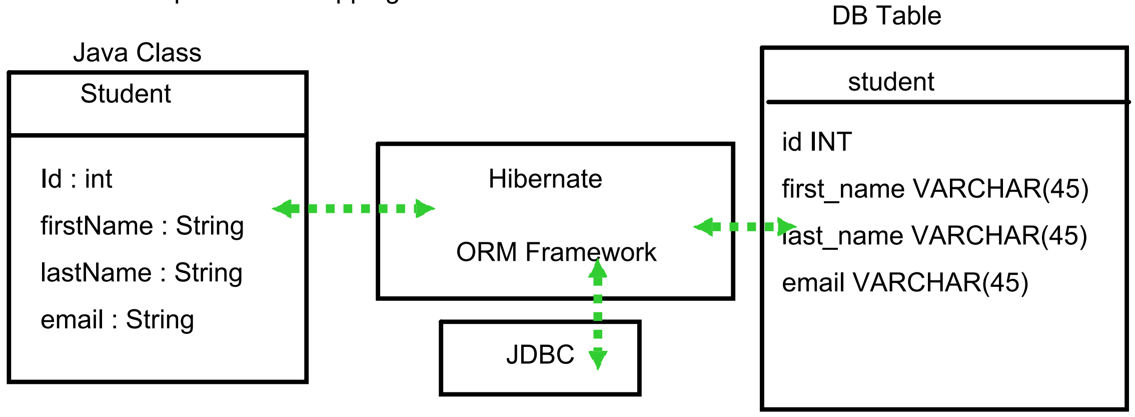
ORM : Object to relational mapping (hibernate)

Hibernate: framework for persisting/saving POJO in database

Benefit:

Hibernate handles all low-level SQL

ORM : developer define mapping between Java class and db table



Any interaction with DB would be OO

eg:

```
//create a POJO
  Student student=new Student("First","Last","first@mail.com");

//persisting in DB
//session : hibernate object(encpsulated the low level db complexities)
session.save(student);
Student stud=session.get(<id>);
```

#appropriate method to perform all DB activities (std CRUD)

#customization : HQL : Hibernate query language

- # simple syntax
- #use of entity class names instead of db schema
- #uniform for any backend DB
- #exception generated by malicious query formation would throw spring dao exception

Setup environment for Hibernate implementation

#Dependency to add

#Config for Hibernate (spring config)

#implement at DAO

Dependency :

1. Spring support for ORM and Transaction handlers

   #to connect with ORM framework : need spring handler

   #Transaction handler : ORM(Hibernate) works in transactional manner

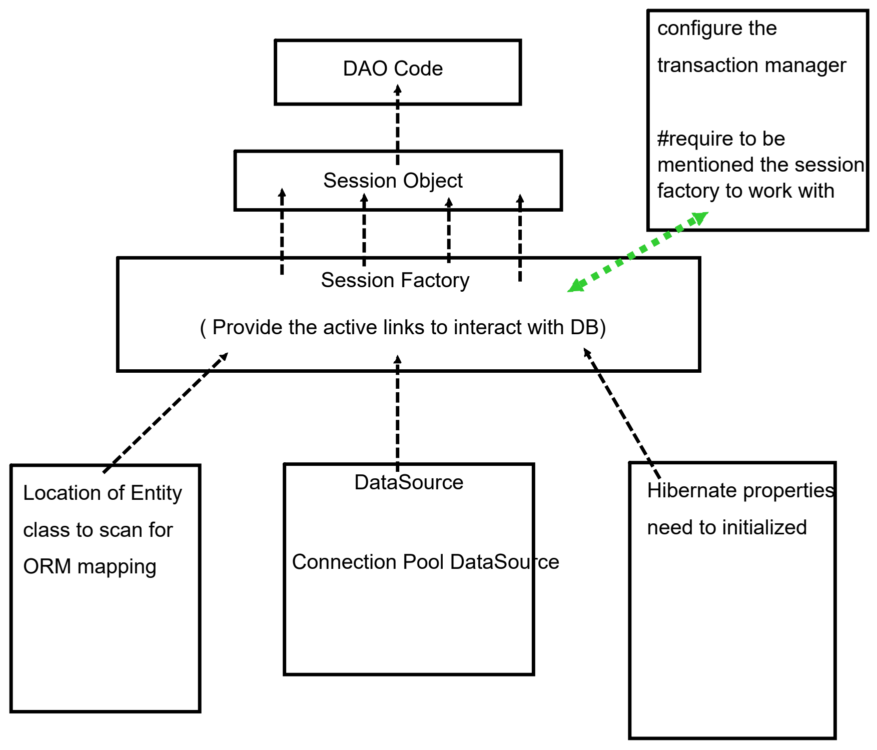
(Hibernate encapsulates all DB activity under transaction)

