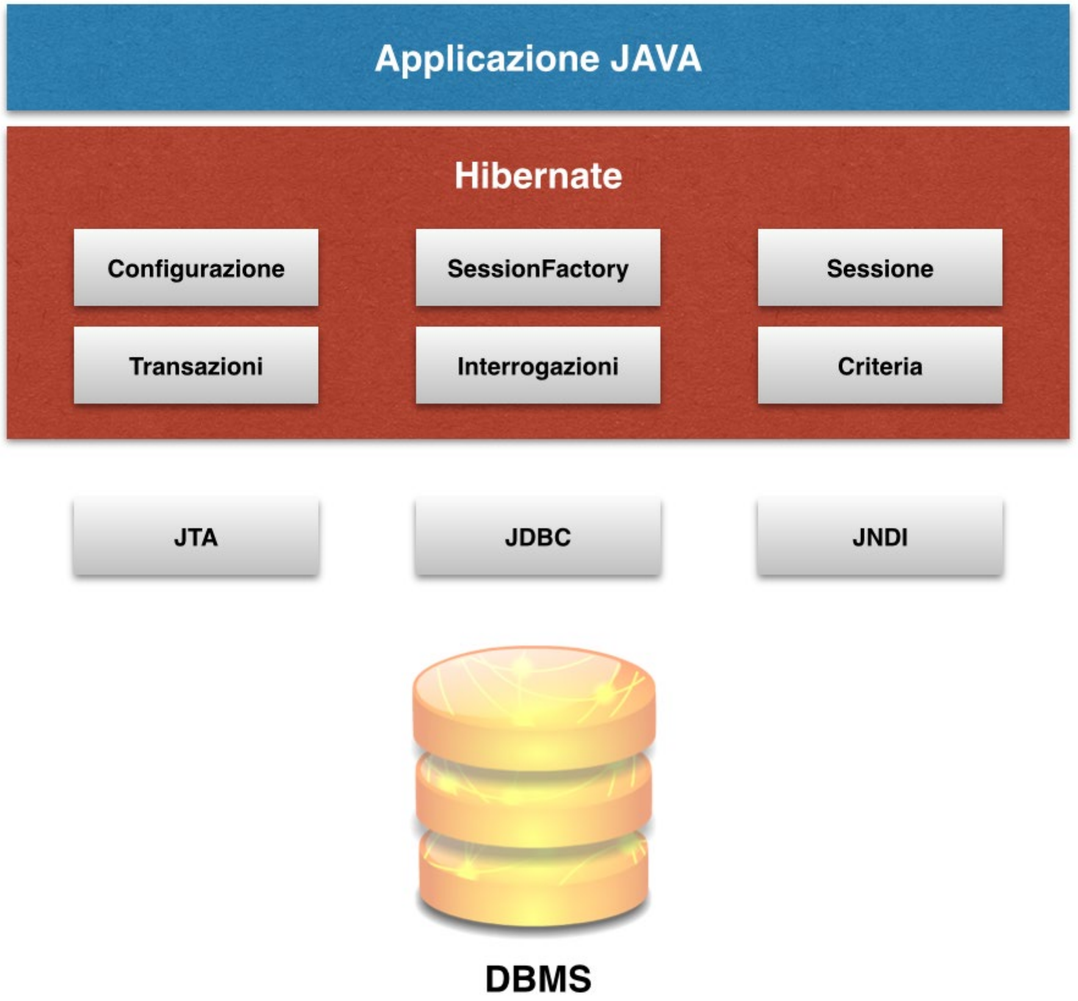
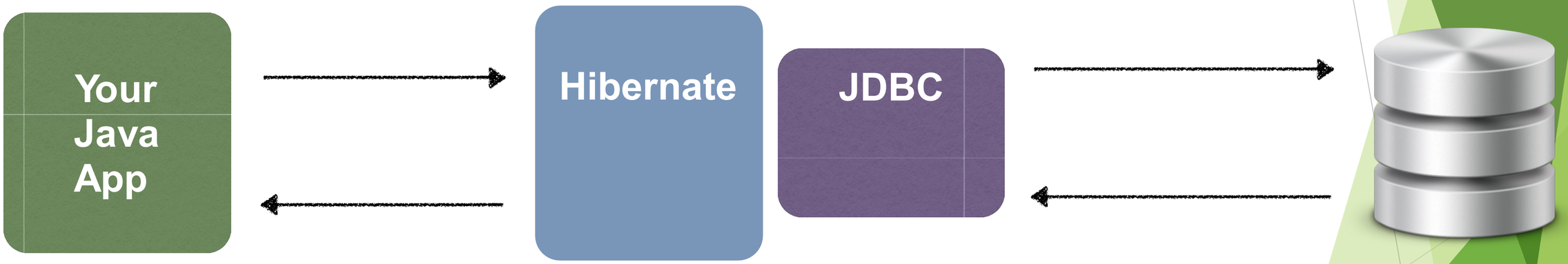
SPRING DATA

Cos'è Hibernate

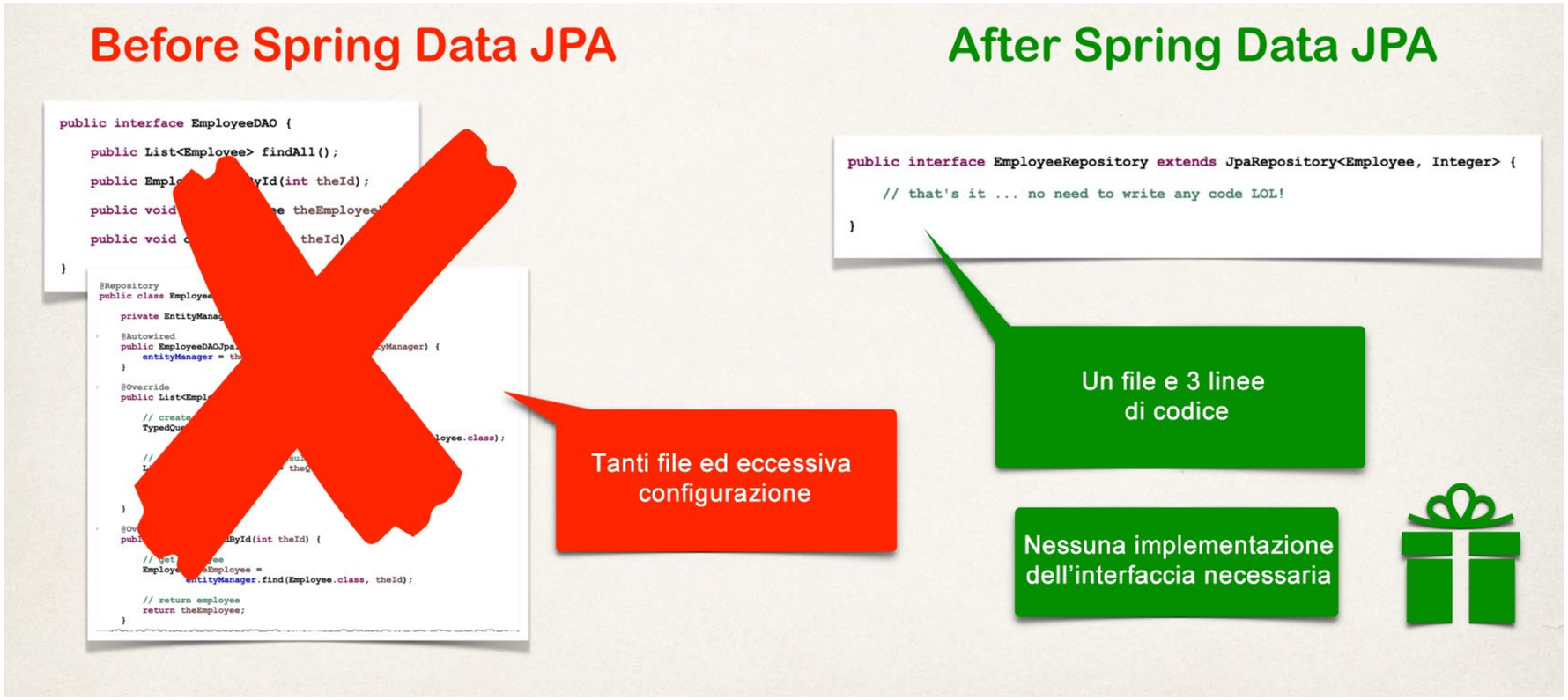


Hibernate e JDBC

- Hibernate utilizza JDBC per tutte le comunicazioni con il database



Con Spring Data



Quali dipendenze Maven?

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Terminologia

- Primary Key
- Identifica in modo univoco ogni riga di una tabella
- Deve essere un valore univoco
- Non può contenere valori NULL

MySQL - Auto Increment

```
CREATE TABLE student (

   id int(11) NOT NULL AUTO_INCREMENT,
   first_name varchar(45) DEFAULT NULL,
   last_name varchar(45) DEFAULT NULL,
   email varchar(45) DEFAULT NULL,
   PRIMARY KEY (id)

)
```

Hibernate Identity - Primary Key

```
@Entity
@Table(name="student")
public class Student {

  @Id
  @Column(name="id")
  private int id;


 …
}
```

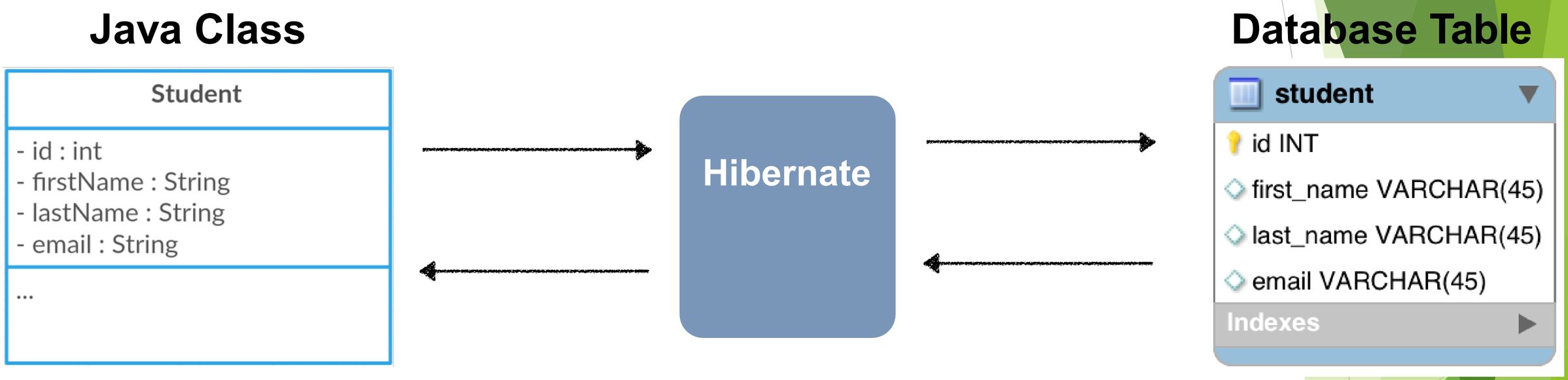
Hibernate Identity - Primary Key

```
@Entity
@Table(name="student")
public class Student {

  @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="id")
  private int id;


 …
}
```

Strategie di generazione degli ID

GenerationType.AUTO	Sceglie una strategia appropriata per il database specifico
GenerationType.IDENTITY	Assegna chiavi primarie utilizzando la colonna identità del database
GenerationType.SEQUENCE	Assegnare chiavi primarie utilizzando una sequenza interna al database
GenerationType.TABLE	Assegnare chiavi primarie utilizzando una tabella di database sottostante per garantire l'univocità

Mapping di base



Mappings avanzati

- Nel database, molto probabilmente avrai:

- Più tabelle
- Relazioni tra tabelle

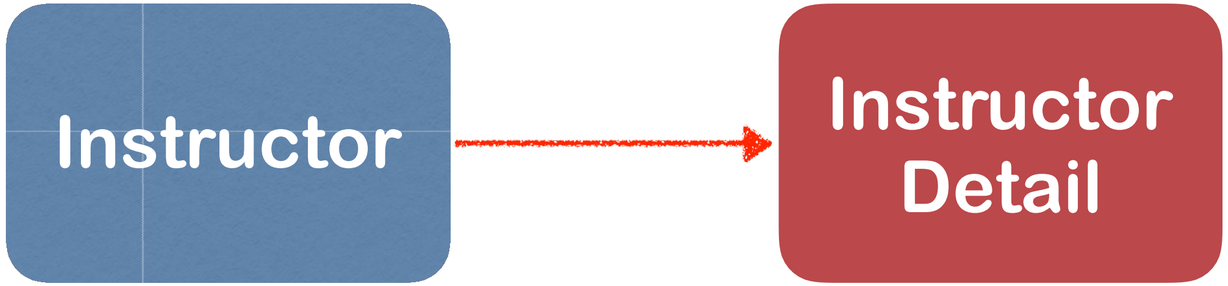
- È necessario modellare più tabelle e le rispettive relazioni attraverso Hibernate e Spring Data

