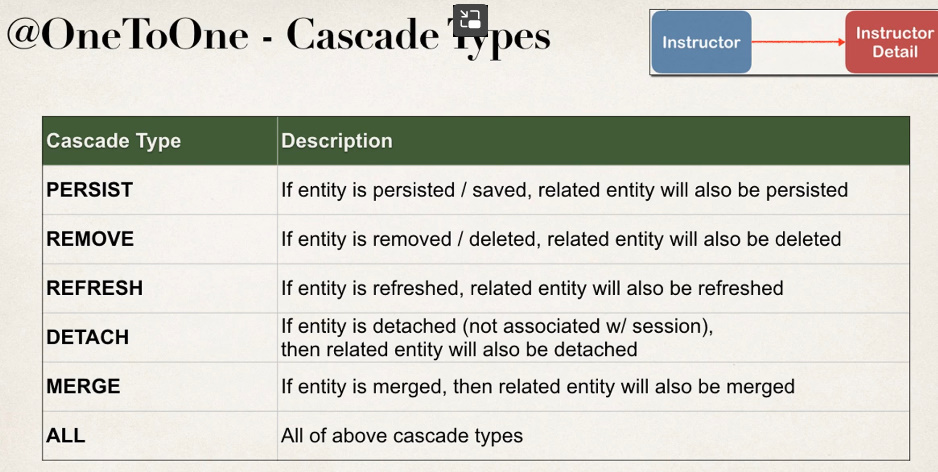
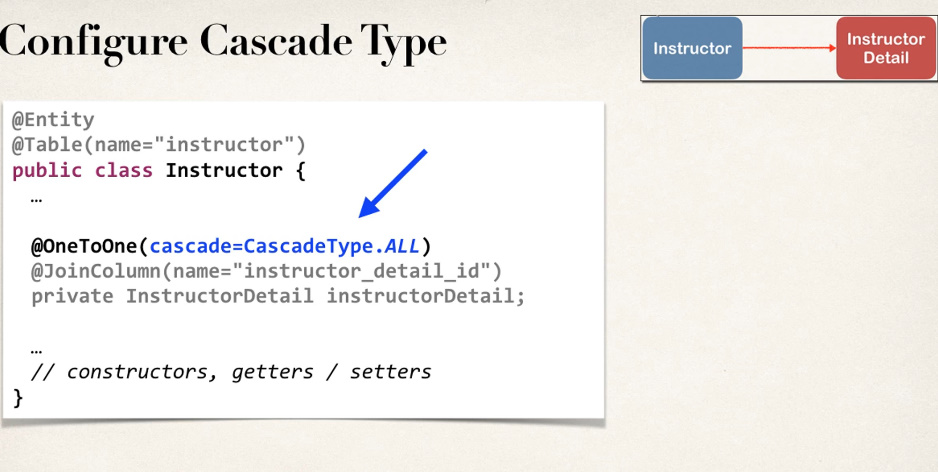
**@OneToOne Cascade Type**

Operatiile efectuate asupra unei entitati nu duc in mod default si la aplicarea lor asupra entitatii asociate, de accea avem nevoie de cascade, ca sa specificam care operatii efectuate asupra uneia vor fi efectuate si asupra alteia.