2. Hibernate Core
3. Support for datasource : Connection pool (efficiency)
4. jdbc-mysql-connector

Hibernate Config : SessionFactory (manages session : initiate/destroy/set up connection/close connection)

session : generate sql query/interactions with db

Configuring Session Factory


DAO Code
Session Object
configure the transaction manager
#require to be mentioned the session factory to work with
Session Factory
(Provide the active links to interact with DB)
Location of Entity class to scan for ORM mapping
DataSource
Connection Pool DataSource
Hibernate properties need to initialized

Entity class mapping:

For ORM to map POJO to db tables

#need to config the mapping

1. XML config file (legacy)
2. Java Annotations (preferred)

Mapping:

1. MAP class to db table
2. MAP fields to DB col

All ORM implementation are based on JPA

suggested to use the core JPA annotation (prevent vendor locking)

Hibernate automatically generate required SQL behind the scene:

need to specify the dialect (syntatical rule corresponding to backend DB Server eg: Oracle,Postgre, MySQL..)

==>All hibernate activities are transaction based

#need to configure Transaction manager (work in background)

#need to enable the transaction management (add annotation in config file)

#All Hibernate dao methods must be decorated with @transactional annotation

#Multiple table

#Relationships between table

==>.Model this with Hibernate

Mapping:
- => One to One
- => One to Many, Many to One
- =>Many to Many

Relationships : Important DB Concepts
1. Primary Key and Foreign Key
2. Cascade

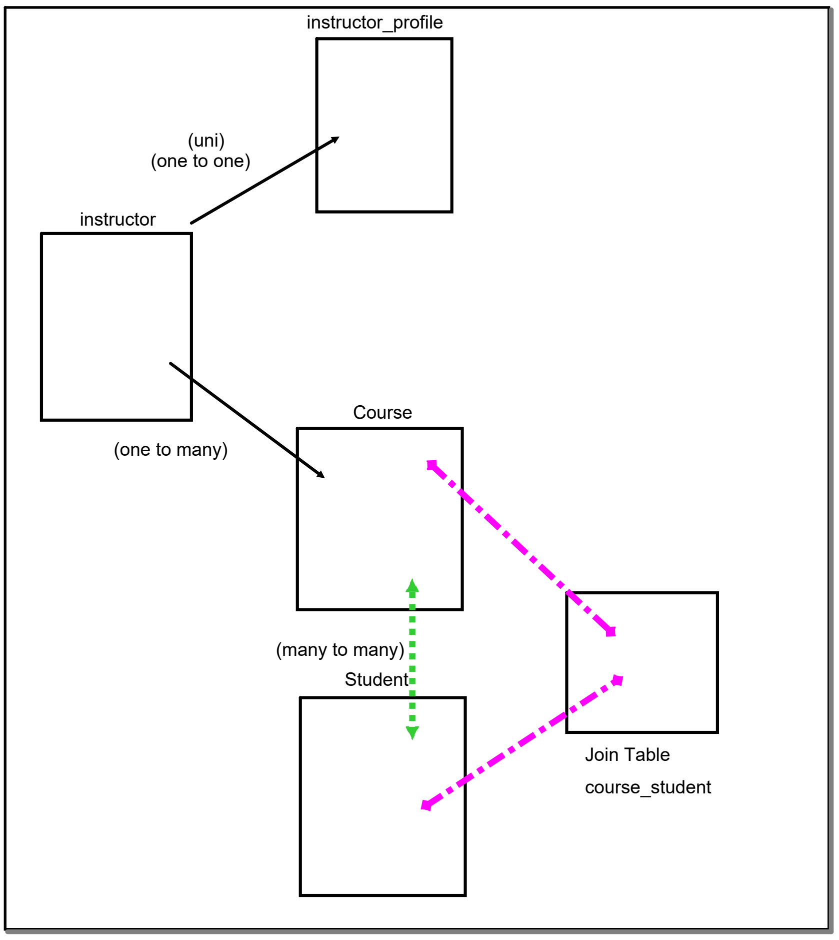
Primary Key : identify a unique row in a table
Foreign key:
Link Table together
a field in one table refers to primary key in another table...