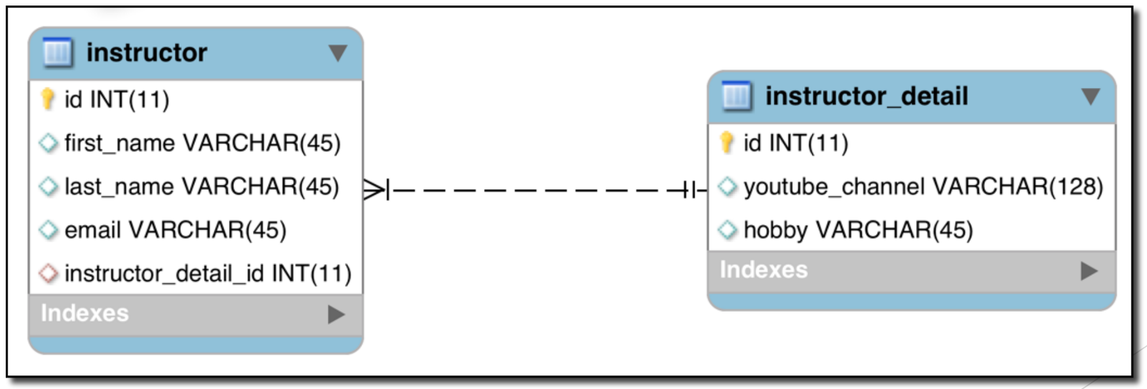
Mappings avanzati

- One-to-One
- One-to-Many, Many-to-One
- Many-to-Many

One-to-One Mapping

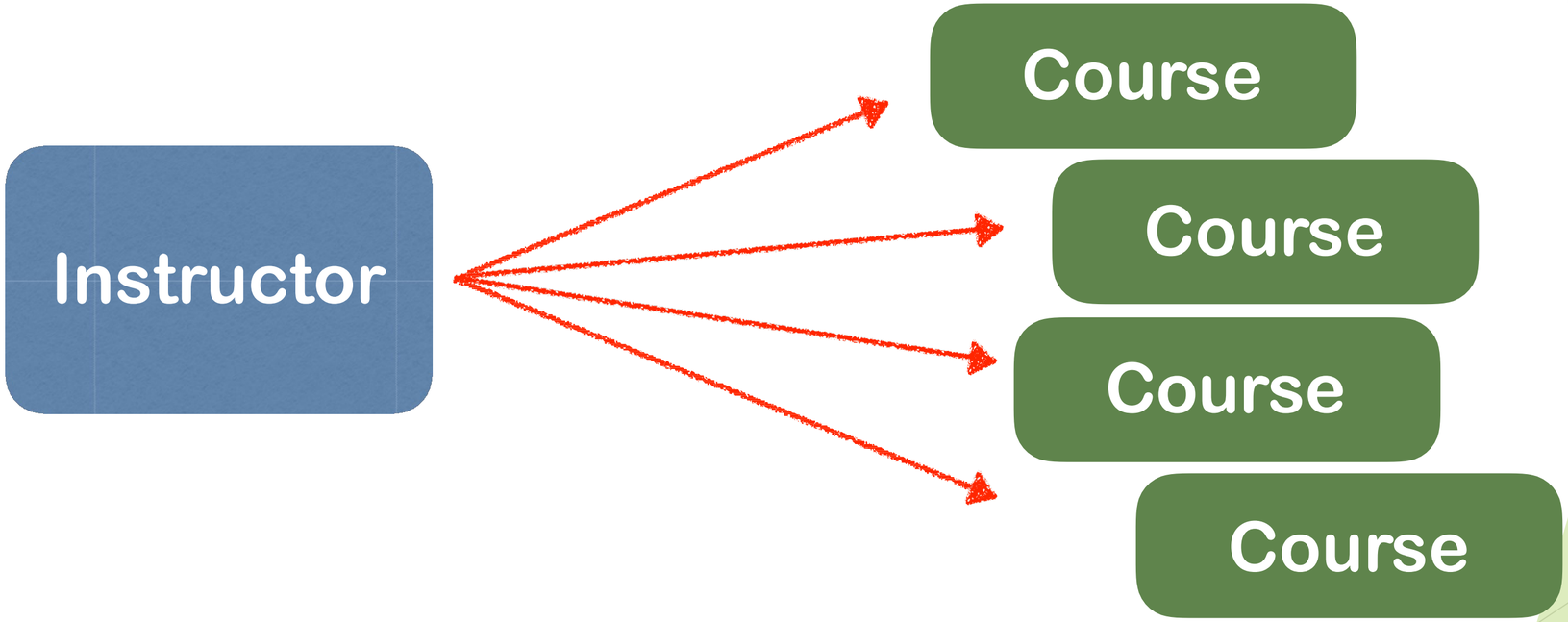
- Un istruttore può avere un'entità "dettaglio istruttore"
- Simile a un "profilo istruttore"





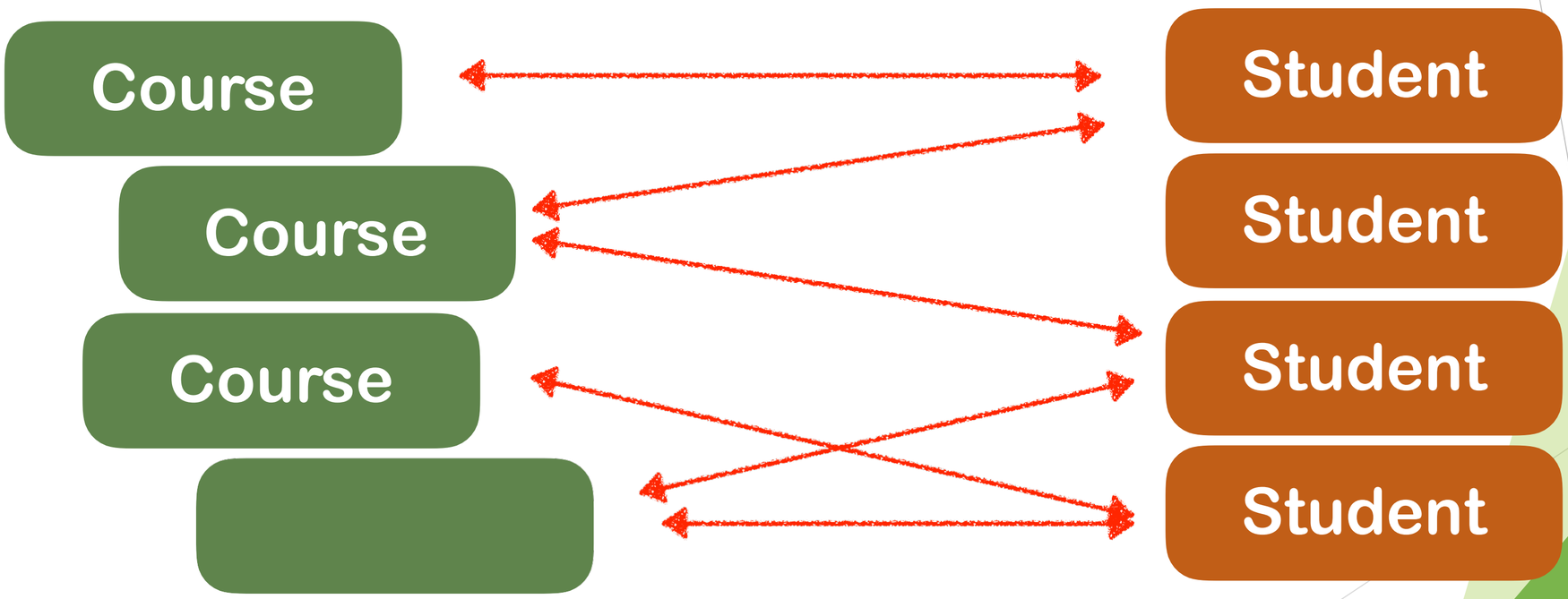
One-to-Many Mapping

- Un istruttore può avere molti corsi



Many-to-Many Mapping

- Un corso può avere molti studenti
- Uno studente può avere molti corsi



Concetti importanti nel contesto database

- Primary key e foreign key
- Cascade

Primary Key e Foreign Key

- Primary key: identifica una riga univoca in una tabella
- Foreign key:
  - Collega le tabelle
  - E’ un campo di una tabella che fa riferimento alla chiave primaria in un'altra tabella

Esempio di Foreign Key

Foreign key column

Table: instructor

id	first_name	last_name	instructor_detail_id
1	Tizio	Bello	100
2	Tizio	Brutto	200

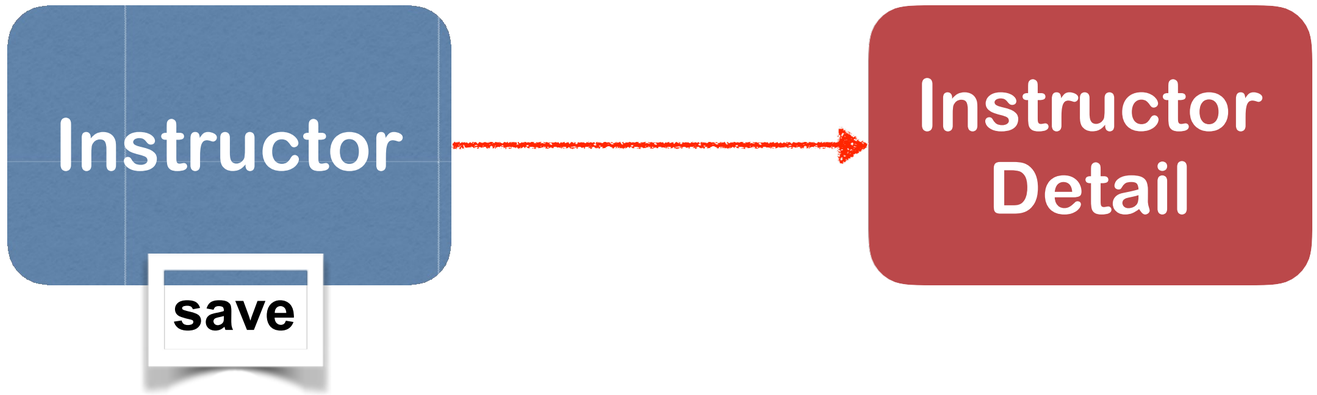
Table: instructor_detail

id	youtube_channel	hobby
100	www.youtube.com/test	Test
200	www.youtube.com	Guitar

Cascade

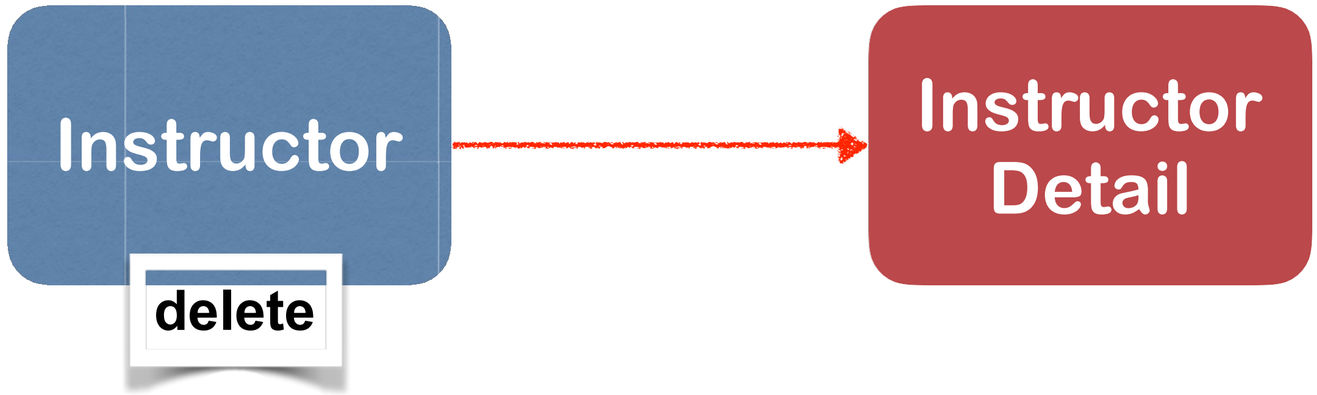
- È possibile eseguire operazioni a cascata (**cascade**)

- Applicare la stessa operazione alle entità correlate



Cascade

- Se eliminiamo un istruttore, dovremmo anche eliminare il suo corrispondente instructor_detail
- Questo è noto come "CASCADE DELETE"



Cascade Delete

Foreign key column

Table: instructor

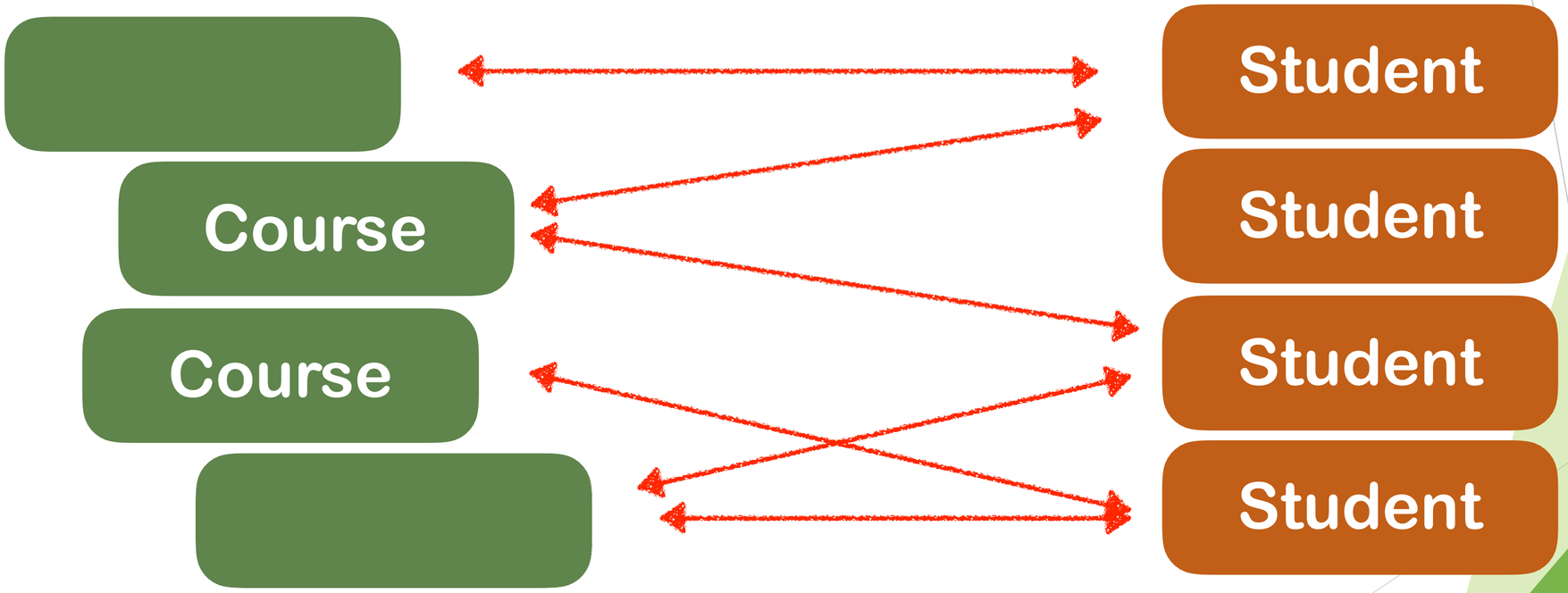
id	first_name	last_name	instructor_detail_id
1	Tizio	Bello	100
2	Tizio	Brutto	200

