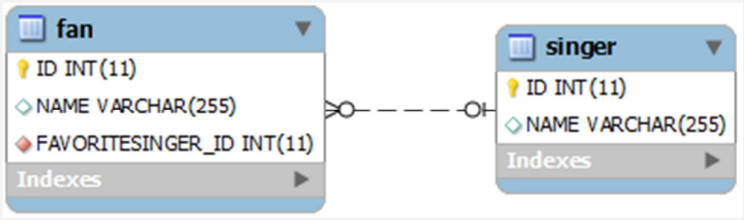
There are millions of music fans out there. Each fan has his favorite singer. Of course, a signer is not able to keep detailed records of his fans.

```
@Entity
public class Fan implements Serializable {

   @Id
   private int id;

   private String name;

   @ManyToOne
   private Singer favoriteSinger;

}

@Entity
public class Singer implements Serializable {

   @Id
   private int id;

   private String name;

}
```

As a result, the foreign key goes to the fan.

```
mysql> describe fan;
+-------------------+--------------+------+-----+---------+-------+
| Field             | Type         | Null | Key | Default | Extra |
+-------------------+--------------+------+-----+---------+-------+
| ID                | int(11)      | NO   | PRI | NULL    |       |
| NAME              | varchar(255) | YES  |     | NULL    |       |
| FAVORITESINGER_ID | int(11)      | YES  | MUL | NULL    |       |
+-------------------+--------------+------+-----+---------+-------+

mysql> describe singer;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| ID    | int(11)      | NO   | PRI | NULL    |       |
| NAME  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
```

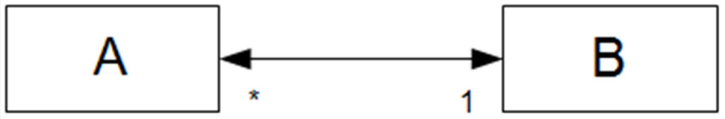
Physical description of a many-to-one unidirectional relationship




ER diagram of a many-to-one unidirectional relationship

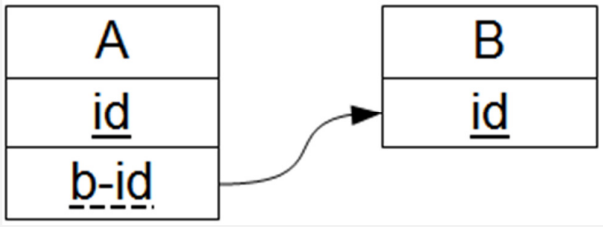
Now we may ask any fan “Who is your favorite signer?”.

# Many-to-One (both directions)




Domain model of a many-to-one bidirectional relationship

```
@Entity
class A {
   @Id int id;

   @ManyToOne
   B b;
}

@Entity
class B {
   @Id int id;

   @OneToMany(mappedBy="b")
   Collection<A> listOfA;
}
```



Relation model of a many-to-one bidirectional relationship

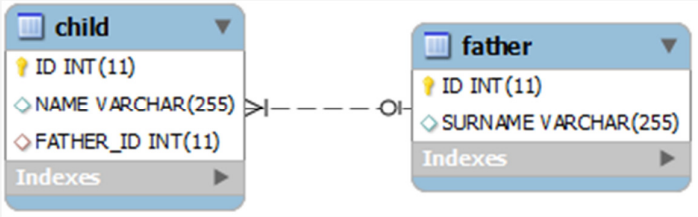
## Example 4, Many-to-One bidirectional relationship in JPA

The children of a father: A child knows his father and the father knows his children.

```
@Entity
public class Child implements Serializable {

   @Id
   private int id;

   private String name;

   @ManyToOne
   private Father father;

}

@Entity
public class Father implements Serializable {

   @Id
   private int id;
   private String surname;

   @OneToMany(mappedBy="father")
   private Collection<Child> children;

}
```

```
mysql> describe child;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| ID        | int(11)      | NO   | PRI | NULL    |       |
| NAME      | varchar(255) | YES  |     | NULL    |       |
| FATHER_ID | int(11)      | YES  | MUL | NULL    |       |
+-----------+--------------+------+-----+---------+-------+

mysql> describe father;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| ID      | int(11)      | NO   | PRI | NULL    |       |
| SURNAME | varchar(255) | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
```