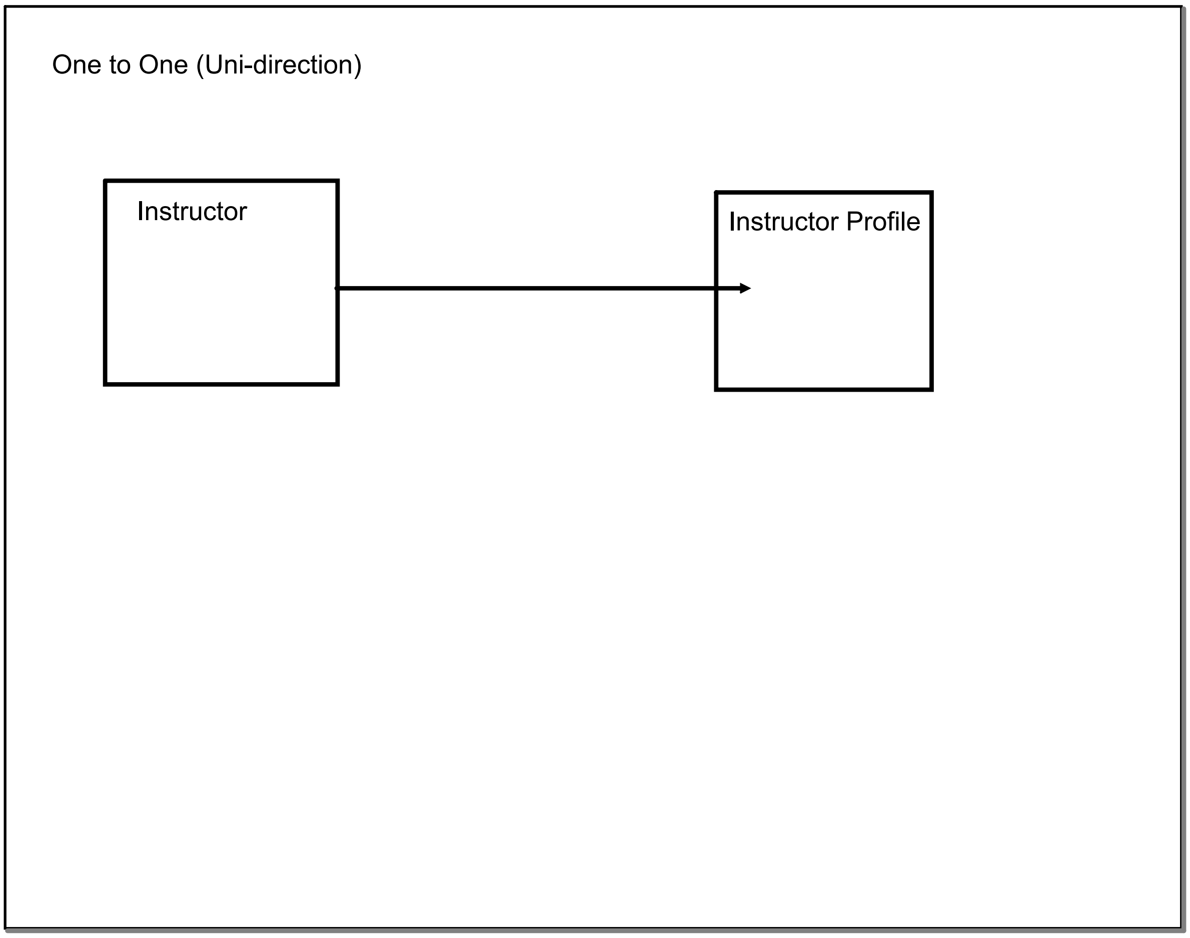
Cascade:
can cascade operations among related tables
#Apply the same operation to related entites