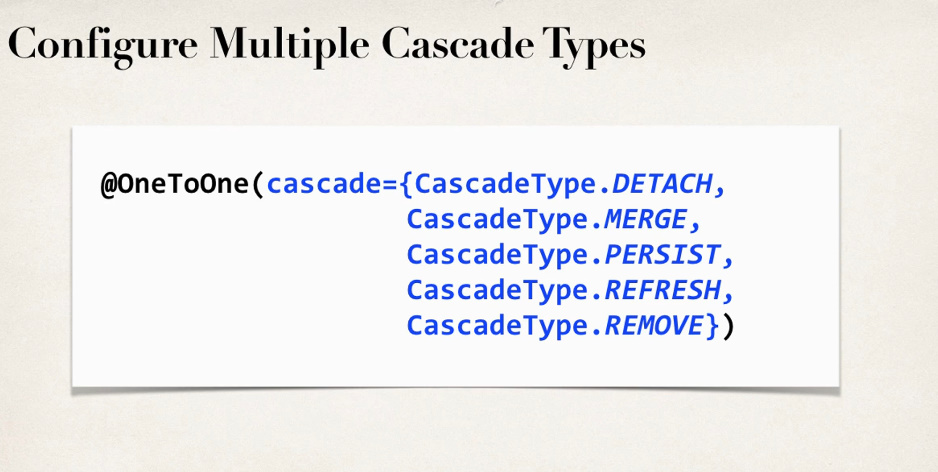


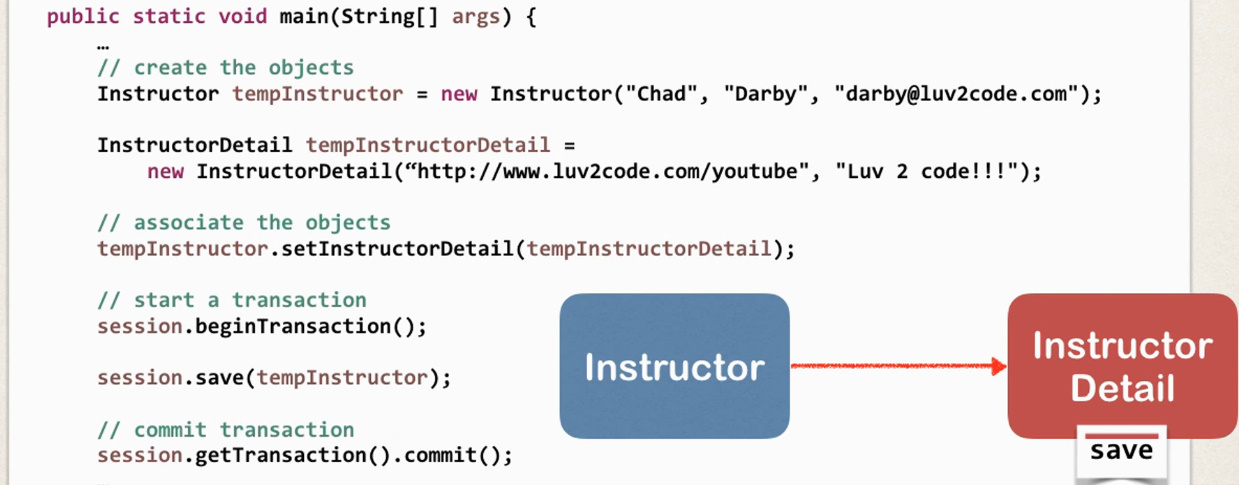


Tot ce vom face asupra enitatii Instructor va afecta si entitatea InstructorDetail

In mod default, nici-un operator nu este Cascaded!

Putem seta si cate dorim:





Aici, cream ca si cum 2 entitati, 2 obiecte din tabele diferite ca si cum, si setam pe a 2 in prima, ca prima sa poata crea asocierea cu ea.

**Add Entity**

1. Cream clasele

@Entity  
@Table(name="instructor\_detail")  
public class InstructorDetail {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "id")  
 private int id;  
  
 @Column(name="youtube\_channel")  
 private String youtubeChannel;  
  
 @Column(name="hobby")  
 private String hobby;

@Entity  
@Table(name="instructor")  
public class Instructor {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Column(name = "id")  
 private int id;  
  
 @Column(name="first\_name")  
 private String firstName;  
  
 @Column(name = "last\_name")  
 private String lastName;  
  
 @Column(name = "email")  
 private String email;  
  
 @OneToOne(cascade = CascadeType.*ALL*)  
 @JoinColumn(name="instructor\_detail\_id")  
 private InstructorDetail instructorDetail;

Nu adaugam si atributul instructor\_detail\_id! E de ajuns sa specificam entitatea cu care se asociaza tabelul in baza de date, ca sa facem asociere si intre entitati, si se va genera automat, ca si cum, un atribut instructor\_detail\_id,care e foreign key, desi clasa Instructor pur si simplu va folosi obiectul instructorDetail pentru a se referi la acel atribut.