Physical description of a many-to-one bidirectional relationship



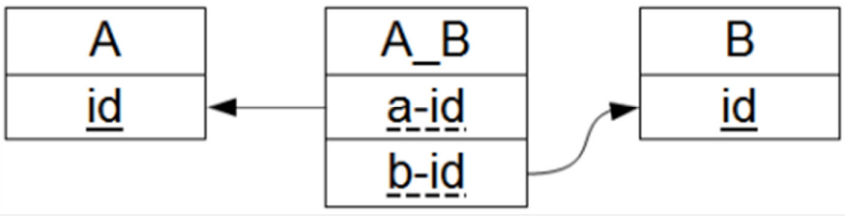

ER diagram of a many-to-one bidirectional relationship

# One-to-Many (one direction)


Domain model of a one-to-many unidirectional relationship

It is the case when an entity has a set of characteristics.

```
@Entity
class A {
   @Id int id;

   // A join table is assumed.
   @OneToMany
   Collection<B> listOfB;
}

@Entity
class B {
   @Id int id;
}
```



Relation model of a one-to-many unidirectional relationship

Thus, when declaring an OneToMany annotation without a mappedBy element, then a join table is assumed.

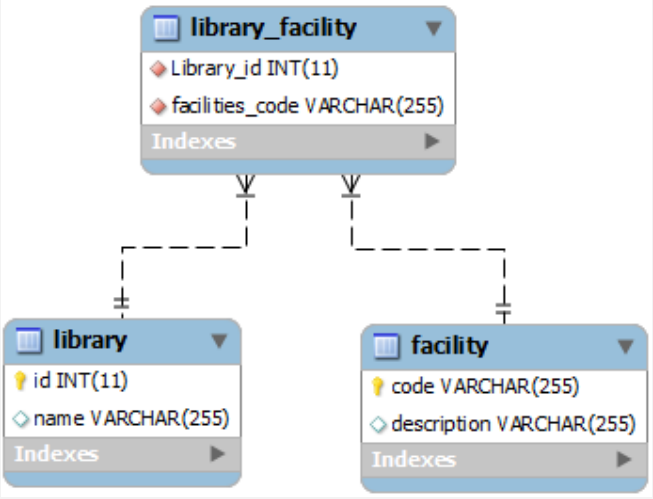
## Example 5, One-to-Many unidirectional relationship in JPA

A library which provides facilities.

```
@Entity
public class Library implements Serializable {

   @Id
   private int id;

   private String name;

   @OneToMany
   private Collection<Facility> facilities;

}

@Entity
public class Facility implements Serializable {

   @Id
   private String code;

   private String description;

}
```

```
mysql> describe library;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int(11)      | NO   | PRI | NULL    |       |
| name  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+

mysql> describe library_facility;
+-----------------+--------------+------+-----+---------+-------+
| Field           | Type         | Null | Key | Default | Extra |
+-----------------+--------------+------+-----+---------+-------+
| Library_id      | int(11)      | NO   | MUL | NULL    |       |
| facilities_code | varchar(255) | NO   | PRI | NULL    |       |
+-----------------+--------------+------+-----+---------+-------+

mysql> describe facility;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| code        | varchar(255) | NO   | PRI | NULL    |       |
| description | varchar(255) | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

Physical description of a one-to-many unidirectional relationship




ER diagram of an one-to-many unidirectional relationship

Now we can ask a library “What facilities do you provide?”. The answer could be “Scanning, Printing and Photocopying”. This set of facilities may be provided by one or more libraries at the same time.

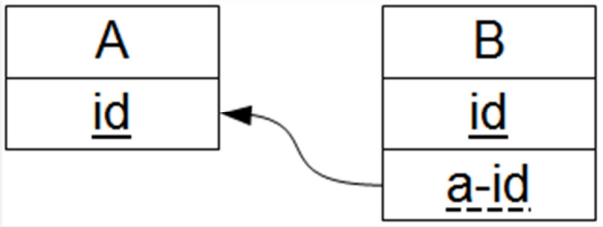
If using a join table is not what you want, then you should use the next relationship.

# One-to-Many (both directions)

It is exactly the same with Many-to-One (both directions), looking via a mirror.


Domain model of a one-to-many bidirectional relationship