Table: instructor_detail

id	youtube_channel	hobby
100	www.youtube.com/test	Test
200	www.youtube.com	Guitar

Cascade Delete

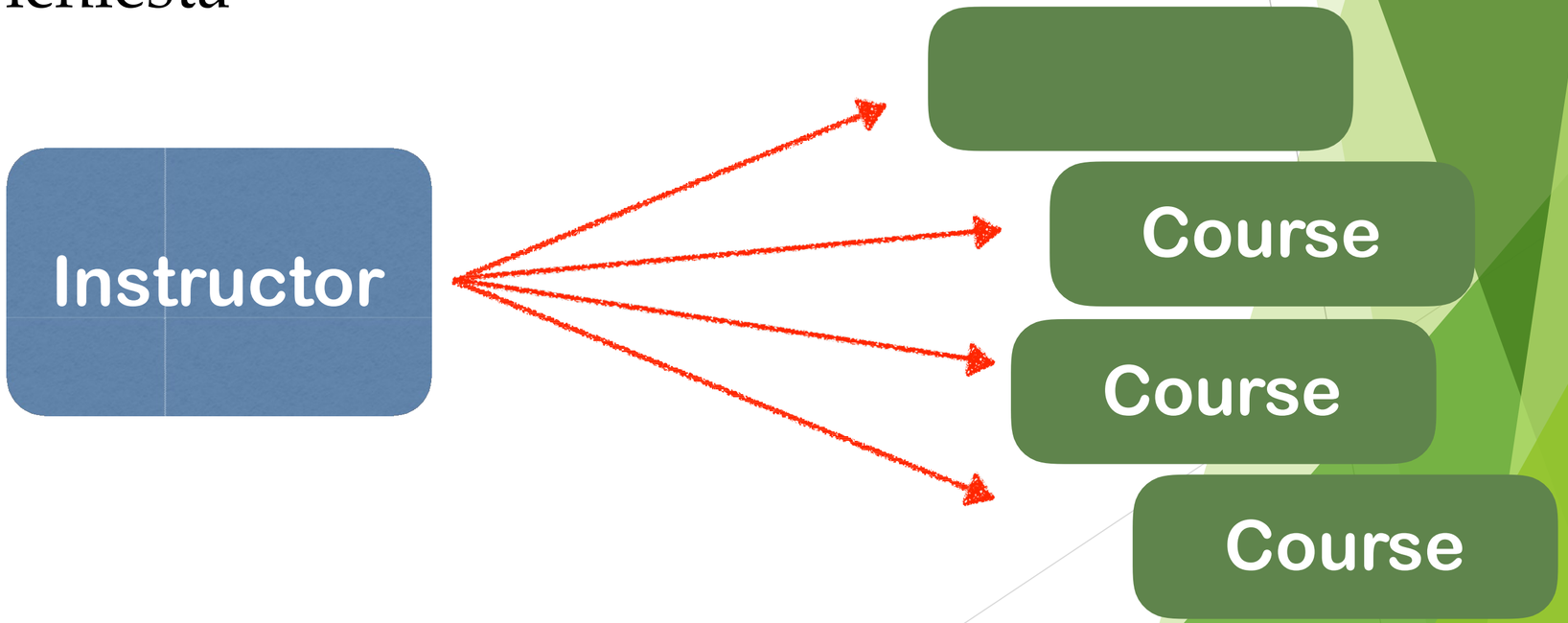
- L'eliminazione a cascata dipende dal caso d'uso
- Dovremmo eliminare a cascata in questo cas?



Fetch Types: Eager vs Lazy Loading

- Quando recuperiamo i dati, dovremmo recuperare TUTTO (quindi anche le relazioni)?

- **Eager** recupera tutto
- **Lazy** recupera su richiesta



One-to-one unidirezionale

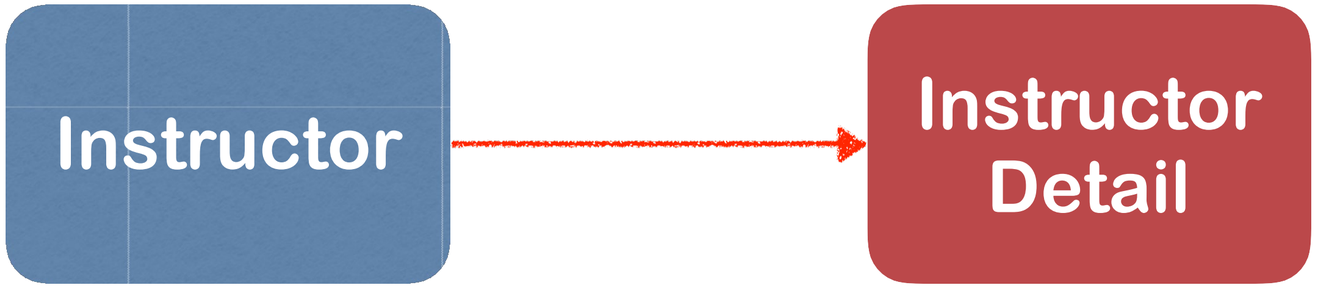


table: instructor_detail

File: create-db.sql

```
CREATE TABLE `instructor_detail` (

  `id`              int(11)      NOT NULL AUTO_INCREMENT,
  `youtube_channel` varchar(128) DEFAULT NULL,
  `hobby`           varchar(45)  DEFAULT NULL,

  PRIMARY KEY (`id`)
);

...
```

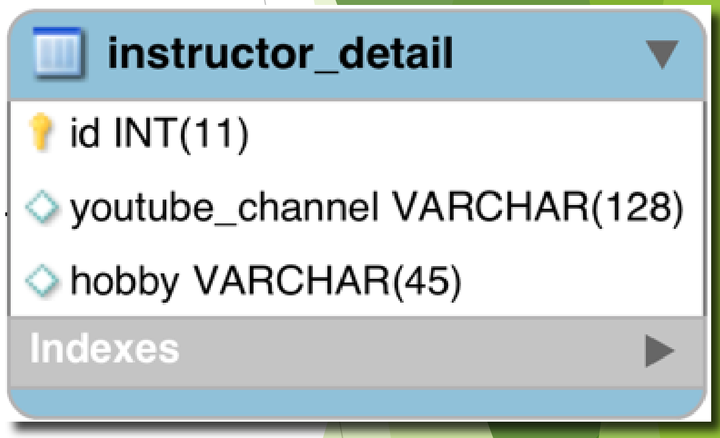
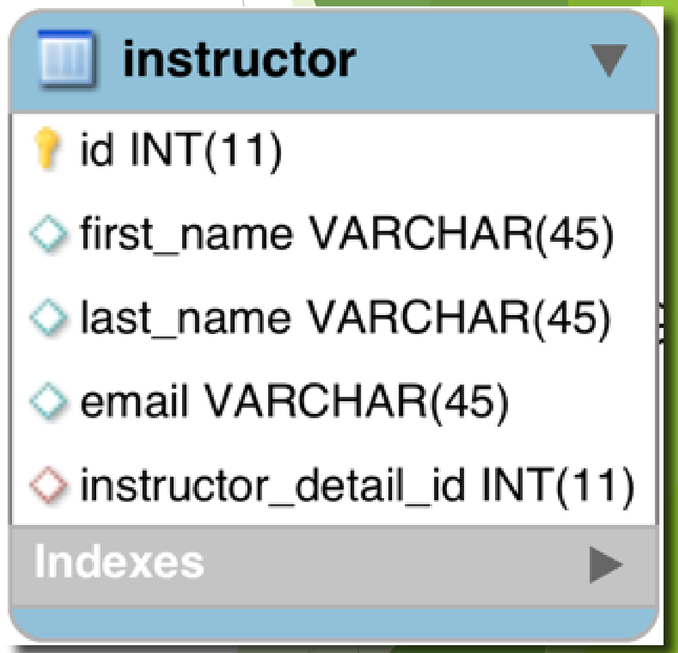


table: instructor

File: create-db.sql

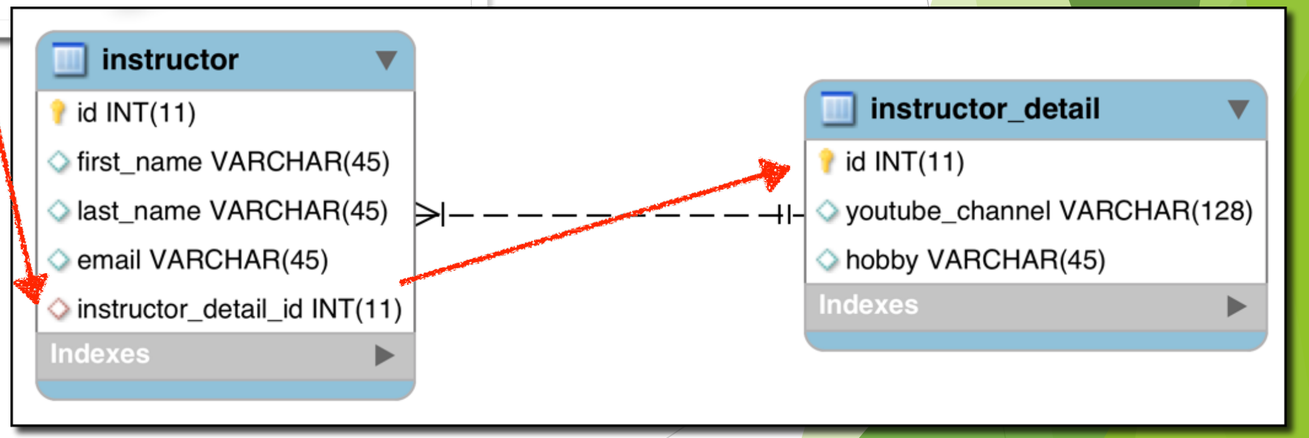
```
...

CREATE TABLE `instructor` (

  `id`          int(11) NOT NULL AUTO_INCREMENT,
  `first_name`  varchar(45) DEFAULT NULL,
  `last_name`   varchar(45) DEFAULT NULL,
  `email`       varchar(45) DEFAULT NULL,
  `instructor_detail_id` int(11) DEFAULT NULL,

  PRIMARY KEY (`id`)
    ...
);
```



Definizione Foreign Key

File: create-db.sql

```
...
CREATE TABLE `instructor` (
  ...

  CONSTRAINT `FK_DETAIL` FOREIGN KEY (`instructor_detail_id`)
  REFERENCES `instructor_detail` (`id`)

);
```



Maggiori informazioni su Foreign Key

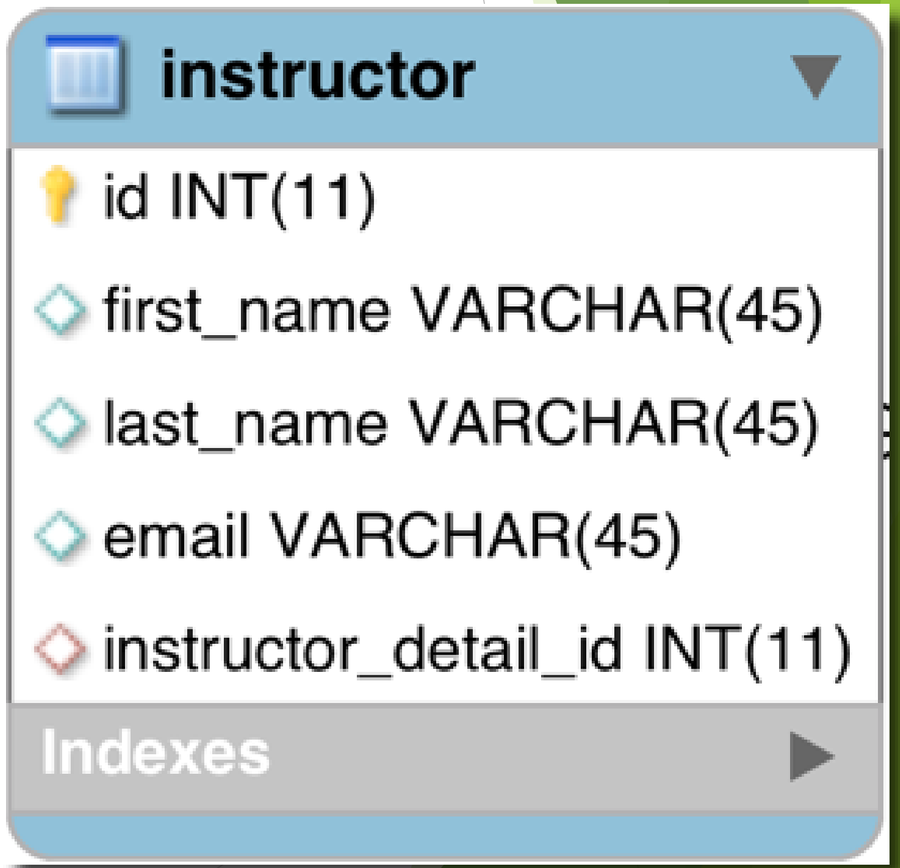
- Lo scopo principale è quello di preservare la relazione tra le tabelle
- Integrità referenziale