1. Cream metoda main
2. SessionFactory sessionFactory = new Configuration().configure().addAnnotatedClass(Instructor.class).addAnnotatedClass(InstructorDetail.class).buildSessionFactory();  
   Session session = sessionFactory.getCurrentSession();  
     
   try{  
   session.beginTransaction();  
     
   Instructor instructor = new Instructor("Mititiuc","Eduard","edikutsu2002@mail.ru");  
   InstructorDetail instructorDetail = new InstructorDetail("AlfaLight","Programming");  
   instructor.setInstructorDetail(instructorDetail);  
     
   session.save(instructor);  
     
   session.getTransaction().commit();  
   }  
   catch (Exception e){  
    e.printStackTrace();  
   }  
   finally {  
    session.close();  
   }

* Atentie, cand cream SesstionFactory, trebuie sa ne referim la ambele entitati, ca si cum la ambele tabele din baza de date, de aceea punem de 2 ori addAnnotatedClass

addAnnotatedClass(Instructor.class).addAnnotatedClass(InstructorDetail.class).buildSessionFactory();  
Session session = sessionFactory.getCurrentSession();

* In rest, totul e la fel. Salvam doar obiectul instructor, caci mai apoi Hibernate se va asigura sa adauge in tabelul instructor si valoarea pentru foreign key, si in tabelul instructor\_detail la fel va salva.
* ATENTIE! Salvarea se va face pentru ambele entitati datorita faptului ca am setat cascade all:

@OneToOne(cascade = CascadeType.*ALL*)

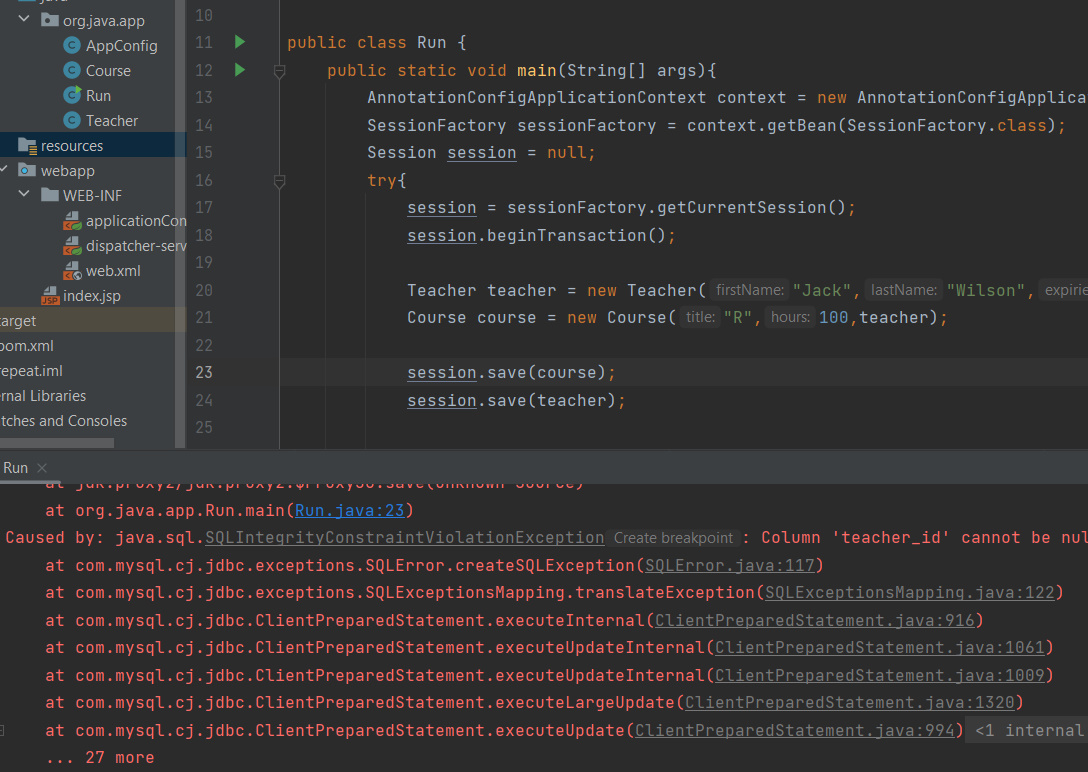
**Delete Entity**

int id = 1;  
Instructor instructor = session.get(Instructor.class,id);  
  
session.delete(instructor);  
session.getTransaction().commit();

