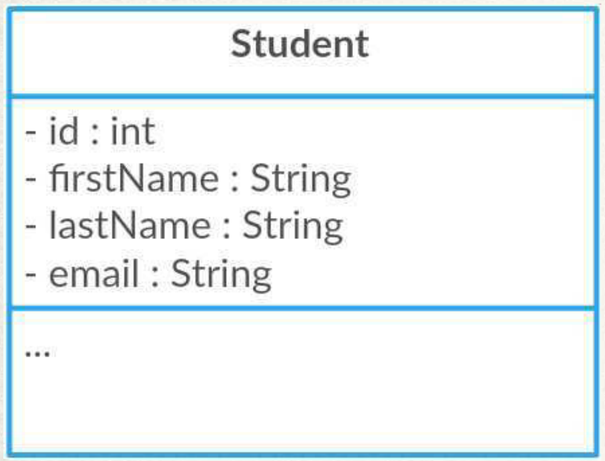
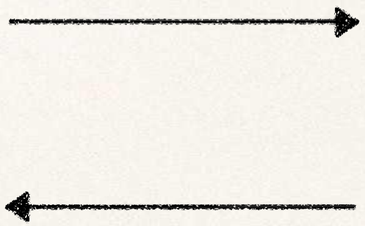
JPA / Hibernate
Advanced Mappings

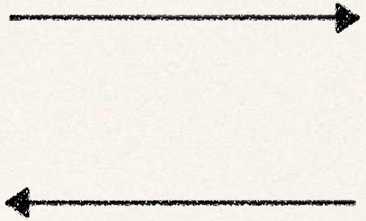
Basic Mapping

Java Class

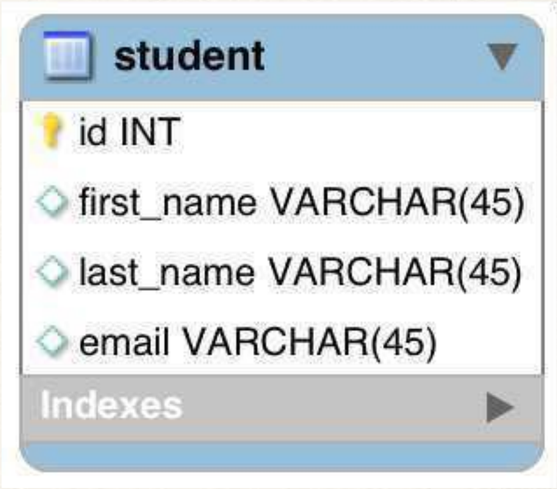




Hibernate



Database Table



Advanced Mappings

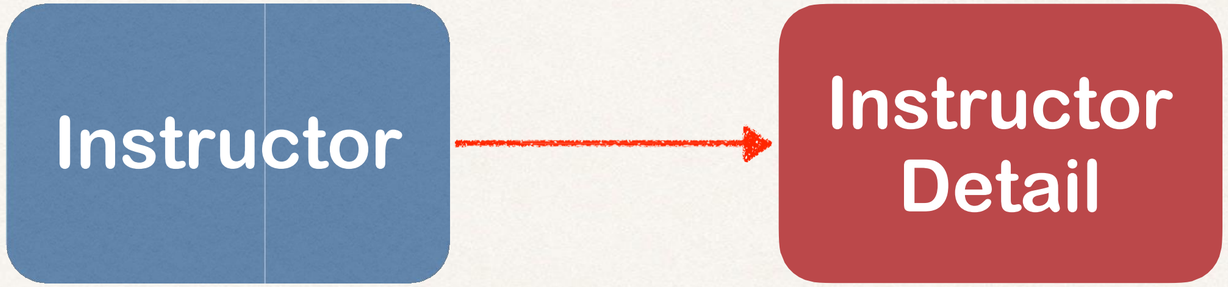
- In the database, you most likely will have
  - Multiple Tables
  - Relationships between Tables
- Need to model this with Hibernate

