

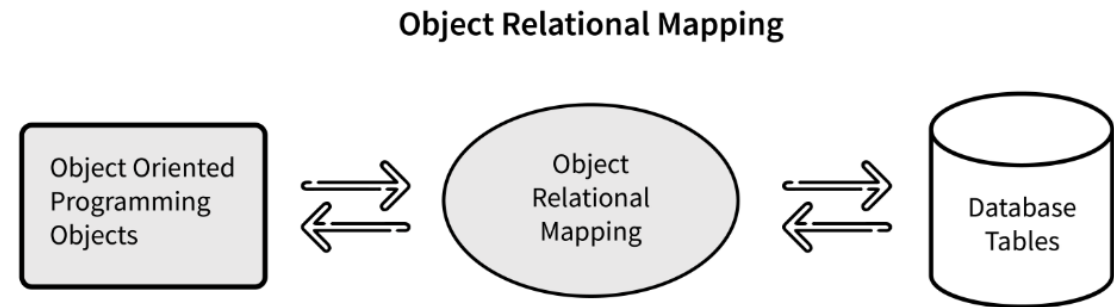


HIBERNATE

WITH JPA

ORM FRAMEWORK

- **ORM stands for Object Relational Mapping .**
- **ORM functionally used to develop and maintain a relationship between Object and Relational Database.**
- **ORM helps to map object states to database columns.**
- **It is capable of handling different database actions easily such as inserting , updating , deleting etc.....**
- **Various frameworks functions on ORM**
 - **Hibernate**
 - **Ibatis**
 - **Toplink**
 - **Django**
 - **ORMLite etc....**



WHAT IS JPA ?

- **JPA stands for java persistence API , its present in javax.persistence package.**
- **It provides specifications to ORM tool which is used to access, manage and persist data between object and relational database.**
- **JPA considered as standard approach for the ORM .**
- **JPA only provides specifications it doesn't perform any task .**
- **JPA required implementations .so , ORM tools like Hibernate , toplink ,ibatis etc.. Implements JPA specifications for data persistence.**

WHAT IS HIBERNATE ?

- **Hibernate is a framework which simplifies the development of java application to interact with database.**
- **It is open source ,lightweight ORM tool .**
- **Hibernate implements the specification of JPA for data persistence .**
- **Basically we use ORM tool to simplify the database actions like creation , data manipulation and data access .**
- **ORM is a programming technique that maps the object to the database row .**
- **Hibernate internally used JDBC API to interact with database.**
- **Hibernate was developed by Gavin king in 2001 .**



POM.XML

- **Pom.xml is a xml file which is used to add dependencies related to project which helps to build the project.**
- **Instead of adding jar files we can add dependencies .**

PERSISTENCE.XML

- **Persistence.xml is an configuration file used to configure a given JPA persistence unit .**
- **The persistence.xml file defines all the metadata we need in order to launch JPA EntityManagerFactory .**

Persistence unit

- **Persistence unit defines all the data required to launch an EntityManagerFactory .**
- **Like data source , mapping and transaction settings .**
- **One persistence.xml file can have multiple persistence units.**
- **Persistence unit will have all the DB config info .**

ENTITYMANAGERFACTORY

- **EntityManagerFactory** is an interface present in `javax.persistence` package .
- The main role of **EntityManagerFactory** instance is to support instantiation of **EntityManager**.
- It is constructed for specific database , and by managing resources efficiently it provides an efficient way to construct multiple **EntityManager** instances for that database.
- By using `createEntityManagerFactory("persistence unit name")` method of **Persistence** class we can create **EntityManagerFactory** instance.
- For this method have to pass persistence unit name as argument then it will establish connection for the particular database to perform actions.