- Impedisce operazioni che distruggerebbero la relazione

- Assicura che nella colonna della chiave esterna vengono inseriti solo dati validi
  - Può contenere solo un riferimento valido alla chiave primaria in un'altra tabella

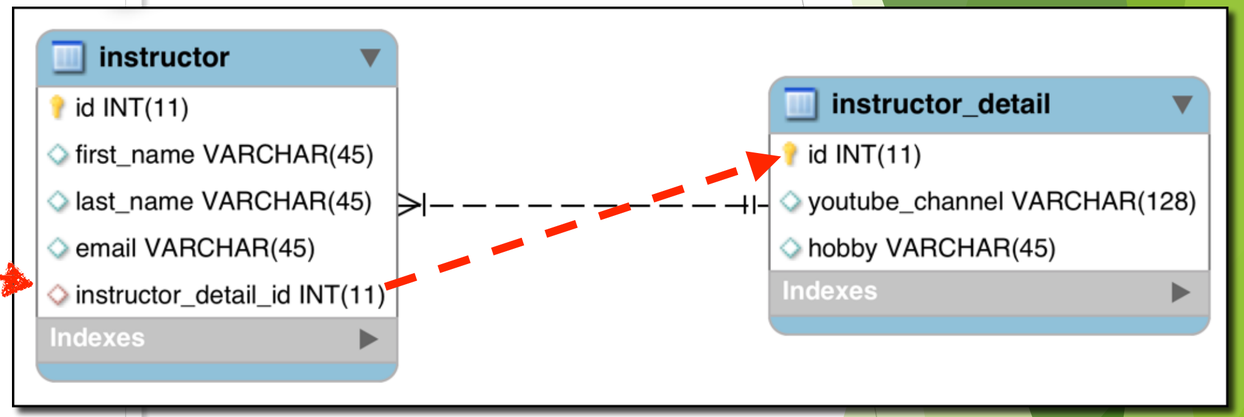
Creare classe Instructor

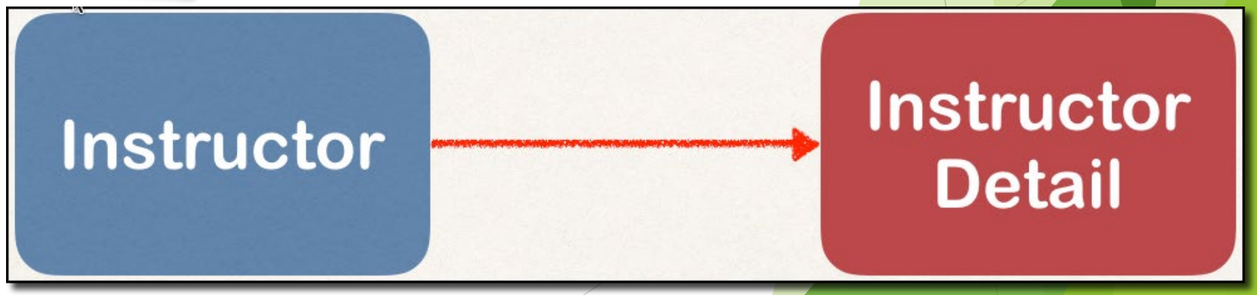
```java
@Entity
@Table(name="instructor")
public class Instructor {

    @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;

    …
    // constructors, getters / setters
}
```



Creare class Instructor - @OneToOne

```
@Entity
@Table(name="instructor")
public class Instructor {
    …

    @OneToOne
    @JoinColumn(name="instructor_detail_id")
    private InstructorDetail instructorDetail;

    …
    // constructors, getters / setters
}
```





Configurare Cascade Type


```
@Entity  @Table(name="instructor")  public class Instructor {

...

@OneToOne(cascade=CascadeType.ALL)  @JoinColumn(name="instructor_detail_id")
private InstructorDetail instructorDetail;

...

// constructors, getters / setters
```

Di default, nessuna cascade è definita

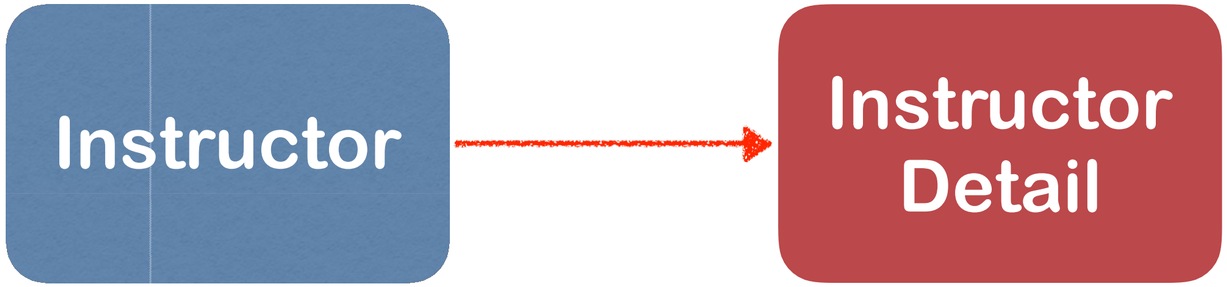
Cascade Types multiple

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

Nuovo caso d'uso

- Se carichiamo un InstructorDetail
- Poi potrebbe essere necessario richiedere l'Instructor associato

- Non è possibile farlo con l'attuale relazione unidirezionale :-(



One-to-one bidirezionale

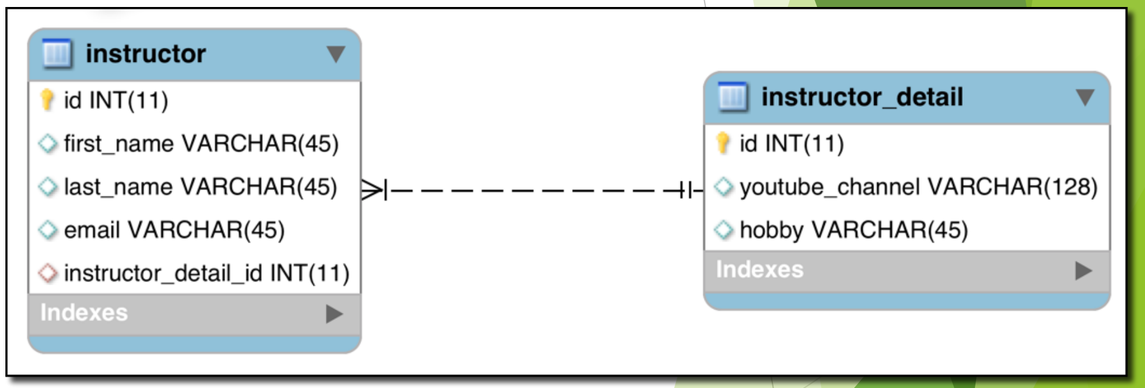
- La relazione bidirezionale è la soluzione

- Possiamo richiedere le informazioni su Instructor a partire da InstructorDetail

Bidirezionale - La buona notizia

- Per utilizzare la relazione bidirezionale, possiamo mantenere lo schema del database esistente
  Nessuna modifica necessaria per il database!

- Basta aggiornare il codice Java



Aggiungere un nuovo campo a cui fare riferimento

```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {
    ...

    private Instructor instructor;

    ...
}
```

Aggiungere metodi getter/setter Instructor

```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {
    ...


    private Instructor instructor;

    public Instructor getInstructor() {
        return instructor;
    }

    public void setInstructor(Instructor instructor) {
        this.instructor = instructor;
    }
    ...

}
```

Aggiungere @OneToOne

```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {
    ...

    @OneToOne(mappedBy="instructorDetail")
    private Instructor instructor;

    public Instructor getInstructor() {
        return instructor;
    }

    public void setInstructor(Instructor instructor) {
        this.instructor = instructor;
    }
    ...

}
```

Fa riferimento alla proprietà "instructorDetail" nella classe "Instructor"

Di più su mappedBy

```
public class InstructorDetail {
    ...

    @OneToOne(mappedBy="instructorDetail")
    private Instructor instructor;
```

- **mappedBy** dice ad Hibernate
  - Guarda la proprietà instructorDetail nella classe Instructor
  - Utilizza le informazioni della classe Instructor annotate con @JoinColumn per trovare l'istruttore associato

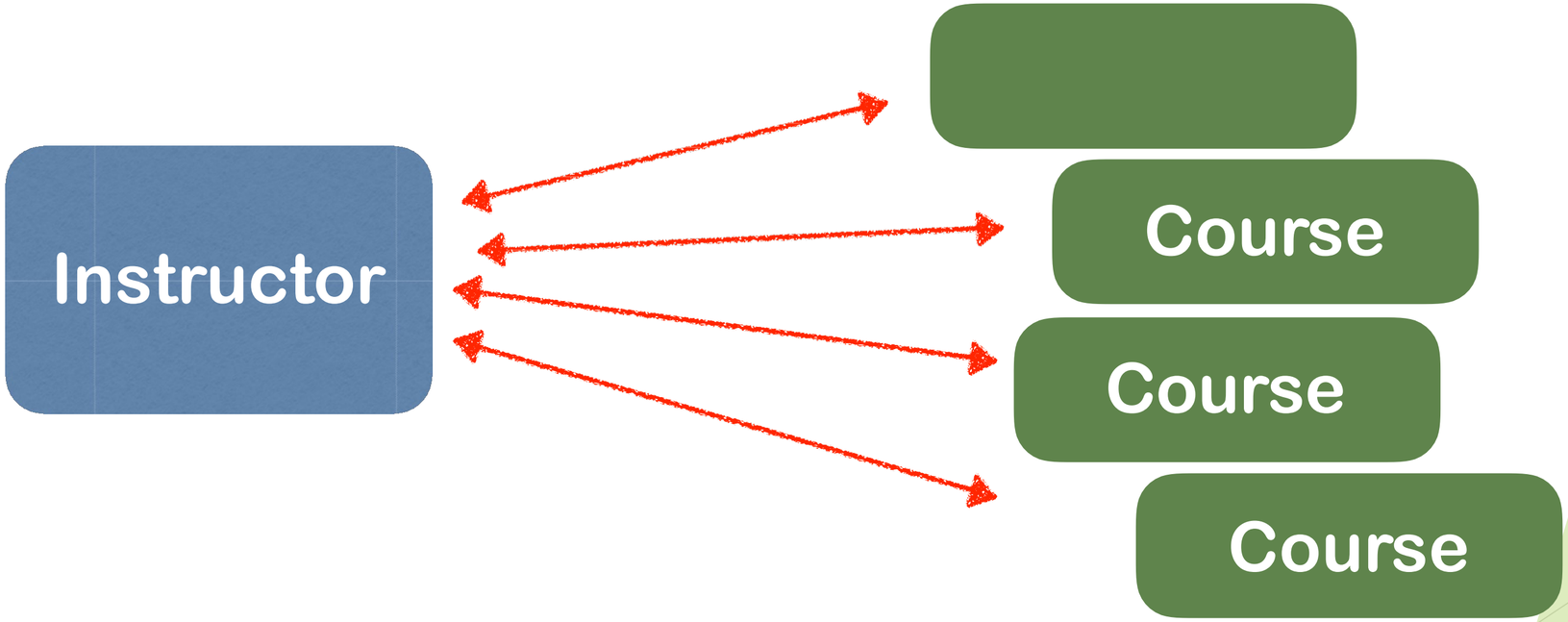
```
public class Instructor {
    ...

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="instructor_detail_id")
    private InstructorDetail instructorDetail;
```

Aggiungere supporto per Cascading

```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {
    ...

    @OneToOne(mappedBy="instructorDetail", cascade=CascadeType.ALL)
    private Instructor instructor;

    public Instructor getInstructor() {
        return instructor;
    }

    public void setInstructor(Instructor instructor) {
        this.instructor = instructor;
    }
    ...

}
```