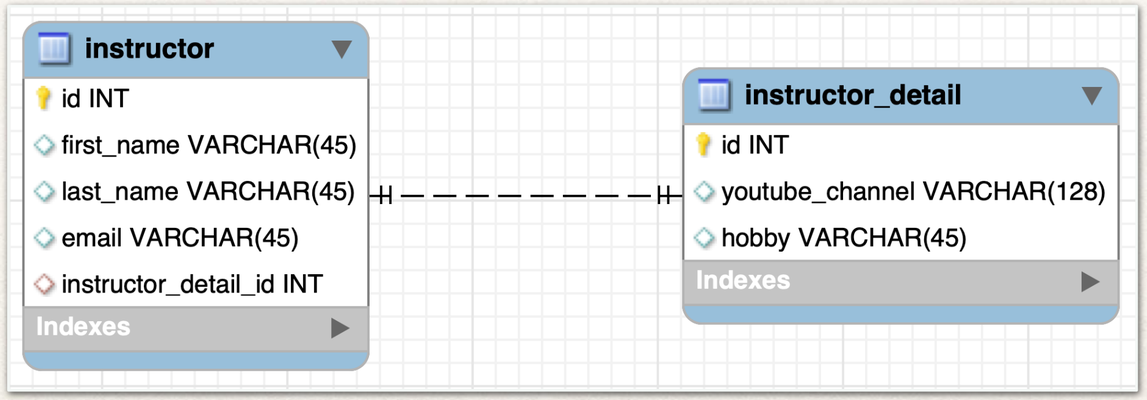
Advanced Mappings

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

One-to-One Mapping

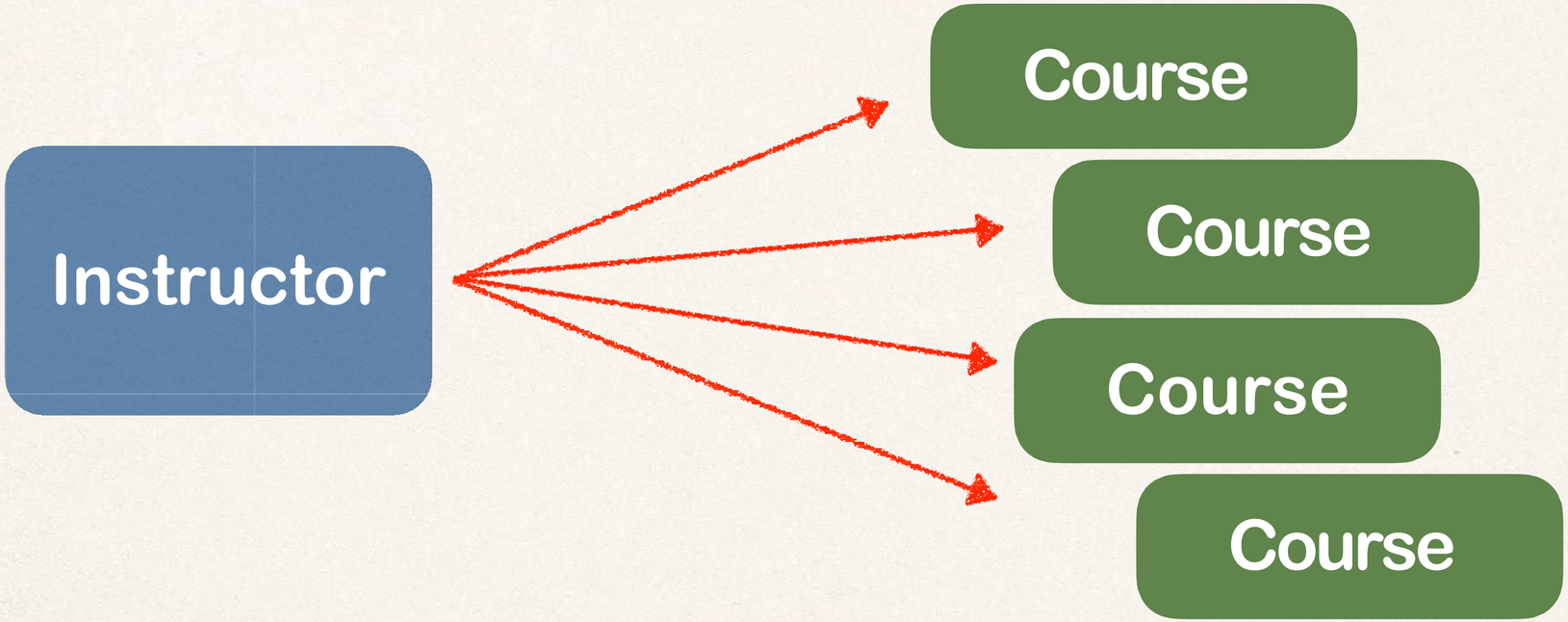
- An instructor can have an “instructor detail” entity
  - Similar to an “instructor profile”