Example :

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("abc") ;
```


ENTITYMANAGER

- **EntityManager** is an interface present in **javax.persistence** package .
- It provides many methods by using them we can perform **CRUD** operations on the database tables .
- By using **createEntityManager()** method of **EntityManagerFactory** interface we can create the instance of **EntityManager** .
- We can create multiple **EntityManager** instances by using same **EntityManagerFactory** instance to perform actions on particular database .

Example :

EntityManager **em** = **emf.createEntityManager()** ;

ENTITYTRANSACTION

- **To perform any database actions which will affect the table data we have to must use the methods specified in the EntityManager interface .**
- **Mostly we use two methods of the EntityManager interface are begin() and commit() .**
- **Where begin() method we have to invoke at first before the EntityManager methods gets invoked .**
- **After invoking all the entity methods we can invoke the commit() method to execute all the methods which invoked from EntityManager interface .**
- **Without begin() and commit() we can't perform database actions like insert , delete and update.**
- **For select operation need not to use the EntityManager methods.**

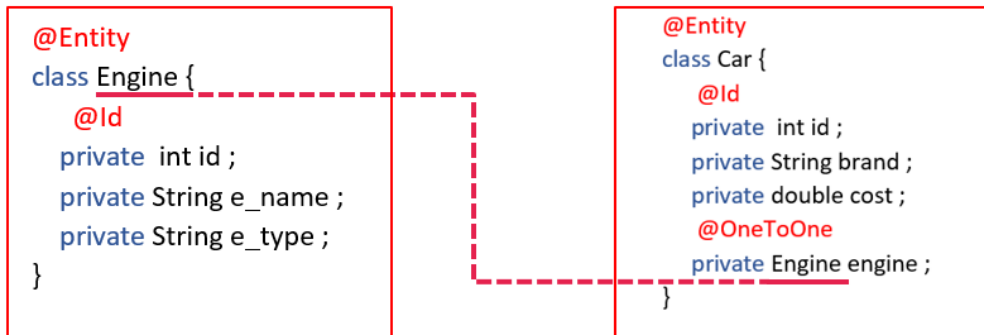
MAPPING IN HIBERNATE

- **Hibernate mapping is one of the key features of the hibernate .**
- **By using mapping we can establish connection between two or more tables.**
- **We can establish either unidirectional or bidirectional relationship .**
- **The relationships can be established between entities are :**
 - 1. One to one**
 - 2. One to many**
 - 3. Many to one**
 - 4. Many to many**

ONE TO ONE UNIDIRECTIONAL

- One entity object having relationship with another one entity object is known as one to one relationship.
- With the help of `@OneToOne` annotation we can achieve this relationship.
- We can achieve one to one unidirectional relation between two entity classes by declaring the data member of one entity class type in another entity class, then we should annotate data member with `@OneToOne`.
- In one to one unidirectional relationship only foreign key column will get created for the table in which entity class has declaration of the another entity class. So we can fetch the data from one side only.

Example:



Result table in database:

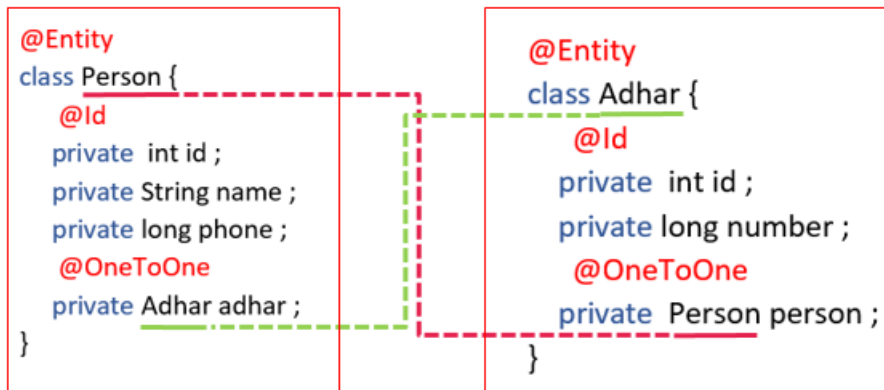
id	e_name	e_type

id	brand	cost	engine_id

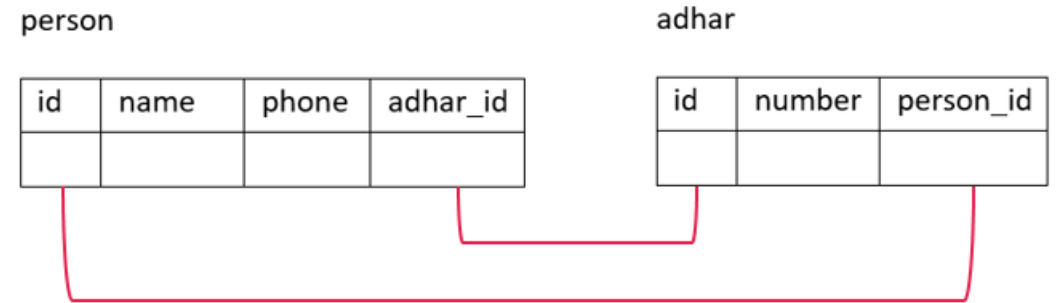
ONE TO ONE BIDIRECTIONAL

- For one to one bidirectional relationship both the entity classes should have data member declared of corresponding entity class and should be annotated with `@OneToOne` .
- In one to one bidirectional relationship foreign key column will get create for both tables. So we can fetch the data from both the sides.

Example:



Result table in database:



ONE TO MANY UNIDIRECTIONAL

- In one to many relationship one entity type object will have many relations with another entity type objects.
- In order to achieve one to many unidirectional relationship we have to create the data member type list of entity type , inside the entity class which willing to build many relations .
- We have to mention the **@OneToMany** annotation on the list data member to achieve one to many relation.
- For one to many relation in database there will be a third table to maintain the relationship mapping data because database cell can't hold multiple values in a single cell.
- in one to many unidirectional relation we can access the data only one side from where the data member was declared.

Example:

```
@Entity
class Laptop {
    @Id
    private int id ;
    private String brand ;
    private String color ;
}

@Entity
class Shop{
    @Id
    private int id ;
    private String name ;
    @OneToMany
    private List<Laptop> laptops ;
}
```

Result table in database:

shop

id	name
1	Cyber tech

laptop

id	brand	color
1	hp	Grey
2	Asus	Black
3	Dell	white

shop_laptop

Shop_id	Laptops_id
1	1
1	2
1	3

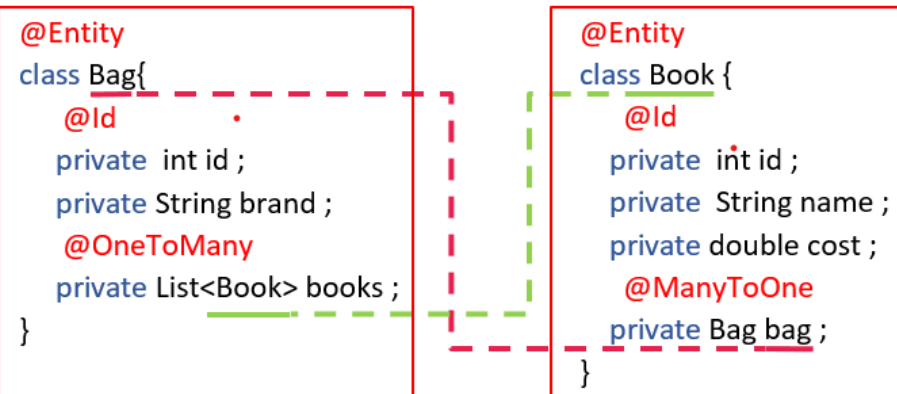
ONE TO MANY BIDIRECTIONAL

- For one to many bidirectional mapping we have to use two different types of data members as well as two different types of annotations.
- The entity class object which has many relationships in that class we have to create the list of entity type of the corresponding entity class along with that need to annotated with `@OneToMany`.
- In the other entity class group of objects having relation with the one entity object of the other class, should be declared with the data member of the entity along with need to annotated with `@ManyToOne`.
- In one to many bidirectional we can achieve data fetching from both the sides.
- There will be three tables in database the one extra table which records the one to many mapping data and for many to one relation mapping foreign column will get generated in table.

Example:

```
@Entity
class Bag{
    @Id
    private int id ;
    private String brand ;
    @OneToMany
    private List<Book> books ;
}

@Entity
class Book {
    @Id
    private int id ;
    private String name ;
    private double cost ;
    @ManyToOne
    private Bag bag ;
}
```



Result table in database:

bag		book				bag_book	
id	brand	id	name	cost	bag_id	Bag_id	books_id
1	Sky bags	1	Harry potter	399.99	1	1	1
2	wildcraft	2	Two states	159.86	1	1	2
		3	How to become rich	999.00	2	2	3
		4	Rich dad poor dad	599.00	2	2	4

MANY TO ONE UNIDIRECTIONAL

- In many to one relationship many objects of same entity will have relation with one object of other entity type . We can say it is reverse of one to many relationship .
- In order to achieve many to one unidirectional relationship we have to create the data member of entity type and need to annotate with @ManyToOne .
- in many to one unidirectional relation we can access the data only one side from where the data member was declared or in which table contains the foreign key column.

Example:

```
@Entity
class Mobile {
    @Id
    private int id ;
    private String brand ;
    private double cost ;
}
```

```
@Entity
class Sim{
    @Id
    private int id ;
    private String service ;
    @ManyToOne
    private Mobile mobile ;
}
```

Result table in database:

mobile

id	brand	cost
1	Moto	13999.19

sim

id	service	mobile_id
1	Airtel	1
2	Jio	1

MANY TO MANY UNIDIRECTIONAL

- In many to many unidirectional relation one entity class object having multiple relations with the other entity class and vice versa .
- Many to many unidirectional can be achieved by creating List of entity type data member of the one entity class in another entity class and it should be annotated using @ManyToMany annotation.
- In many to many unidirectional there will be extra table will get created in database to maintain the mapping information .
- Here also we can access the data only from one side where the data member is declared .

Example:

@Entity

```
class Person{  
  @Id  
  private int id ;  
  private String name ;  
  private long phone ;  
}
```

@Entity