Iarasi, se va sterge si InstructorDetail, datorita la @OneToOne(cascade = CascadeType.*ALL*)

**Atentie! Desi la @OneToMany avem Lazy fetch automat, oricum stergerea lui istructor, ce are in el o lista de courses, va duce si la stergerea lor. Cand s-a extras instructor din baza de date, nu s-a returnat si lista sa de cursuri, insa daca vom incerca sa il stergem, datorita la cascade delete(daca e setat), chiar si cu lazy fetch, lista sa cursuri va fi stearsa cu el. Asta e din cauza ca hibernate va cere acea lista, si deci ea va fi returnata la cerere. Fara Cascade, vom primi eroare.**  
**Ordinea de salvare**

* In lipsa lui cascade, ordinea in care salvam 2 obiecte ce sunt legate intre ele prin foreign key conteaza. Astfel, daca Curs are un foreign key catre Teacher si intai salvam obiectul Curs si apoi Teacher, vom primi eroare:

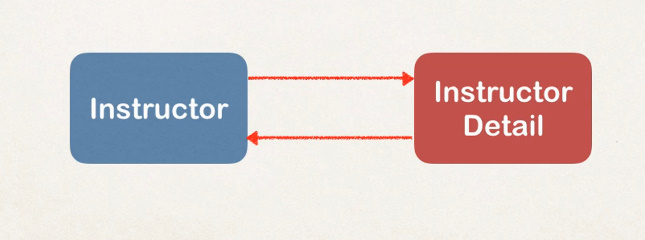


Asta e din cauza ca in course se gaseste un teacher, dar la momentul salvarii la curs, acel teacher inca nu exista in baza de date, si nu poate fi pus automat fara Cascade. Cu cascade, ordinea deja nu conteaza, asa cum oricum salvam un singur obiect.

**OneToOne Bi-Directional**

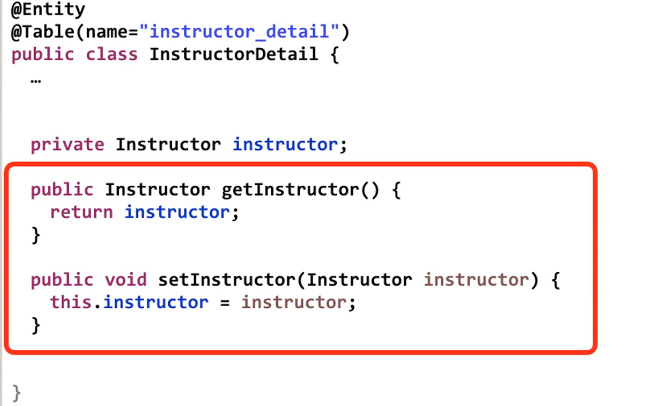
Mai sus am avut uni-directional mapping intre Instructor si InstructorDetail. Adica, de la Instructor puteam obtine InstructorDetail

* Acum, daca am vrea sa incarcam un InstructorDetail si sa obtinem pe baza lui un Instructor asociat, nu mai putem face asta. Avem nevoie deja de Bi-Directional mapping pentru a o putea face. **Unidirectional inseamna doar intr-o directie.**
* Asta si inseamna bi-directional, putem incepe cu primul ca sa il obtinem pe al doilea, si putem incepe cu al doilea ca sa il obtinem pe primul.

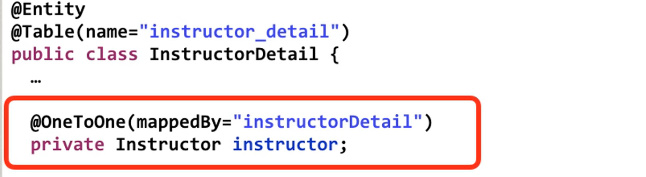


* Bi-Directional mapping nu necesita sa facem modificari la schema bazei de date, adica nu trebuie sa adaugam un foreign key in InstructorDetail, sau mai bine zis sa setam primary key ca foreign. Modificam doar codul Java.
* E necesar sa facem doar modificari in InstructorDetail class.