One-to-Many Mapping

- Un istruttore può avere molti corsi
- Bidirezionale



Many-to-One Mapping

- Molti corsi possono avere un istruttore
- Inverso / opposto a One-to-Many

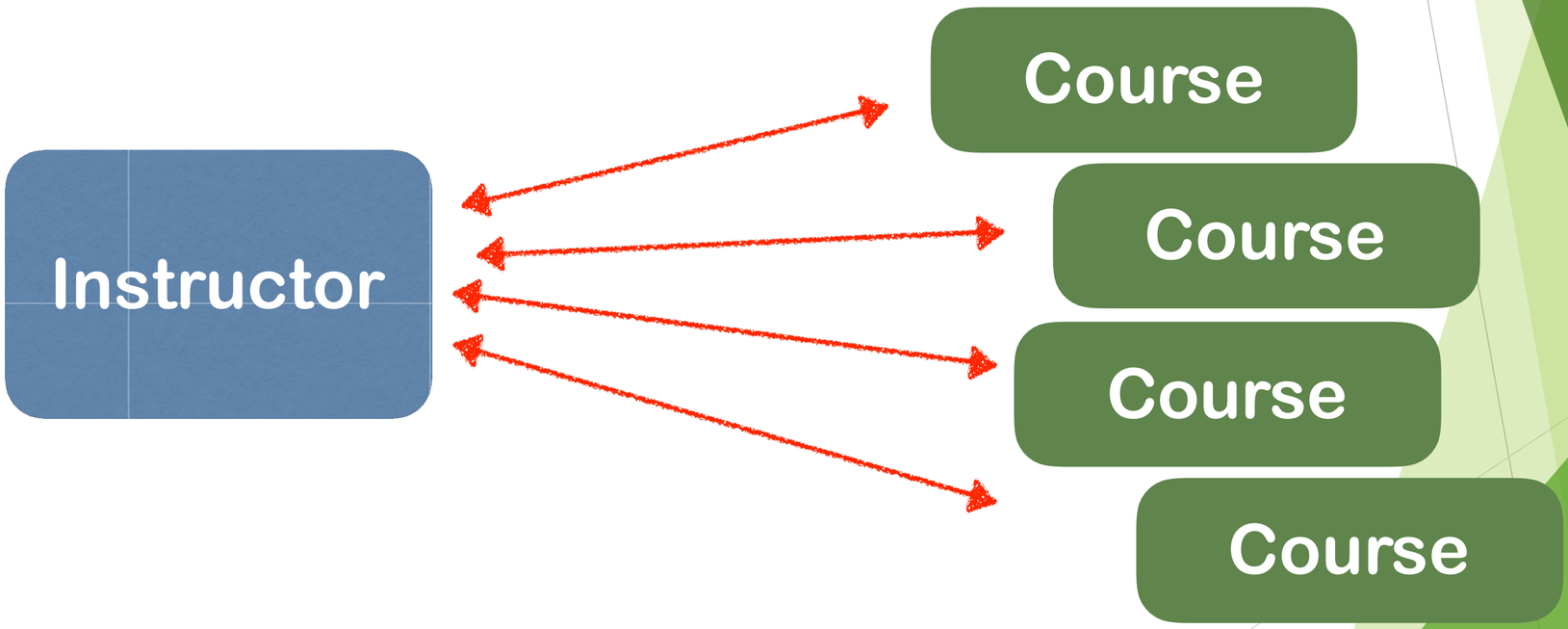
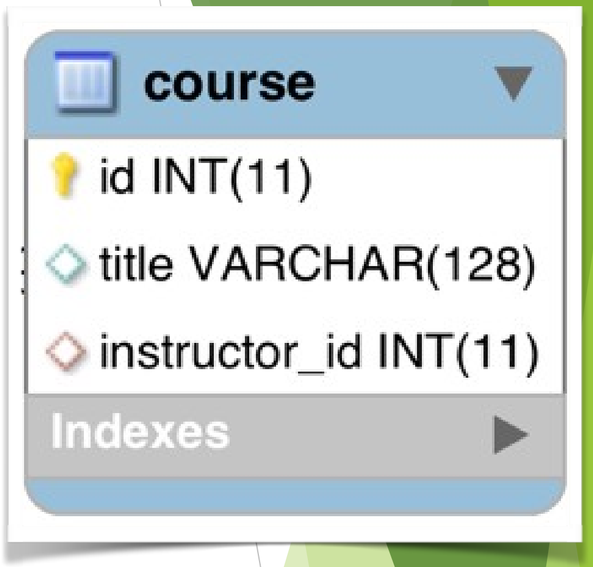


table: course

File: create-db.sql

```
CREATE TABLE `course` (

  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(128) DEFAULT NULL,
  `instructor_id` int(11) DEFAULT NULL,

  PRIMARY KEY (`id`),

  UNIQUE KEY `TITLE_UNIQUE` (`title`),
    ...
);
```



Prevent duplicate course titles

table: course - foreign key

File: create-db.sql

```
CREATE TABLE `course` (
  …
  KEY `FK_INSTRUCTOR_idx` (`instructor_id`),
  CONSTRAINT `FK_INSTRUCTOR`
  FOREIGN KEY (`instructor_id`)
  REFERENCES `instructor` (`id`)
  …
);
```

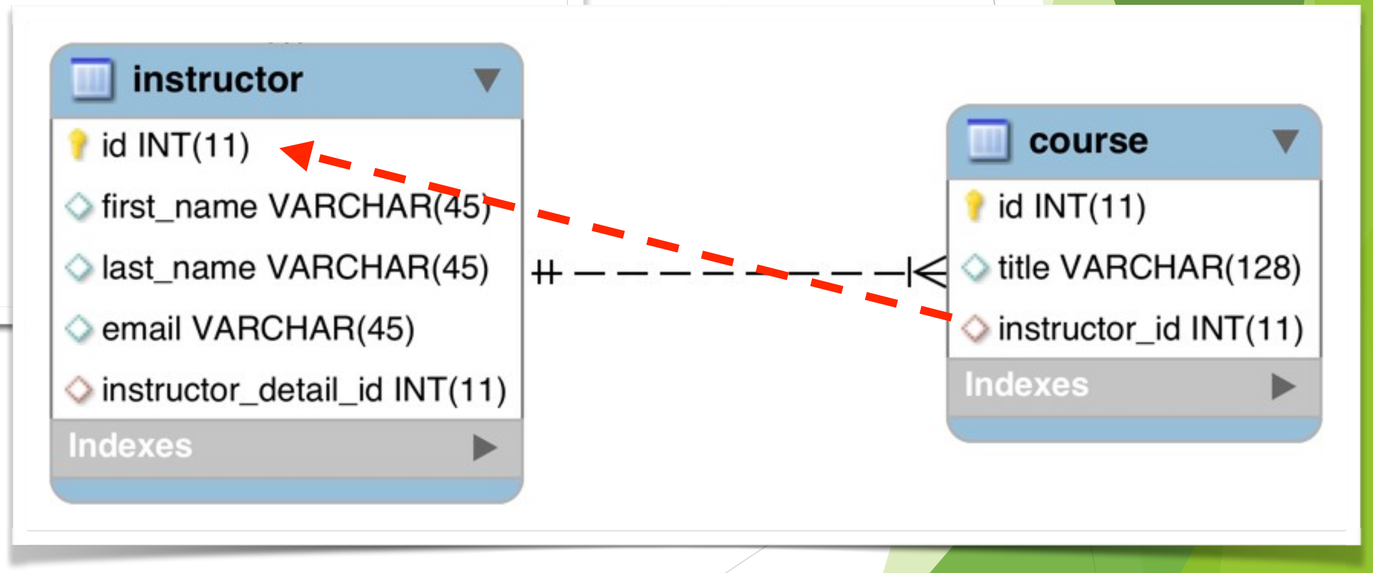
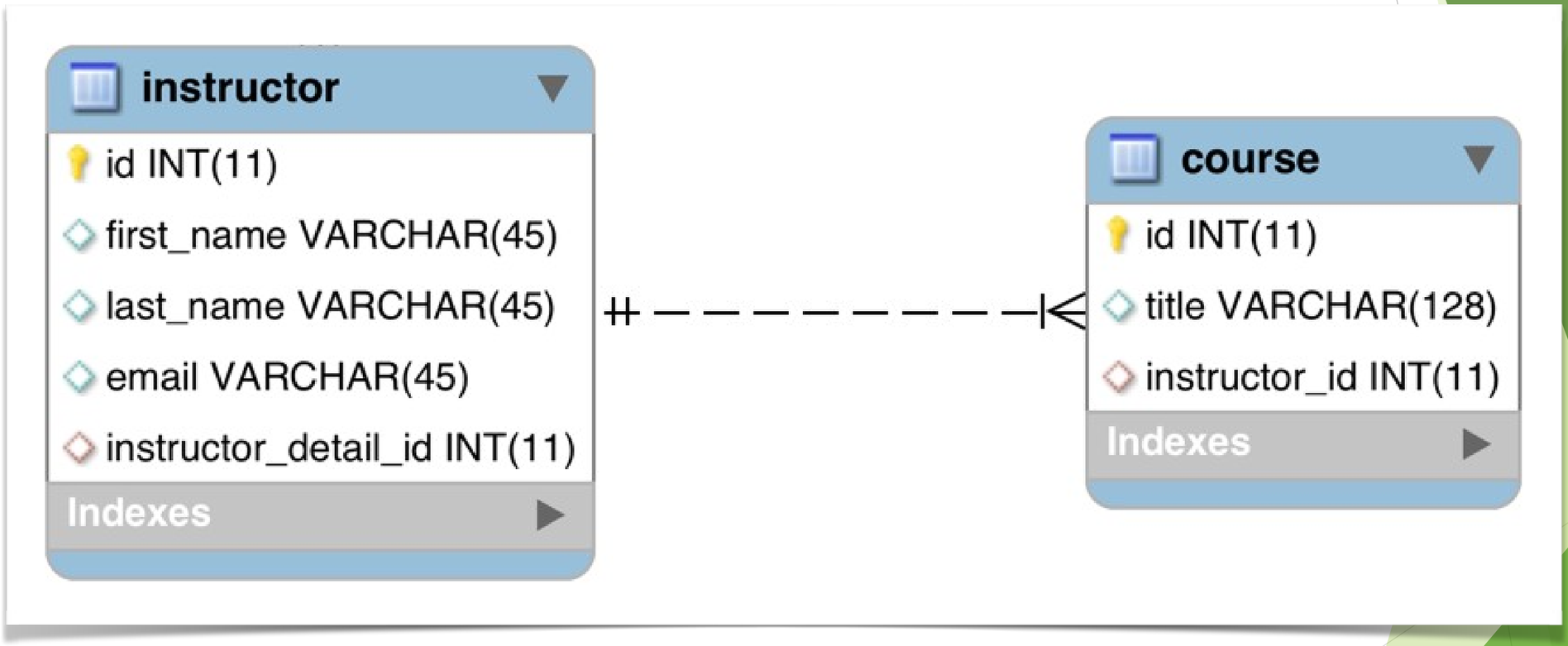