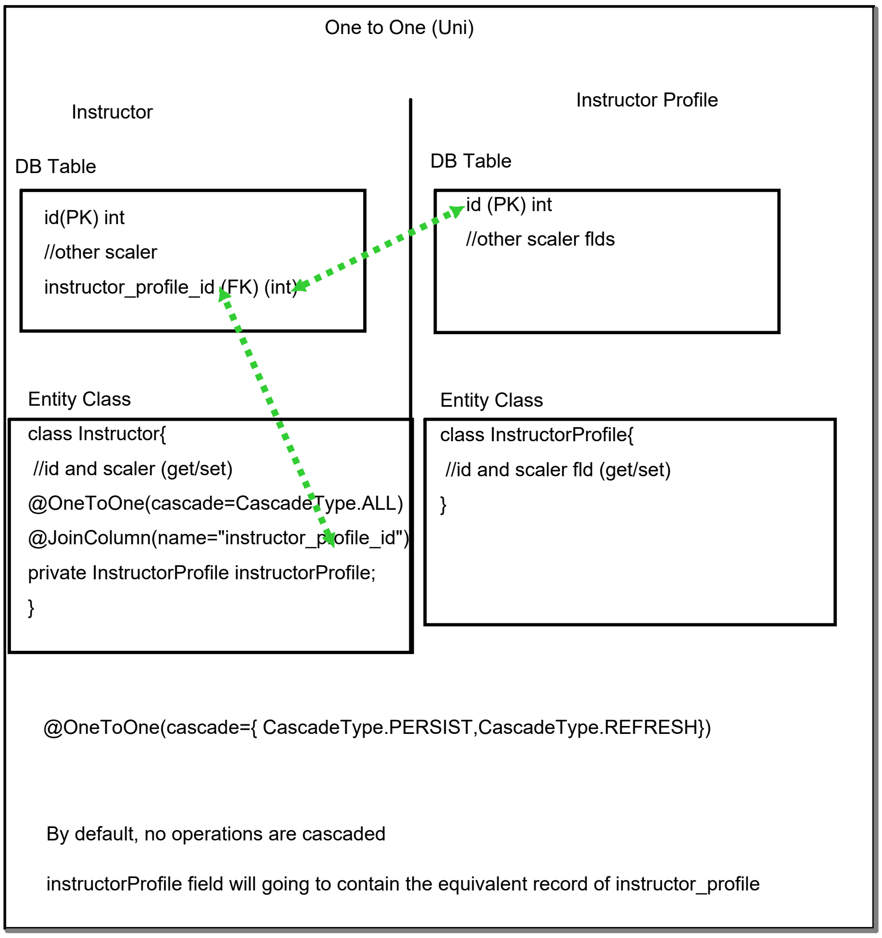
Cascading need to used cautiously : can be configured

Data Fetch : Eager and Lazy Loading
fetching data of related table based on two config
Eager : retrieve everything directly
Lazy : retrieve on demand

=>Required changes/structure of DB table
=>Required changes/structure of corresponding db classes


instructor_profile
(uni)
(one to one)
instructor
Course
(one to many)
(many to many)
Student
Join Table
course_student