```
class Cab{  
  @Id  
  private int id ;  
  private String type;  
  private String number ;  
  @ManyToMany  
  private List< Person > persons ;  
}
```

Result table in database:

person

id	name	phone
1	bunny	7001444666
2	gani	9988899988

cab

id	type	number
1	suv	KA 01 ej 1112
2	sedan	KA 05 jp 0023

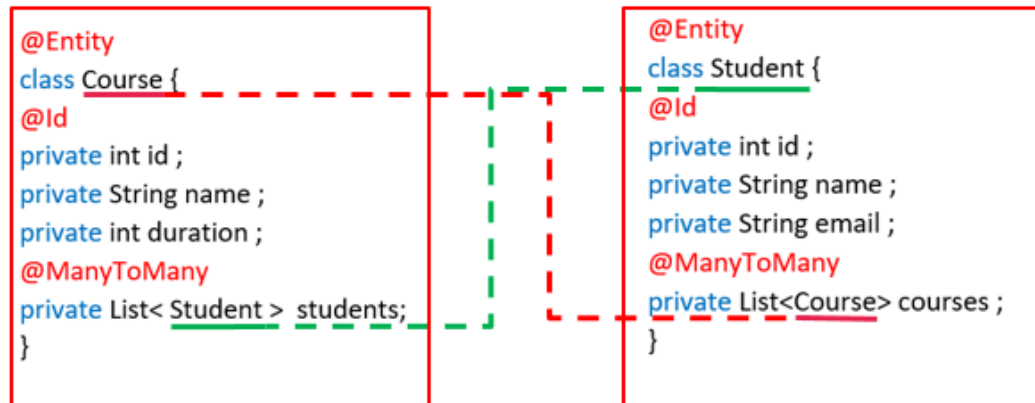
Cab_person

cab_id	person_id
1	1
1	2
2	2
2	1

MANY TO MANY BIDIRECTIONAL

- For many to many bidirectional we have to create List data member of the corresponding entity class type for both the entity classes and data members should be annotated with @ManyToMany.
- In many to many bidirectional there will be two additional tables will get generated in the database.
- Both the table uses to maintain them.

Example:



Result table in database:

course

Id	name	duration
1	Java	6
2	Python	3
3	JS	5

student

Id	name	email
1	rita	rit@mail.com
2	geeta	geeet@mail.com
3	seeta	seet@mail.com

course_student

course_id	student_id
1	2
1	3
2	1
3	2
3	1

student_course

student_id	course_id
1	2
1	3
2	1
2	3
3	1

@JOINCOLUMN & MAPPEDBY

- **In bidirectional mapping where every result tables will holds the duplicate or repeated data. There is possibility to achieve the bidirectional relationship without holding the duplicate data.**
- **But first we have to decide which table should own the foreign key column and which table should reefer the foreign key Colum present in other table .**
- **We have to use @JoinColumn annotation on the owning side and mappedby to the referral side .**
- **We have to use the @JoinColumn under the relationship mapping annotation and mappedBy is an attribute which can be used within the parenthesis of the mapping annotation of referral side and need to pass the value as the variable name which is used for the .**
- **By using this process we can avoid the creation of additional table which gets created for @ManyToOne bidirectional relationship.**
- **By using this process we can't join the additional tables which gets created while building @ManyToMany bidirectional relationship also we cant avoid the additional table which gets created while achieving @ManyToMany unidirectional .**

@JOINCOLUMN & MAPPEDBY

@JoinColumn & mappedBy for @OneToOne

- Under the @OneToOne annotation we have to declare the @JoinColumn at owning side .
- Within the parenthesis of the referring side @OneToOne annotation we have to use mappedBy attribute and need to initialize with the variable name which is specified in another entity class.

Example:

Result table in database:

```
@Entity
class Person {
    @Id
    private int id ;
    private String name ;
    private long phone ;
    @OneToOne
    @JoinColumn
    private Adhar adhar ;
}

@Entity
class Adhar {
    @Id
    private int id ;
    private long number ;
    @OneToOne (mappedBy = "adhar ")
    private Person person ;
}
```

Owning side (points to `@JoinColumn` in Person class)

Referring side (points to `mappedBy = "adhar "` in Adhar class)

person

id	name	phone	adhar_id

adhar

id	number

@JOINCOLUMN & MAPPEDBY

@JoinColumn & mappedBy for @OneToMany

- We have to specify the @JoinColumn under the @OneToMany annotation and have to declare the mapped by attribute within the parenthesis of the @ManyToOne and need to initialize it with the variable name which is specified in other entity class .
- Here we cant avoid the table creation because owning side entity having many relations with its corresponding class .

Example:

```
@Entity
class Apartment {
    @Id
    private int id ;
    private String name ;
    @OneToMany
    @JoinColumn
    private Plot plot ;
}

@Entity
class Plot {
    @Id
    private int id;
    private int door_no;
    private String owner_name;
    @ManyToOne(mappedBy = "plot")
    private Apartment apartment;
}
```

Owned side

referring side

Result table in database:

apartment	
Id	name
1	Swagruha

plot		
Id	door_no	owner_name
1	111	Akhil
2	112	Kushal
3	113	hoshitha

apartment_plot	
apartment_Id	plot_id
1	1
1	2
1	3

@JOINCOLUMN & MAPPEDBY

@JoinColumn & mappedBy for @ManyToOne

- We have to specify the @JoinColumn under the @ManyToOne annotation and have to declare the mapped by attribute within the parenthesis of the @OneToMany and need to initialize it with the variable name which is specified in other entity class .
- Here we can avoid the additional table creation .

Example:

```
@Entity
class Plot {
    @Id
    private int id;
    private int door_no;
    private String owner_name;
    @ManyToOne
    @JoinColumn
    Private Apartment apartment;
}

@Entity
class Apartment {
    @Id
    private int id;
    private String name;
    @OneToMany(mappedBy = "apartment ")
    private Plot plot ;
}
```

Owned side

referring side

Result table in database:

apartment	
Id	name
1	Swagruha

plot			
Id	door_no	owner_name	apartment_Id
1	111	Akhil	1
2	112	Kushal	1
3	113	hoshitha	1

@JOINTABLE & MAPPEDBY

- **@JoinTable** is used to eliminate the additional table which gets generated in many to many bidirectional .
- **Basically there will be two tables will get generated in order to achieve many to many bidirectional which holds the duplicate data .**
- **We can avoid any one of the table which containing the duplicate data by using @JoinTable and mappedBy.**
- **For @JoinTable we have to declare and initialize two attributes as listed below.**
 - **joinColumns = @JoinColumn**
 - **inverseJoinColumns = @JoinColumn**
- **mappedBy should declare within the parenthesis of the refereeing side @ManyToMany annotation an need to initialize it with reference variable name .**

Example:

```
@Entity
class Cab{
    @Id
    private int id ;
    private String type;
    private String number ;
    @ManyToMany
    @JoinTable (joinColumns = @JoinColumn,
               inverseJoinColumns = @JoinColumn)
    private List< Person > persons ;
}

@Entity
class Person{
    @Id
    private int id ;
    private String name ;
    private long phone ;
    @ManyToMany (mappedBy = "persons")
    private List<Cab> cabs ;
}
```



Result table in database:

person			cab			Cab_person	
id	name	phone	id	type	number	cab_id	person_id
1	bunny	7001444666	1	suv	KA 01 ej 1112	1	1
2	gani	9988899988	2	sedan	KA 05 jp 0023	1	2
						2	1

CASCADING

- To establish a dependency between related entities, JPA provides `javax.persistence.CascadeType` that define the cascade operations.
- When we perform any actions on a entity, then cascading will perform the same operations on the dependent objects.
- These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many.

Cascade Operations	Description
PERSIST	In this cascade operation, if the parent entity is persisted then all its related entity will also be persisted.
MERGE	In this cascade operation, if the parent entity is merged then all its related entity will also be merged.
DETACH	In this cascade operation, if the parent entity is detached then all its related entity will also be detached.
REFRESH	In this cascade operation, if the parent entity is refreshed then all its related entity will also be refreshed.
REMOVE	In this cascade operation, if the parent entity is removed then all its related entity will also be removed.
ALL	In this case, all the above cascade operations can be applied to the entities related to parent entity.

ENTITY LIFECYCLE

- In hibernate we can create the instance of the entity class and we can map the entity instance to the database table and we can fetch the existing data of the entity from the database .
- So each entity is associated with the lifecycle the entity instances passes through different phases of the lifecycle .
- The hibernate entity life cycle contains following steps :
 1. Transient state
 2. Persistence state
 3. Detached state

Transient state :

- It is initial state of the entity object .
- Once after creating instance of the POJO class then object enters into transient state .
- In this state object and table wont have any association ,so the modifications took place in object that don't show any impact on database table.

ENTITY LIFECYCLE

Persistence state:

- **When the object associated with the session ,it enters into the persistence state .**
- **When we invoke persist() or merge() methods object enters into the persistence state.**
- **Where modification on the object will show effect on the entity mapped in database.**

Detached state:

- **When we break the association between instance and the database then object will enter into detached state .**
- **We can use detach() method to detach the object from the database .**
- **We can take back object to persist state from the detached state by invoking the load() , merge() and save() methods .**