```
@Entity
class A {
   @Id int id;

   @OneToMany(mappedBy="a")
   Collection<B> listOfB;
}

@Entity
class B {
   @Id int id;

   @ManyToOne
   A a;
}
```




Relation model of an one-to-many bidirectional relationship

## Example 6, One-to-Many bidirectional relationship in JPA

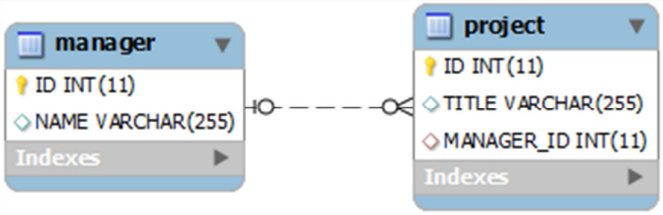
A manager who manages projects.

```
@Entity
public class Manager implements Serializable {

   @Id
   private int id;

   private String name;

   @OneToMany(mappedBy="manager")
```

```
    private Collection<Project> projects;

}

@Entity
public class Project implements Serializable {

    @Id
    private int id;

    private String title;

    @ManyToOne
    private Manager manager;

}
```

```
mysql> describe manager;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| ID    | int(11)      | NO   | PRI | NULL    |       |
| NAME  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+

mysql> describe project;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| ID         | int(11)      | NO   | PRI | NULL    |       |
| TITLE      | varchar(255) | YES  |     | NULL    |       |
| MANAGER_ID | int(11)      | YES  | MUL | NULL    |       |
+------------+--------------+------+-----+---------+-------+
```

Physical description of a one-to-many bidirectional relationship




ER diagram of an one-to-many bidirectional relationship

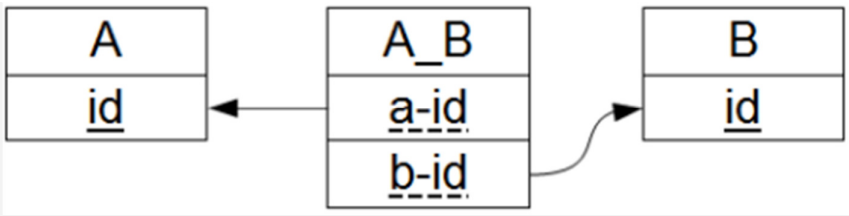
# Many-to-Many (both directions)


Domain model of a many-to-many bidirectional relationship

```
@Entity
```

```
class A {
   @Id int id;

   @ManyToMany
   Collection<B> listOfB;
}

@Entity
class B {
   @Id
   int id;

   @ManyToMany(mappedBy="listOfB")
   Collection listOfA; }
```

Of course, a join table is used.




Relation model of a many-to-many bidirectional relationship

Note: In a birectional many-to-many relationship we should *always*include the mappedBy element to either of the sides. This ensures that exactly one join table is used.

## Example 7, Many-to-Many bidirectional relationship in JPA

An engineer works in many projects. A project occupies many engineers. This is a classic many-to-many relationship. Moreover it expands to both directions, as:

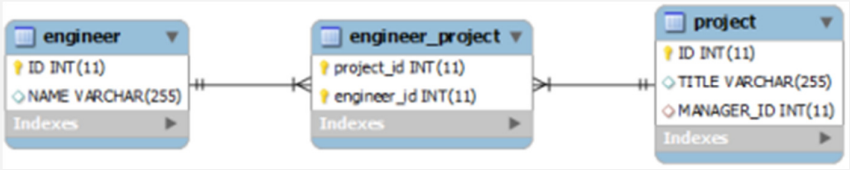
- An engineer should in which projects he is working.
- The project should know the engineers it occupies.