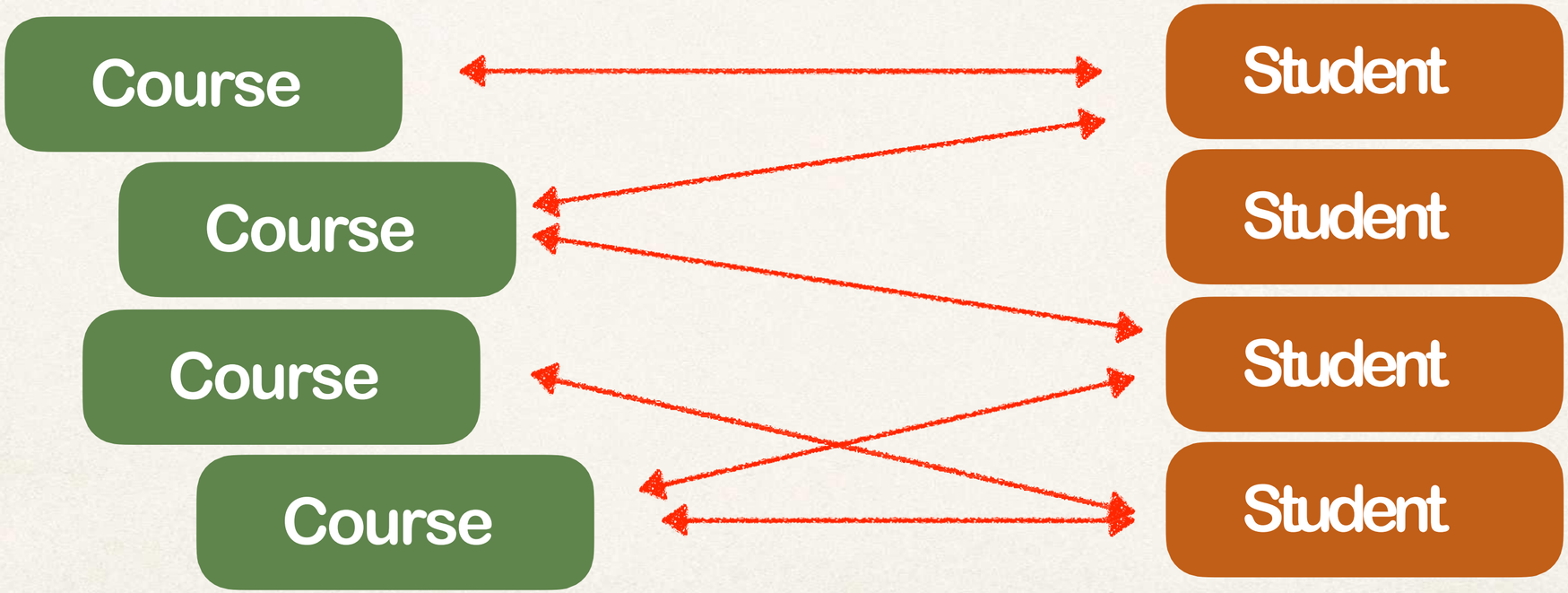
One-to-Many Mapping

- An instructor can have many courses



Many-to-Many Mapping

- A course can have many students
- A student can have many courses



Important Database Concepts

- Primary key and foreign key

- Cascade

Primary Key and Foreign Key

- Primary key: identify a unique row in a table

- Foreign key:
  - Link tables together
  - a field in one table that refers to primary key in another table

Foreign Key Example

Table: instructor

id	first_name	last_name	instructor_detail_id
1	Chad	Darby	100
2	Madhu	Patel	200

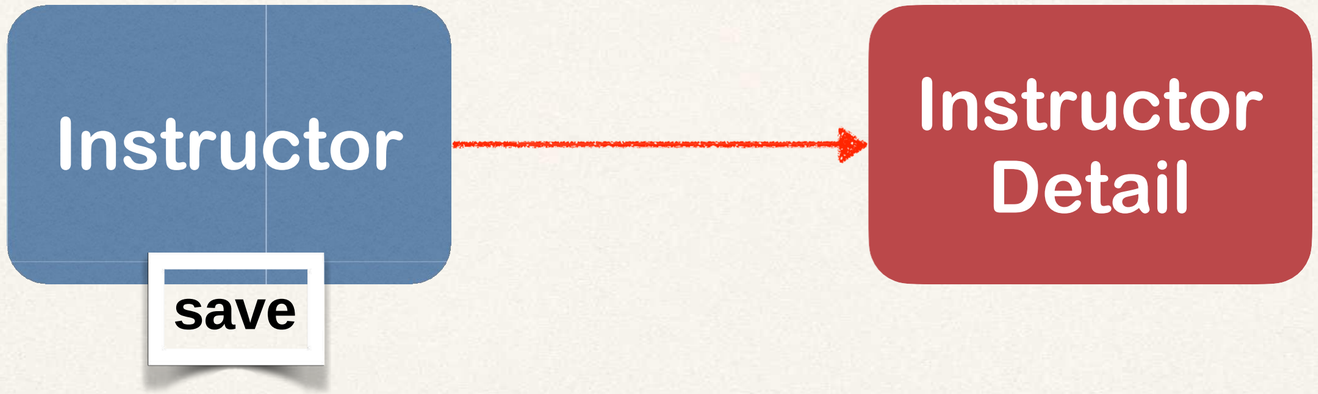
Table: instructor_detail

id	youtube_channel	hobby
100	www.luv2code.com/youtube	Luv 2 Code!!!
200	www.youtube.com	Guitar