1. Cream in InstructorDetail o referinta la Instructor entity
2. Adaugam anotatia @OneToOne(mappedBy=”instructorDetail”)
3. Ne siguram sa crem getteri si seterii pentru el, exact cum facem si la Instructor.



Si adaugam:

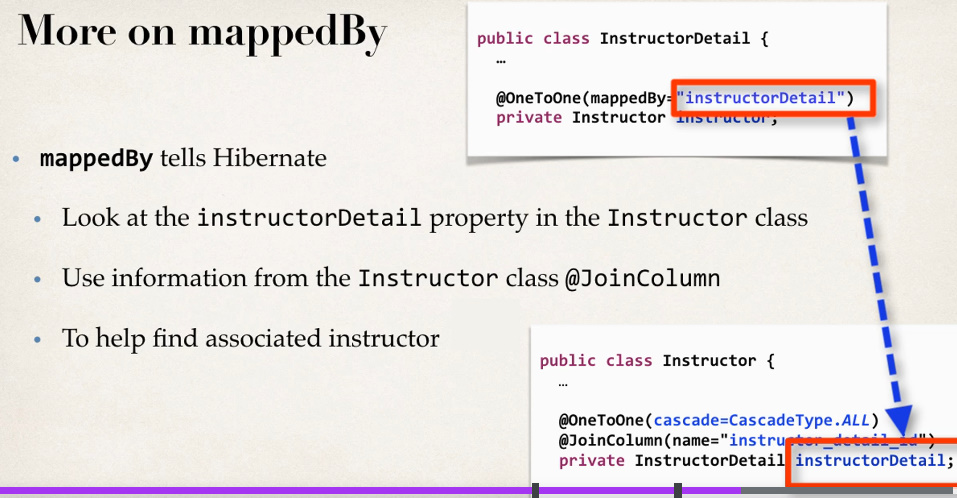


**@OneToOne(mappedBy=”instructorDetail”)**

mappedBy ii spune lui Hibernate ca noi ne referim la obiectul cu numele instructorDetail din Instructor class, adica la obiectul instructorDetail din clasa Instructor si ca el trebuie sa se uite acolo. Asa, el va avea o referinta la obiectul instructor, care il contine pe acest instructorDetail in el. Un fel de recursie, fiecare are referinta la altul.

Instructor ar putea avea mai multe foreign key, si Hibernate trebuie cumva sa stie care e foreign key a lui instructorDetail in tabel ca sa se uite, si asa se va uita la cel ce are instructorDetail ca object cu acest nume, si deci va sti si foreign\_key din tabel pe baza lui @JoinColumn si asa va alege Instructorul potrivit din tabel, adica cel ce are la atributul instructor\_detail\_id, idul corespunzator.

Cand Hibernate va crea obiectul de tip InstructorDetail, el il va crea si pe cel de tip Instructor, si va avea o referinta la el, insa si in obiectul de tip Instructor, el va cauta un field ce e de tip InstructorDetail cu numele instructorDetail ca sa-i dea referinta la obiectul de tip InstructorDetail, ca sa il poata accesa, sa fie dublu accesare



1. **Daca Hibernate il creaza mai intai pe InstructorDetail, el va recupera datele despre el, apoi va crea un obiect Instructor, se va duce la @JoinColumn, ii va da referinta lui instructorDetail, si se va uita la @JoinColumn sa vada care e foreign key, si foreign key e instructor\_detail\_id, deci va cauta id a lui InstructorDetail in foreign key de la Instructor, si odata ce o va gasi, va umple obiectul Instructor cu date.**
2. **Daca Hibernate il creaza intai pe Instructor, el recupereza datele lui si apoi creaza obiectul instructorDetail, dupa care se va uita la primary key din InstructorDetail pentru a gasi obiectul al carui id din primary key coincide cu foreign key din Instructor.**