```
@Entity
public class Engineer implements Serializable {
```

```
    @Id
    private int id;

    private String name;

    @ManyToMany
    @JoinTable(name="ENGINEER_PROJECT",
     joinColumns=@JoinColumn(name="ENGINEER_ID"),
     inverseJoinColumns=@JoinColumn(name="PROJECT
_ID"))
    private Collection<Project> projects;

}

@Entity
public class Project implements Serializable {

    @Id
    private int id;

    private String title;

    @ManyToMany(mappedBy="projects")
    private Collection<Engineer> engineers;

}
```

```
mysql> describe engineer;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| ID    | int(11)      | NO   | PRI | NULL    |       |
| NAME  | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+

mysql> describe project;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| ID    | int(11)      | NO   | PRI | NULL    |       |
| TITLE | varchar(255) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+

mysql> describe engineer_project;
+-------------+---------+------+-----+---------+-------+
| Field       | Type    | Null | Key | Default | Extra |
+-------------+---------+------+-----+---------+-------+
| PROJECT_ID  | int(11) | NO   | PRI | NULL    |       |
| ENGINEER_ID | int(11) | NO   | PRI | NULL    |       |
+-------------+---------+------+-----+---------+-------+
```

Physical description of a many-to-many bidirectional relationship




ER diagram of a many-to-many bidirectional relationship

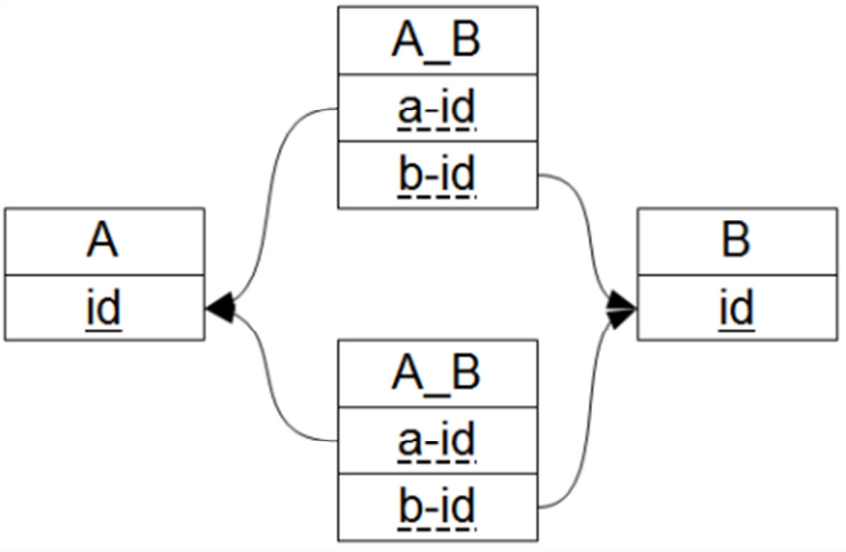
# Pitfall: Many-to-Many (one direction)

Not specifying the mappedBy element in a many-to-many relationship, assumes two join tables and should be avoided.

```
@Entity
class A {
   @Id int id;
   @ManyToMany
   Collection<B> listOfB;
}

@Entity
class B {
   @Id int id;

   @ManyToMany
   Collection<A> listOfA;
}
```

The same happens when the ManyToMany annotation is ommitted on one side.

```
@Entity
class A {
   @Id int id;

   @ManyToMany
   Collection<B> listOfB;
}

@Entity
class B {
   @Id int id;

   Collection<A> listOfA;
}
```




Relation model of two many-to-many unidirectional relationship

To avoid this pitfall, I would follow the instructions of Many-to-Many (both directions) relationship.

### References


- M. Keith, M. Schincariol, *"Pro JPA 2: Mastering the Java™ Persistence API"*, Apress, United States of America, 2009.