Cascade

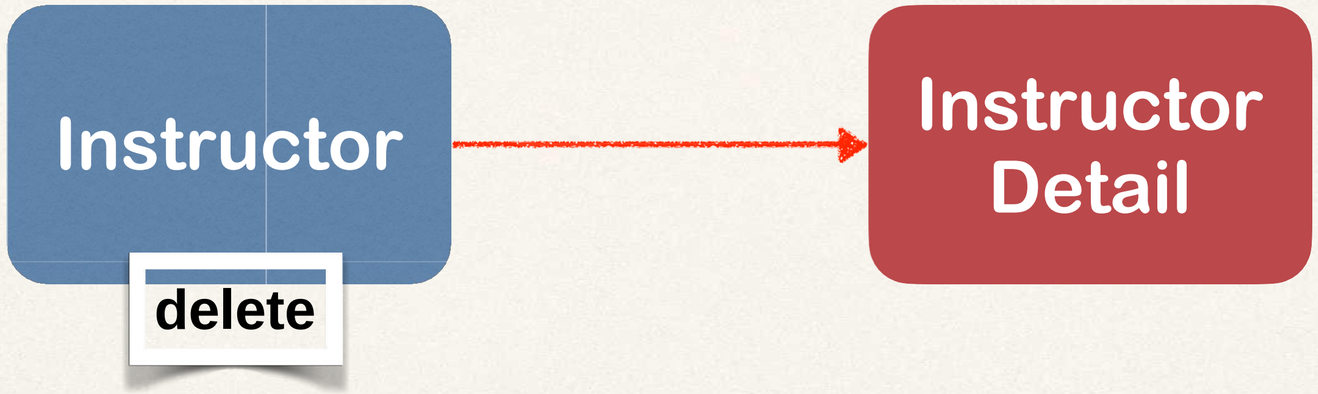
- You can **cascade** operations

- Apply the same operation to related entities



Cascade

- If we delete an **instructor**, we should also delete their **instructor_detail**
  - This is known as “CASCADE DELETE”



Cascade Delete

Foreign key column

Table: instructor

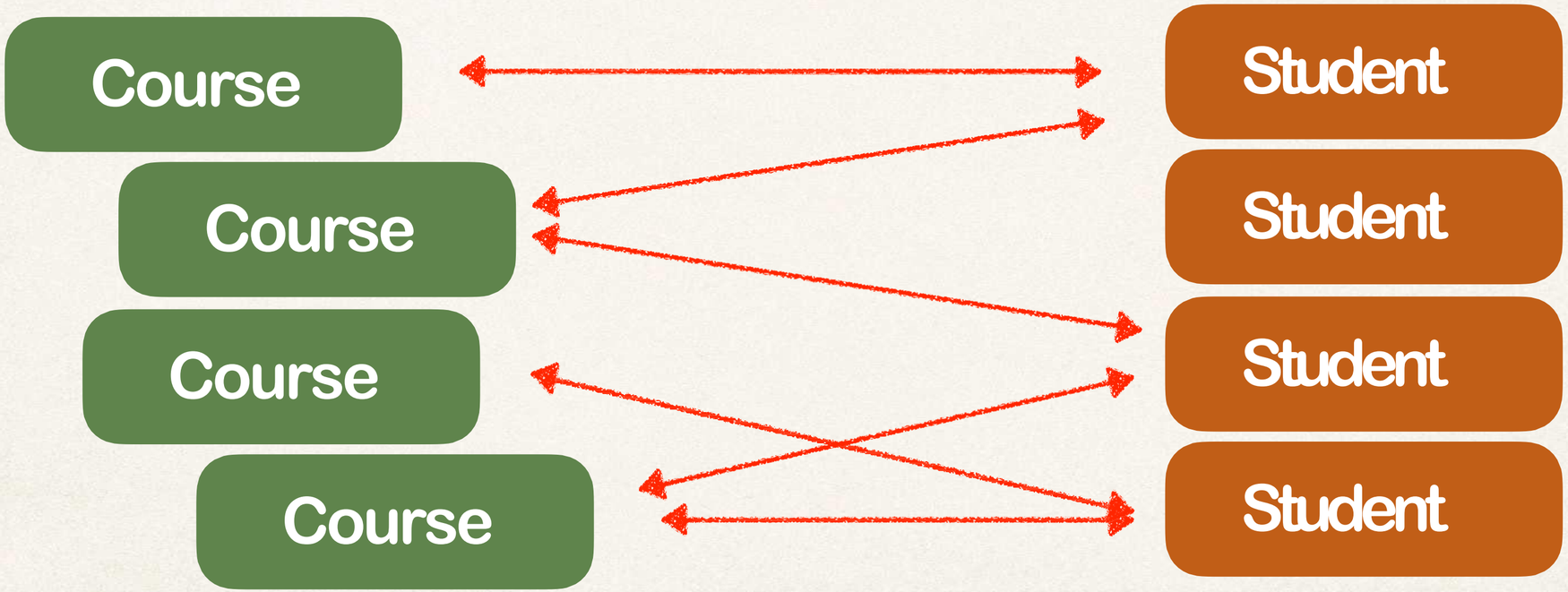
id	first_name	last_name	instructor_detail_id
1	Chad	Darby	100
2	Madhu	Patel	200