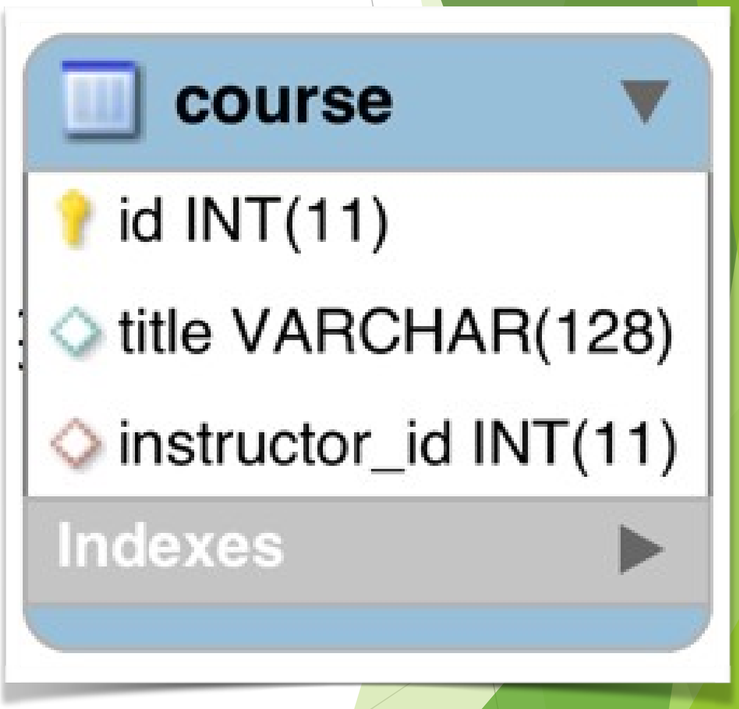
table: instructor

Nessun cambiamento



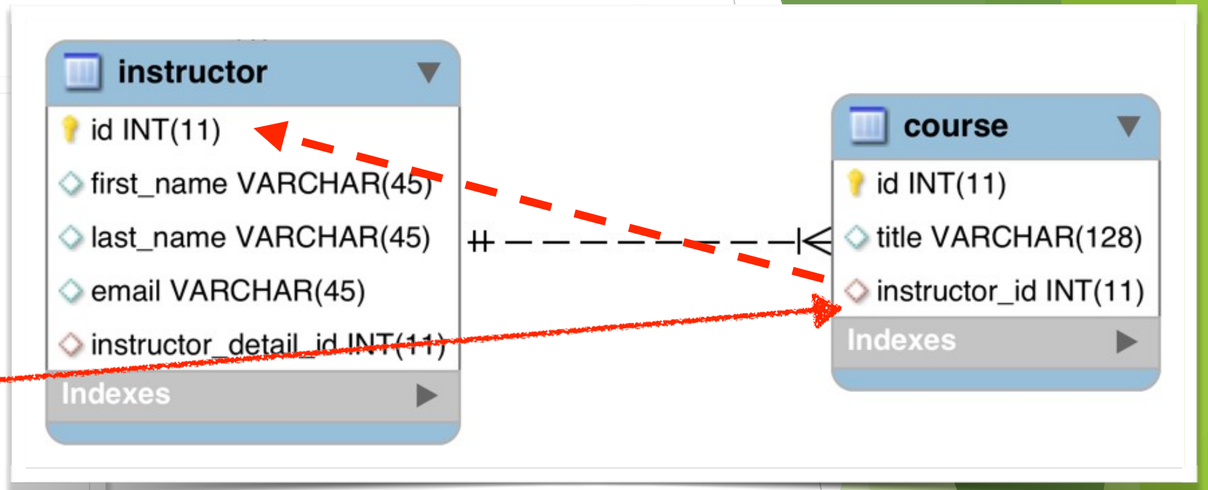
Creare class Course

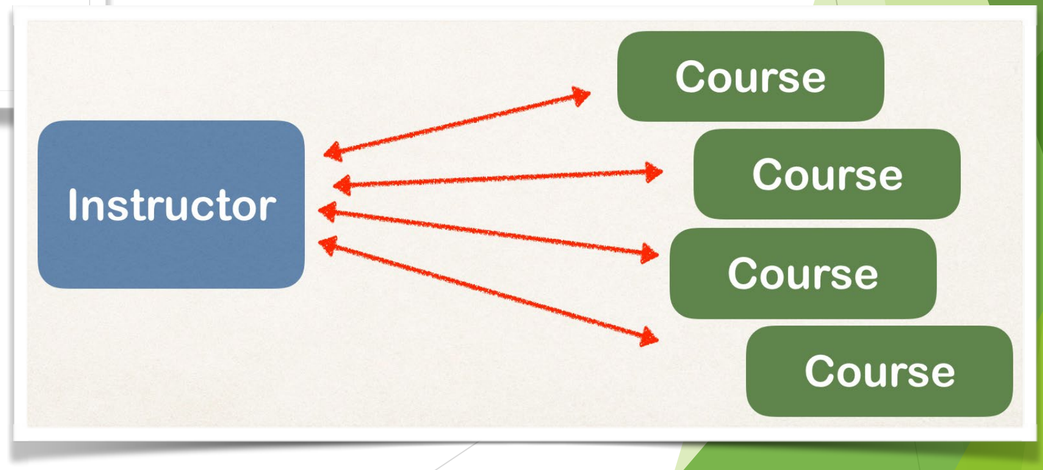
```
@Entity
@Table(name="course")
public class Course {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="title")
    private String title;


    …
    // constructors, getters / setters
}
```



Creare classe Course - @ManyToOne

```
@Entity
@Table(name="course")
public class Course {
    …

    @ManyToOne
    @JoinColumn(name="instructor_id")
    private Instructor instructor;

    …
    // constructors, getters / setters
}
```





Aggiornare Instructor

```
@Entity
@Table(name="instructor")
public class Instructor {
   ...

   private List<Course> courses;

   public List<Course> getCourses() {

      return courses;
   }

   public void setCourses(List<Course> courses) {
      this.courses = courses;
   }
   ...

}
```

Aggiungere @OneToMany

```
@Entity
@Table(name="instructor")
public class Instructor {
   ...

   @OneToMany(mappedBy="instructor")
   private List<Course> courses;

   public List<Course> getCourses() {
      return courses;
   }

   public void setCourses(List<Course> courses) {
      this.courses = courses;
   }

   ...

}
```

Si riferisce alla proprietà "instructor" nella classe "Course"

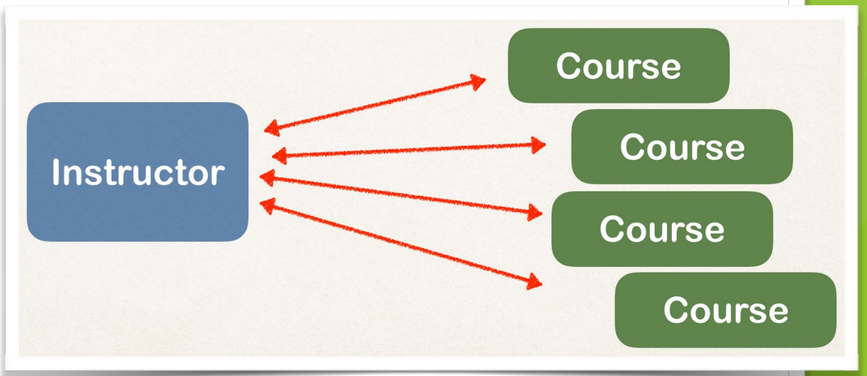
Aggiungere Cascading

No DELETE

```
@Entity
@Table(name="instructor")
public class Instructor {
    ...

    @OneToMany(mappedBy="instructor",
                    cascade={CascadeType.PERSIST, CascadeType.MERGE
                        CascadeType.DETACH, CascadeType.REFRESH})
    private List<Course> courses;



    ...

}
```

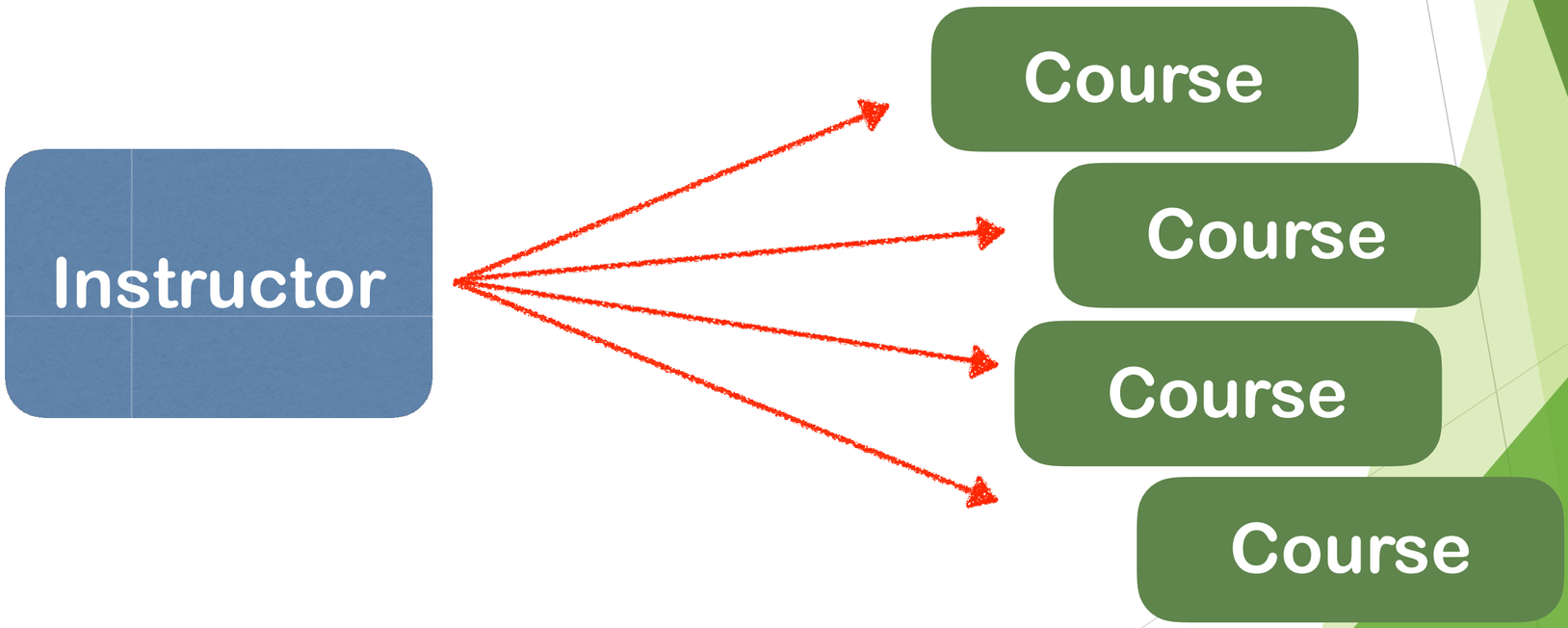
Aggiungere utilities per gestire bidirezionalità

```
@Entity
@Table(name="instructor")
public class Instructor {
    ...
    // add convenience methods for bi-directional relationship

    public void add(Course tempCourse) {

        if (courses == null) {
            courses = new ArrayList<>();
        }

        courses.add(tempCourse);

        tempCourse.setInstructor(this);
    }
    ...
}
```



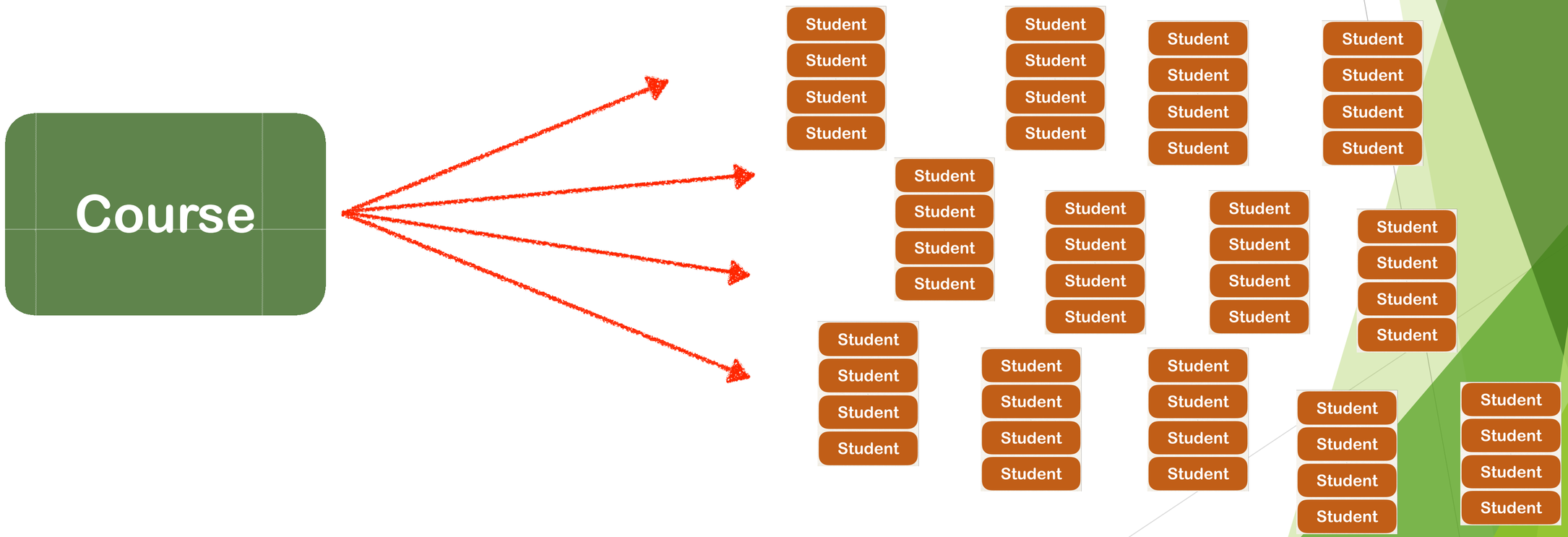
Eager Loading

- Il caricamento eager caricherà tutte le entità dipendenti
  - Carica l'istruttore e tutti i suoi corsi contemporaneamente



Eager Loading

Potrebbe facilmente trasformarsi in un incubo di prestazioni ...



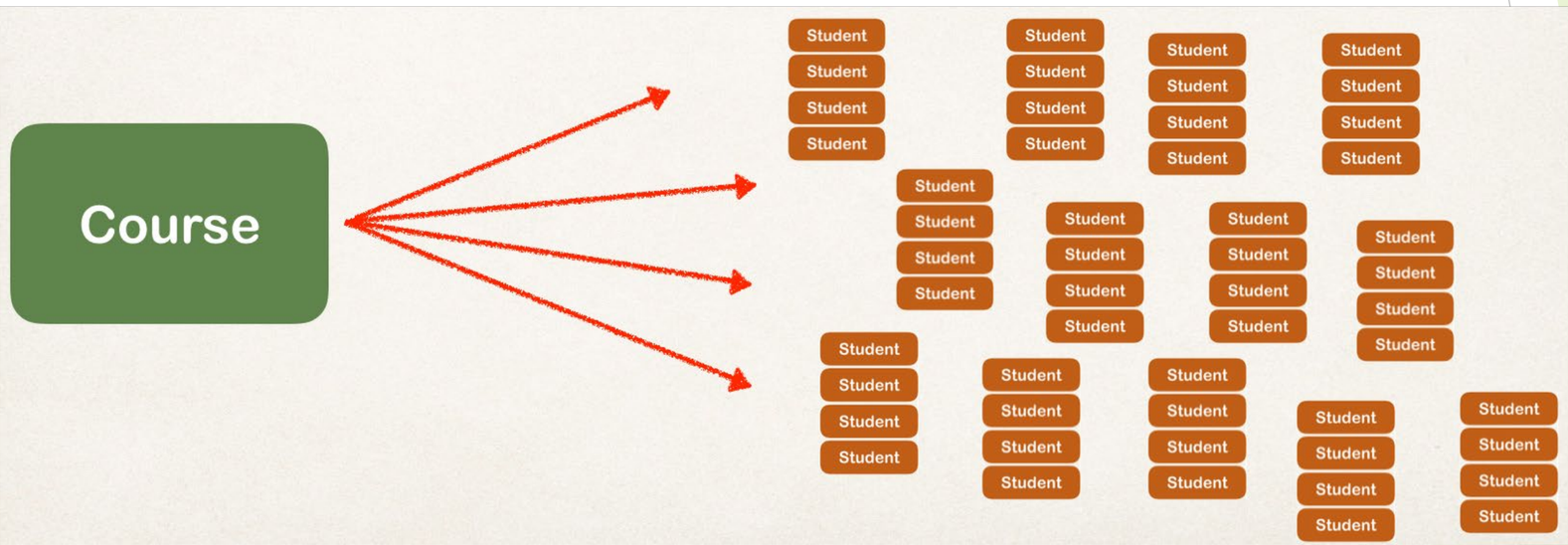
Best Practice

Caricare i dati solo quando assolutamente necessario

Preferire lazy invece di eager

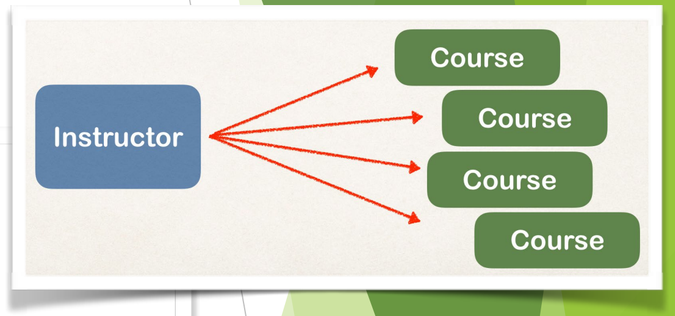
Lazy Loading

- Il lazy loading carica prima l'entità principale
- Poi, le entità dipendenti quando richieste