One to One (Uni-direction)
Instructor
Instructor Profile



```
id(PK) int
//other scaler
instructor_profile_id (FK) (int)
```

```
id (PK) int
//other scaler flds
```

```
class Instructor{
 //id and scaler (get/set)
@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name="instructor_profile_id")
private InstructorProfile instructorProfile;
}
```

```
class InstructorProfile{
 //id and scaler fld (get/set)
}
```

```
@OneToOne(cascade={ CascadeType.PERSIST,CascadeType.REFRESH})
```

By default, no operations are cascaded

instructorProfile field will going to contain the equivalent record of instructor_profile

Cascade Type

#PERSIST : If entity is persisted/saved , related entity is also saved

#REMOVE : delete

Session object cascading :

Session allows to re-fetch(refresh) the record

#REFRESH :

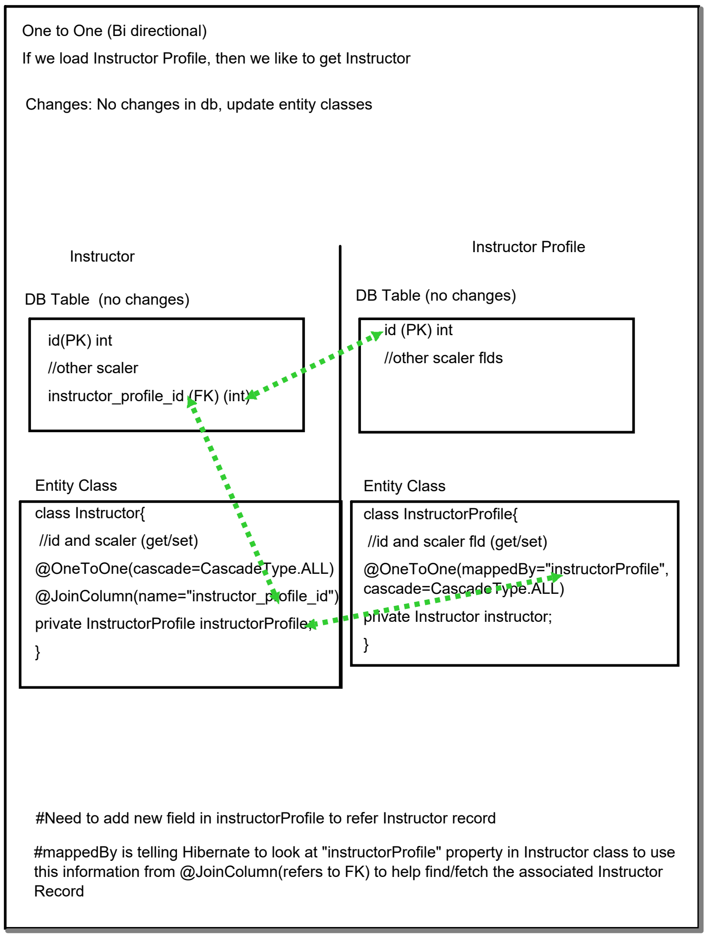
detach : entity is no more associated with session

#DETACH :

merge: to reconnect/re-attach entity with session object

#MERGE:

#ALL : All above operation would be cascaded among related entities



One to One (Bi directional)

If we load Instructor Profile, then we like to get Instructor

Changes: No changes in db, update entity classes

```
id(PK) int
//other scaler
instructor_profile_id (FK) (int)
```

```
id (PK) int
//other scaler flds
```

```
class Instructor{
 //id and scaler (get/set)
@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name="instructor_profile_id")
private InstructorProfile instructorProfile
}
```