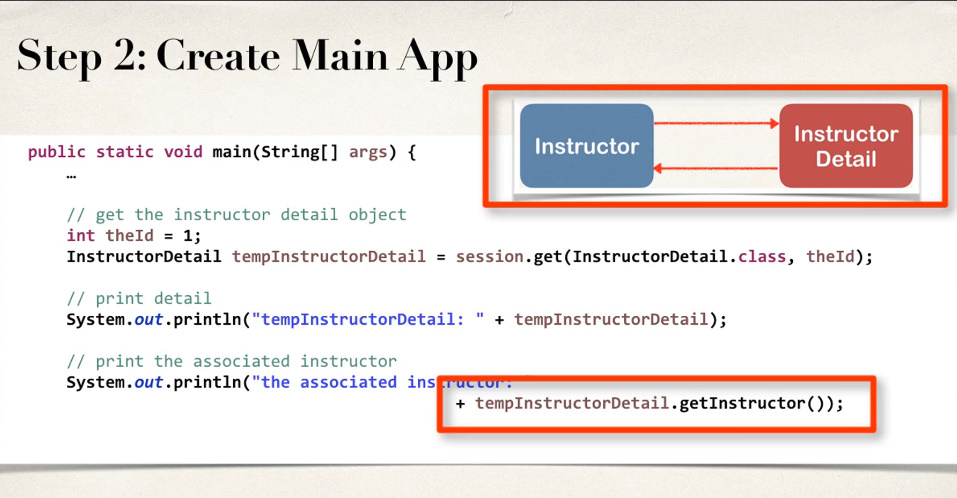
**Hibernate mereu creaza ambele obiecte! Unul in celalalt! De aceea putem sa obtinem InstructorDetail din Instructor:**

Instructor instructor = session.get(Instructor.class,1);  
  
 System.*out*.println(instructor.getInstructorDetail());  
session.getTransaction().commit();

* Putem adauga si aici cascade = cascadeType.ALL;

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

Aplicatia va fi asa:



**De la instructorDetail il obtinem pe instructor, si acel theId este anume Id al intructorDetail, nu Instructor! Asa ne va returna deci instructorul ce are ca instructor\_detail\_id theId**

int theId = 1;  
  
InstructorDetail instructorDetail = session.get(InstructorDetail.class, theId);  
System.*out*.println(instructorDetail);  
Instructor instructor = instructorDetail.getInstructor();  
System.*out*.println(instructor);

**mappedBy punem doar la entitatea copil1**

@OneToOne(mappedBy = "instructorDetail"

**Refactoring and Exceptions**

Daca de exemplu, apare vreo exceptie pana la inchiderea sesiunii, avem problema ca sesiunea nu se inchide si programul da crash. De aceea, trebuie sa cream mereu exceptii.

**OneToOne bi-Directional Cascade Delete**

Stergerea este identica ca si in cazul la unu-directional,si daca avem la ambele entitati:

@OneToOne(cascade = CascadeType.*ALL*)

Atunci stergerea unuia implica si stergerea al doilea.

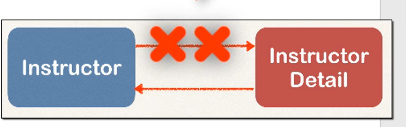
Daca dorim sa stergem doar un obiect, modificam cascade:

**Cascade={CascadeType.MERGE, CascadeType.Detach, CascadeType.PERSIST,CascadeType.REFRESH}**

Putem pune mai multe in {} si le separam prin , **insa acest lucru nu e suficient**, deoarece intre ele oricum inca exista o legatura bidirectionala care le face sa fie unite unul de celalalt reciproc.Asa, cand Instructor Detail e sters, Instructor va continua sa pastreze referinta catre un obiect inexistent.De ex, daca vrem sa rupem legatura dine Instructor si InstructorDetail, dar sa o pastram pe cea dintre InstructorDetail si Instructor, facem asa:

instructorDetail.getInstructor().setInstructorDetail(null);





Un entity poate fi stearsa doar atat timp cat nu e copilul cuiva.

Pentru a readauga ce am sters din tabelul asociat:

InstructorDetail instructorDetail = new InstructorDetail("AlfaLight","Programming");  
  
Instructor instructor = session.get(Instructor.class,theId);  
  
instructor.setInstructorDetail(instructorDetail);