Fetch Type

- Quando si definisce una relazione
- È possibile specificare il tipo di loading: EAGER o LAZY




```
@Entity
@Table(name="instructor")
public class Instructor {
   ...

   @OneToMany(fetch=FetchType.LAZY, mappedBy="instructor")
   private List<Course> courses;

   ...

}
```

Default Fetch Types

@OneToOne	FetchType.EAGER
@OneToMany	FetchType.LAZY
@ManyToOne	FetchType.EAGER
@ManyToMany	FetchType.LAZY

Overriding Default Fetch Type

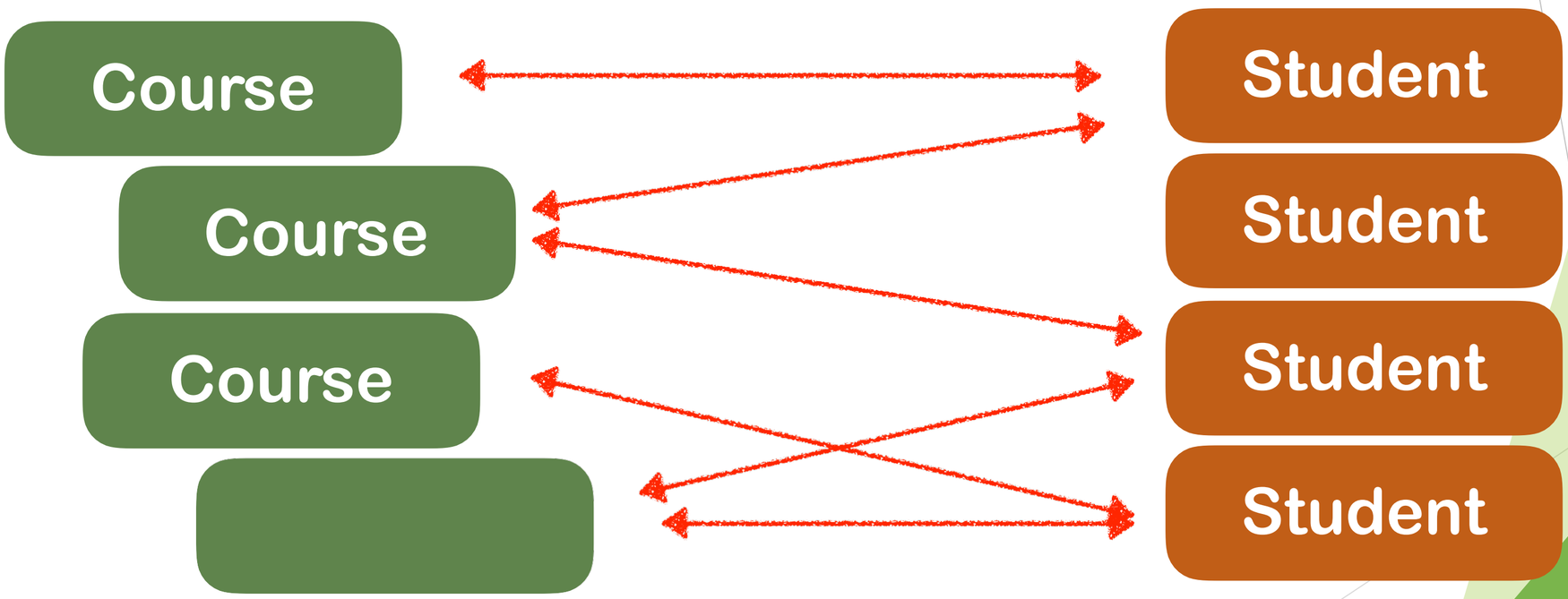
- Specificando il tipo di caricamento, si esegue l'override delle impostazioni predefinite

Mapping	Default Fetch Type
@ManyToOne	FetchType.EAGER

```
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="instructor_id")
private Instructor instructor;
```

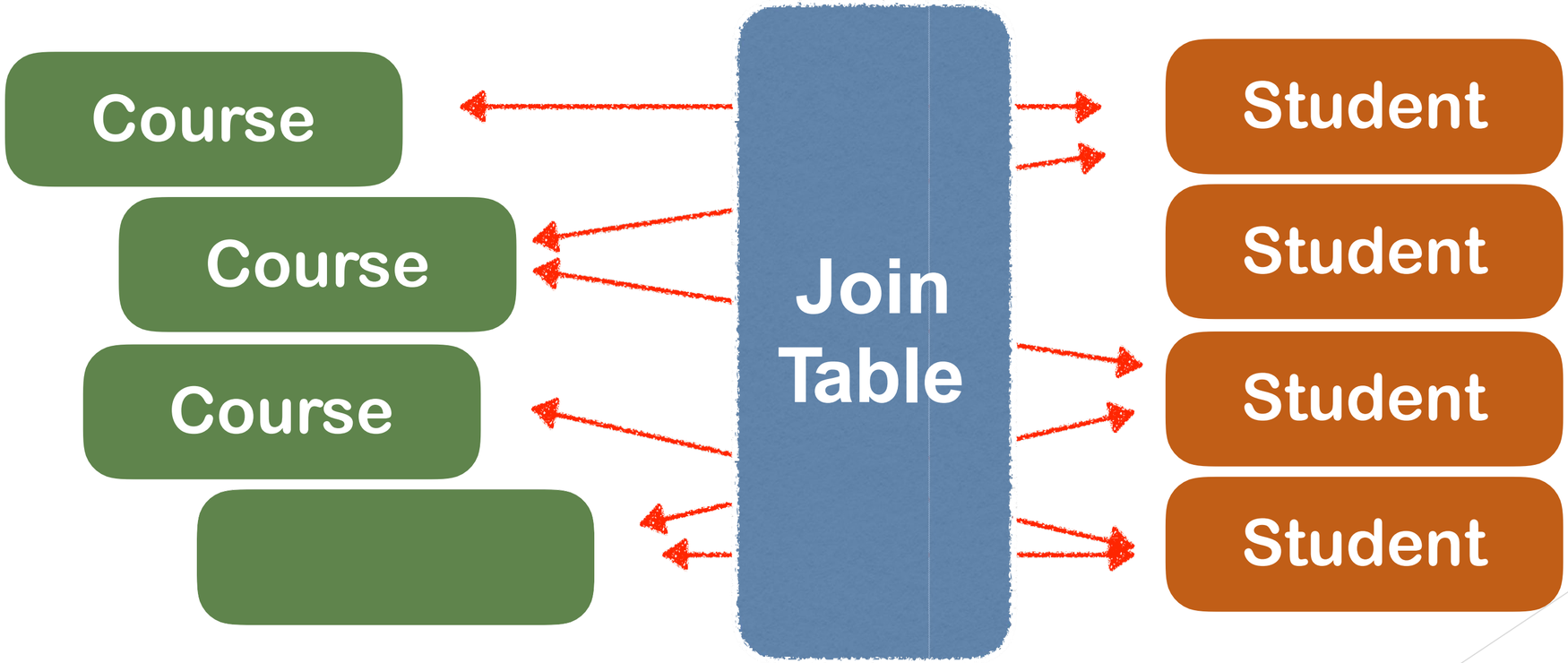
Many-to-Many

- Un corso può avere molti studenti
- Uno studente può avere molti corsi



Tenere traccia delle relazioni

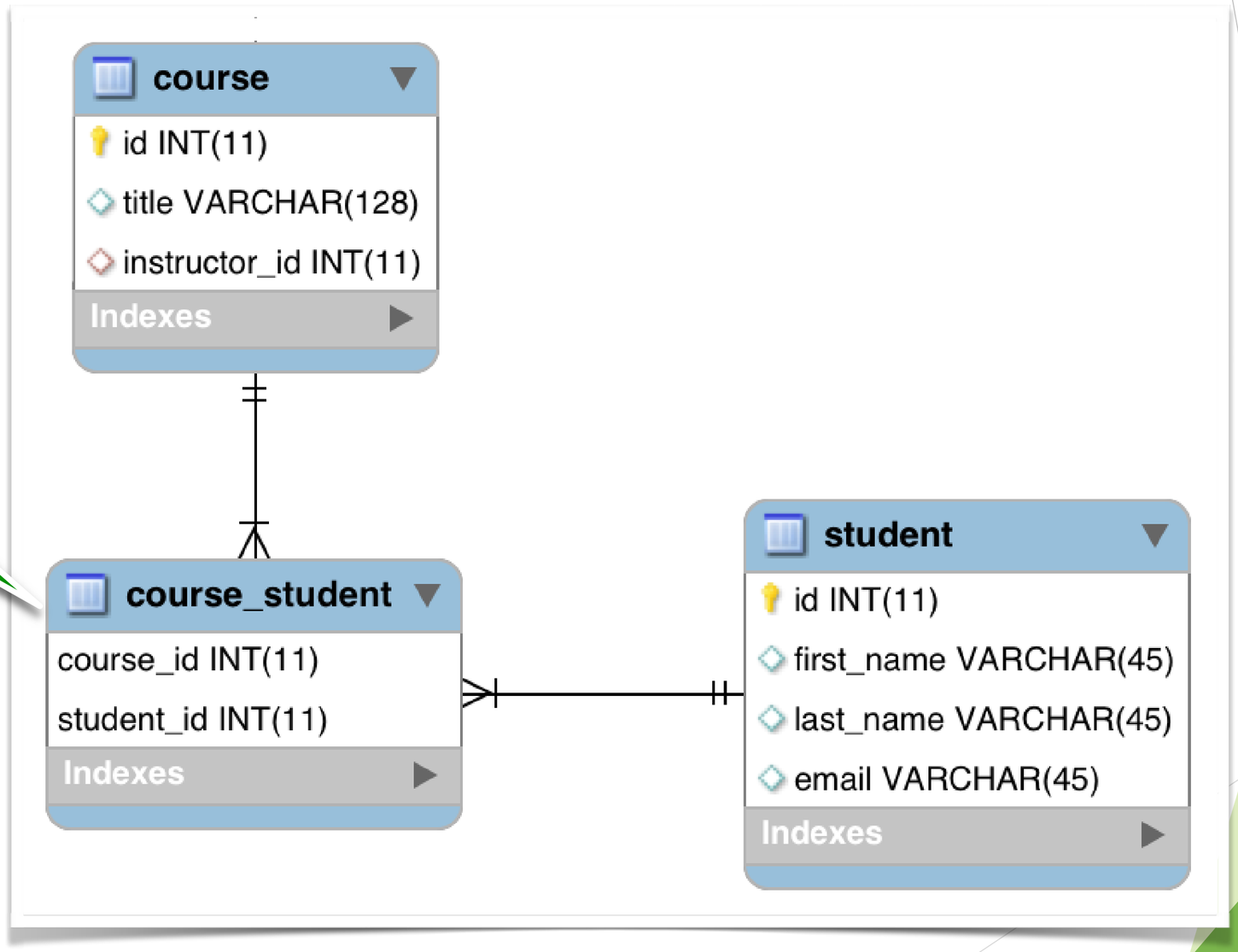
- Necessità di tenere traccia di quale studente è in quale corso e viceversa