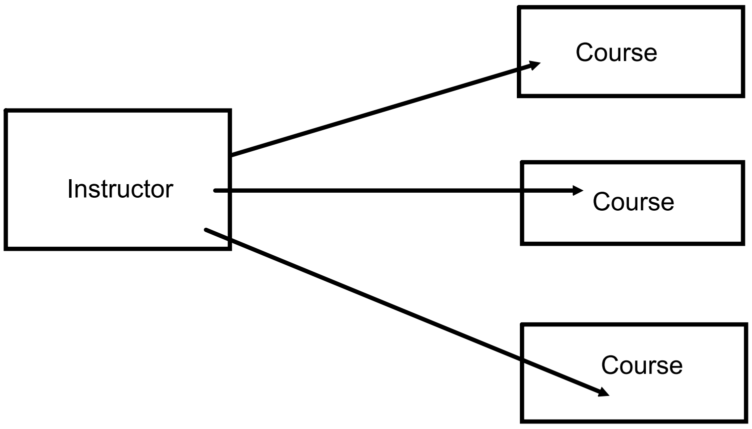
```
class InstructorProfile{
 //id and scaler fld (get/set)
@OneToOne(mappedBy="instructorProfile",
cascade=CascadeType.ALL)
private Instructor instructor;
}
```

#Need to add new field in instructorProfile to refer Instructor record

#mappedBy is telling Hibernate to look at "instructorProfile" property in Instructor class to use this information from @JoinColumn(refers to FK) to help find/fetch the associated Instructor Record

One to many (Bi)

Instructor can have many course

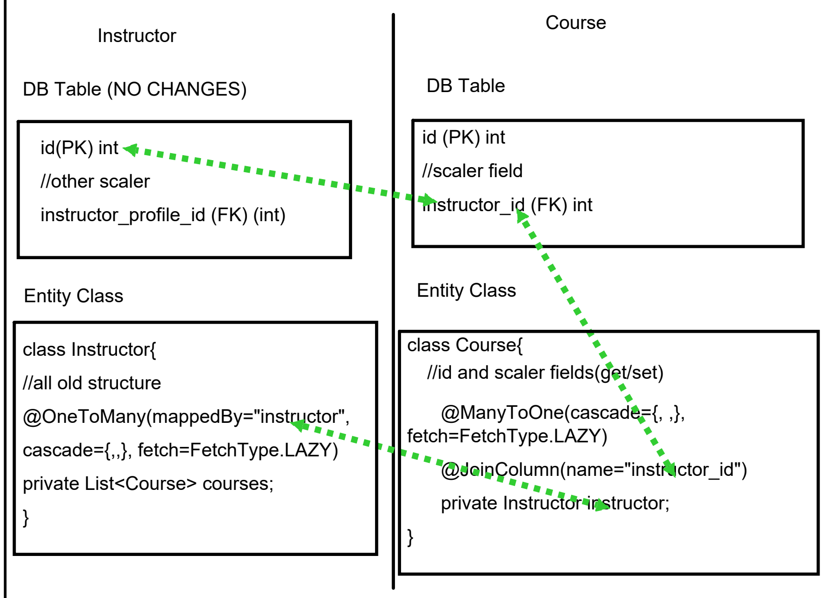


Inverse : Many courses can have one instructor

Many to One

Cascading: sensitive

One to Many (bi)



```
id(PK) int
//other scaler
instructor_profile_id (FK) (int)
```

```
class Instructor{
//all old structure
@OneToMany(mappedBy="instructor",
cascade={,,}, fetch=FetchType.LAZY)
private List<Course> courses;
}
```

```
id (PK) int
//scaler field
instructor_id (FK) int
```

```
class Course{
   //id and scaler fields(get/set)
      @ManyToOne(cascade={, ,},
fetch=FetchType.LAZY)
      @JoinColumn(name="instructor_id")
      private Instructor instructor;
}
```

#no Cascade deleting

mappedBy : telling hibernate to look as "instructor" property in Course class to use info from @JoinColumn to help find/fetch associated courses for an instructor

Best Practice:

Only load data when absolutely needed

Prefer LAZY LOADING instead of Eager Loading

Default Fetch Type:

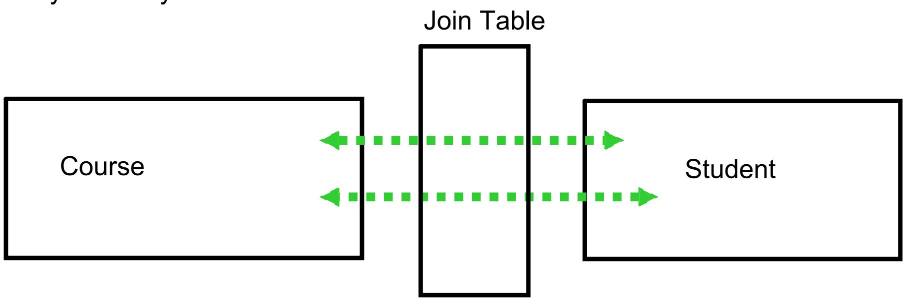
OneToOne : EAGER

OneToMany : LAZY

ManyToOne:EAGER

ManyToMany:LAZY

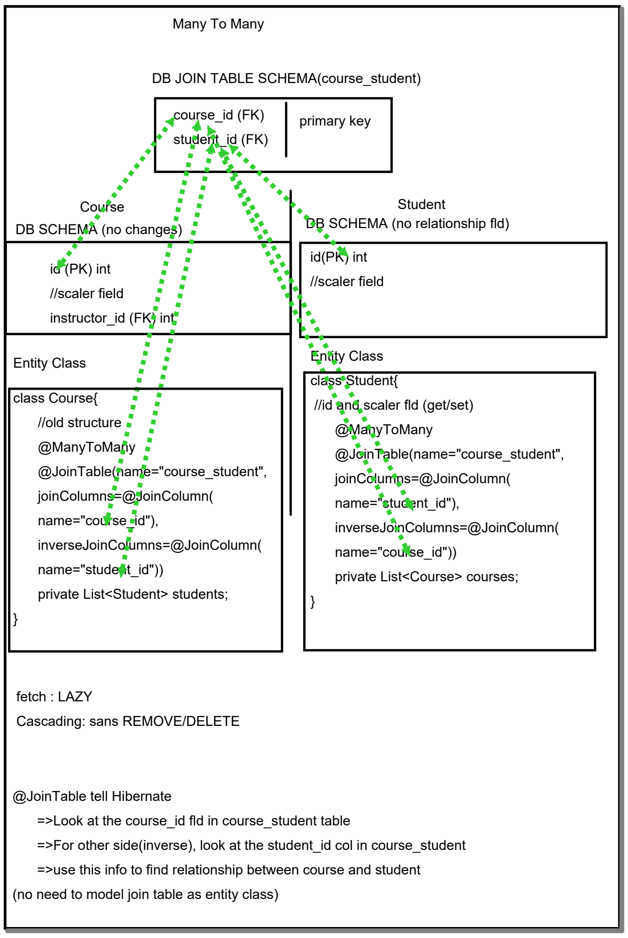
Many To Many



Need to track which student is in which course and vice-versa

Join Table : A table that provides a mapping between two tables

It has foreign key for each table to define the mapping



Many To Many

DB JOIN TABLE SCHEMA(course_student)

course_id (FK) student_id (FK)	primary key

Course
DB SCHEMA (no changes)

```
id (PK) int
//scaler field
instructor_id (FK) int
```

Entity Class

```
class Course{
    //old structure
    @ManyToMany
    @JoinTable(name="course_student",
    joinColumns=@JoinColumn(
    name="course_id"),
    inverseJoinColumns=@JoinColumn(
    name="student_id"))
    private List<Student> students;
}
```

Student
DB SCHEMA (no relationship fld)

```
id(PK) int
//scaler field
```

Entity Class

```
class Student{
 //id and scaler fld (get/set)
    @ManyToMany
    @JoinTable(name="course_student",
    joinColumns=@JoinColumn(
    name="student_id"),
    inverseJoinColumns=@JoinColumn(
    name="course_id"))
    private List<Course> courses;
}
```

fetch : LAZY
Cascading: sans REMOVE/DELETE

@JoinTable tell Hibernate
- =>Look at the course_id fld in course_student table
- =>For other side(inverse), look at the student_id col in course_student
- =>use this info to find relationship between course and student

(no need to model join table as entity class)