Join Table

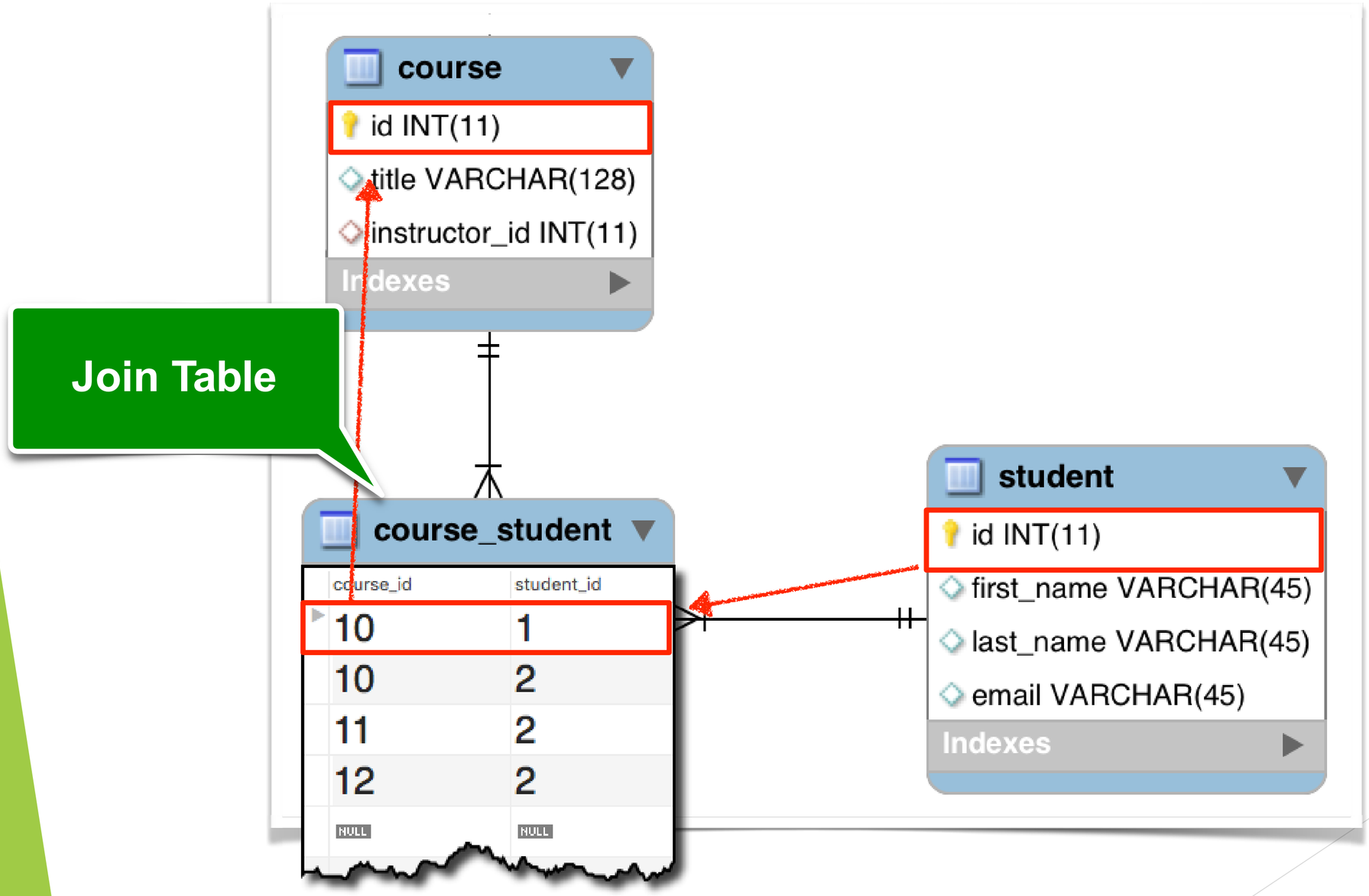
Tabella che fornisce un mapping tra due tabelle. Dispone di chiavi esterne per ogni tabella per definire la relazione di mapping.

@ManyToMany



Join Table

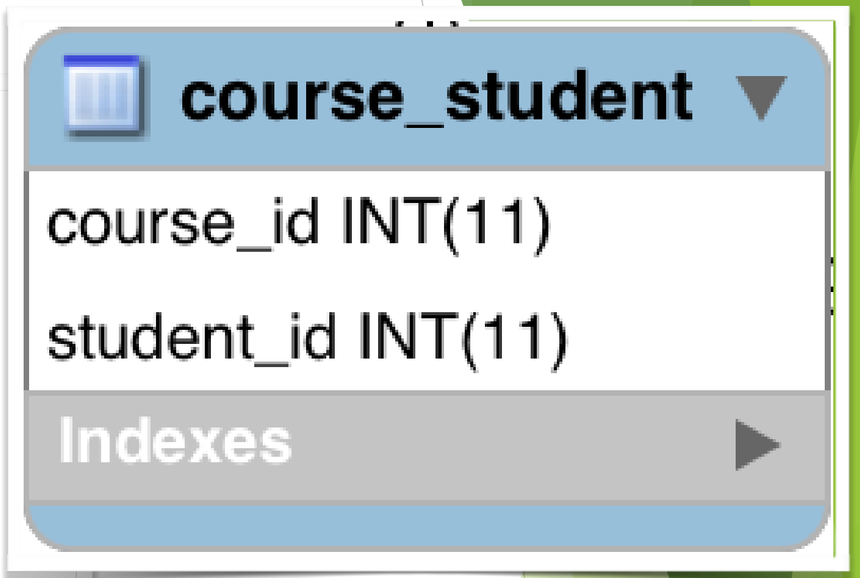
Join Table Example



join table: course_student

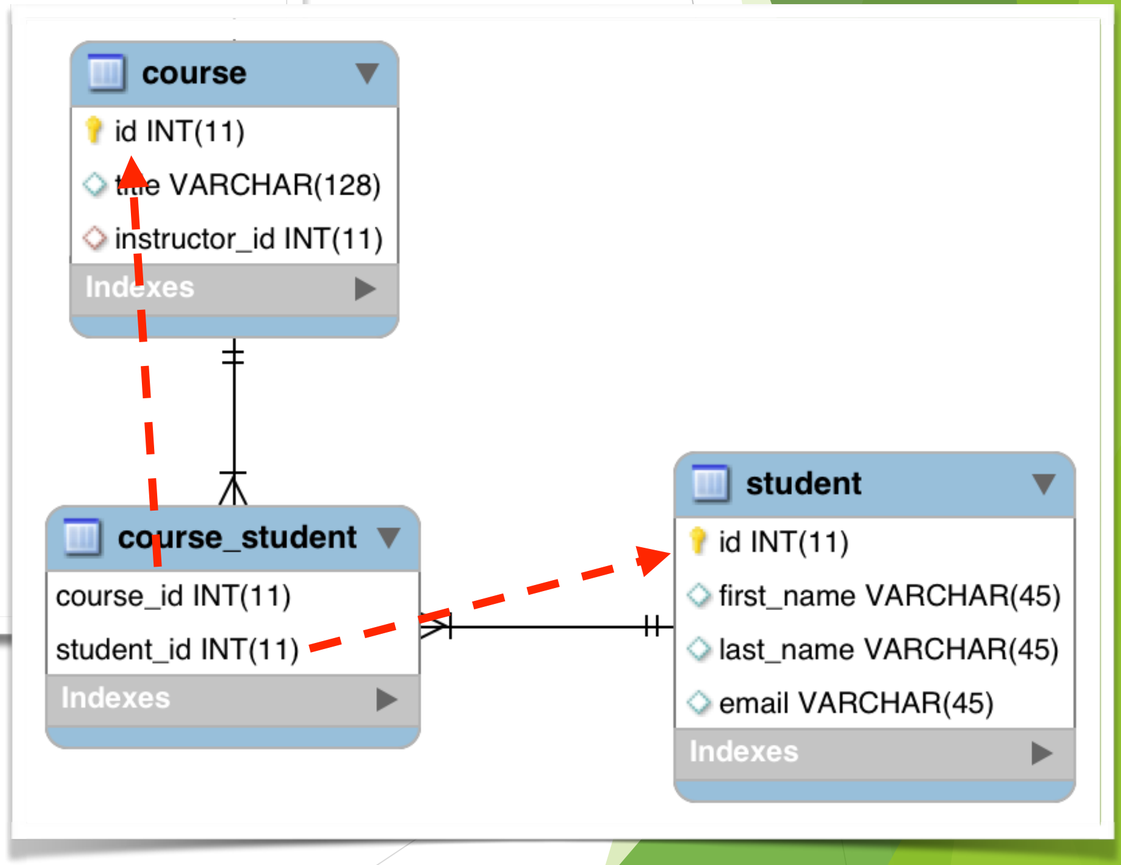
File: create-db.sql

```
CREATE TABLE `course_student` (

  `course_id` int(11) NOT NULL,
  `student_id` int(11) NOT NULL,

  PRIMARY KEY (`course_id`, `student_id`),

    ...
);
```



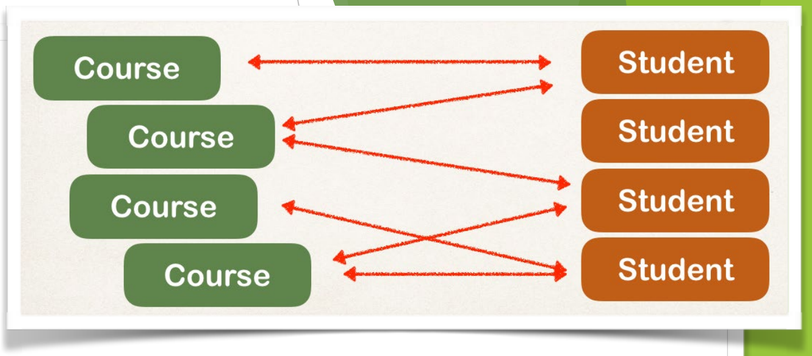
join table: course_student - foreign keys

```
CREATE TABLE `course_student` (
  …
  CONSTRAINT `FK_COURSE_05`
  FOREIGN KEY (`course_id`)
  REFERENCES `course` (`id`),

  CONSTRAINT `FK_STUDENT`
  FOREIGN KEY (`student_id`)
  REFERENCES `student` (`id`)
  …
);
```



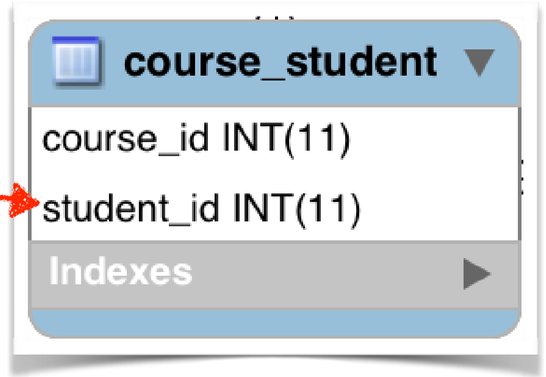
Aggiornare classe Course

```
@Entity
@Table(name="course")
public class Course {
   ...



   private List<Student> students;

   // getter / setters
   ...

}
```



Aggiungere @ManyToMany

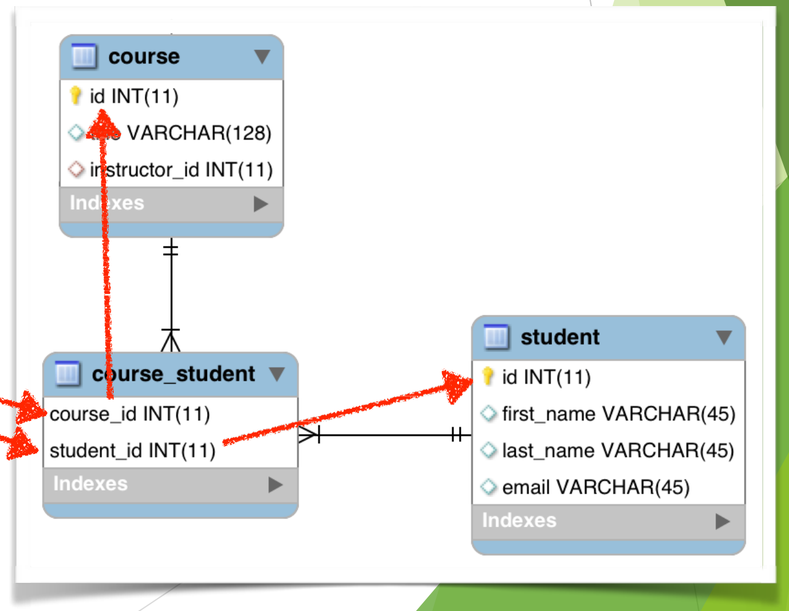
```
@Entity
@Table(name="course")
public class Course {
   …

   @ManyToMany
   @JoinTable(
         name="course_student",
         joinColumns=@JoinColumn(name="course_id"),
         inverseJoinColumns=@JoinColumn(name="student_id")
         )
   private List<Student> students;

   // getter / setters
   …

}
```



@JoinTable

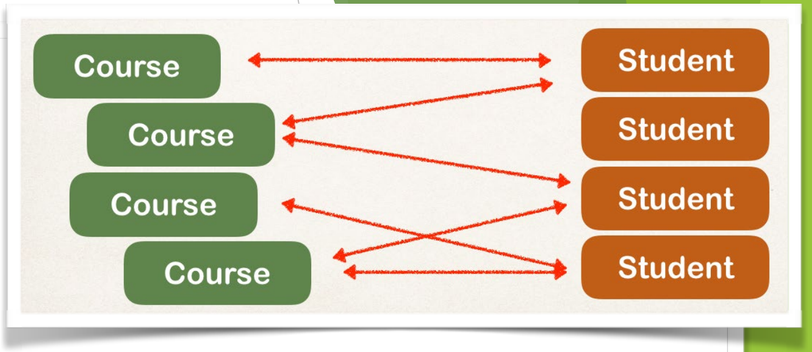
- **@JoinTable** dice ad Hibernate
  - Guarda la colonna course_id nella tabella course_student
  - Per ogni course_id recupera i valori di student_id per ottenere le informazioni sugli studenti correlati al corso rappresentato da course_id

```
public class Course {

    @ManyToMany
    @JoinTable(
            name="course_student",
            joinColumns=@JoinColumn(name="course_id"),
            inverseJoinColumns=@JoinColumn(name="student_id")
            )
    private List<Student> students;

}
```



Aggiornare Student

```
@Entity
@Table(name="student")
public class Student {
   ...



   private List<Course> courses;

   // getter / setters
   ...

}
```



Aggiungere @ManyToMany

```
@Entity
@Table(name="student")
public class Student {
   …

   @ManyToMany
   @JoinTable(
         name="course_student",
         joinColumns=@JoinColumn(name="student_id"),
         inverseJoinColumns=@JoinColumn(name="course_id")
         )
   private List<Course> courses;

   // getter / setters
   …

}
```