Table: instructor_detail

id	youtube_channel	hobby
100	www.luv2code.com/youtube	Luv 2 Code!!!
200	www.youtube.com	Guitar

Cascade Delete

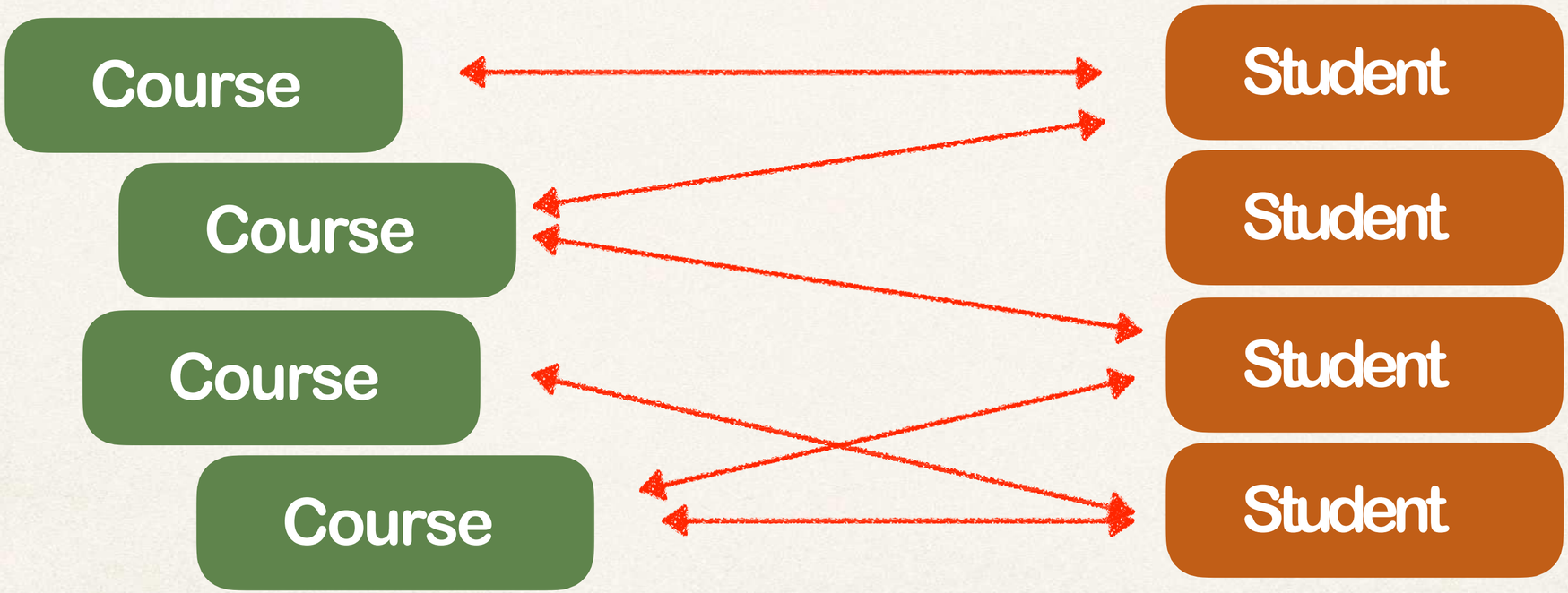
- Cascade delete depends on the use case
- Should we do cascade delete here???



Cascade Delete

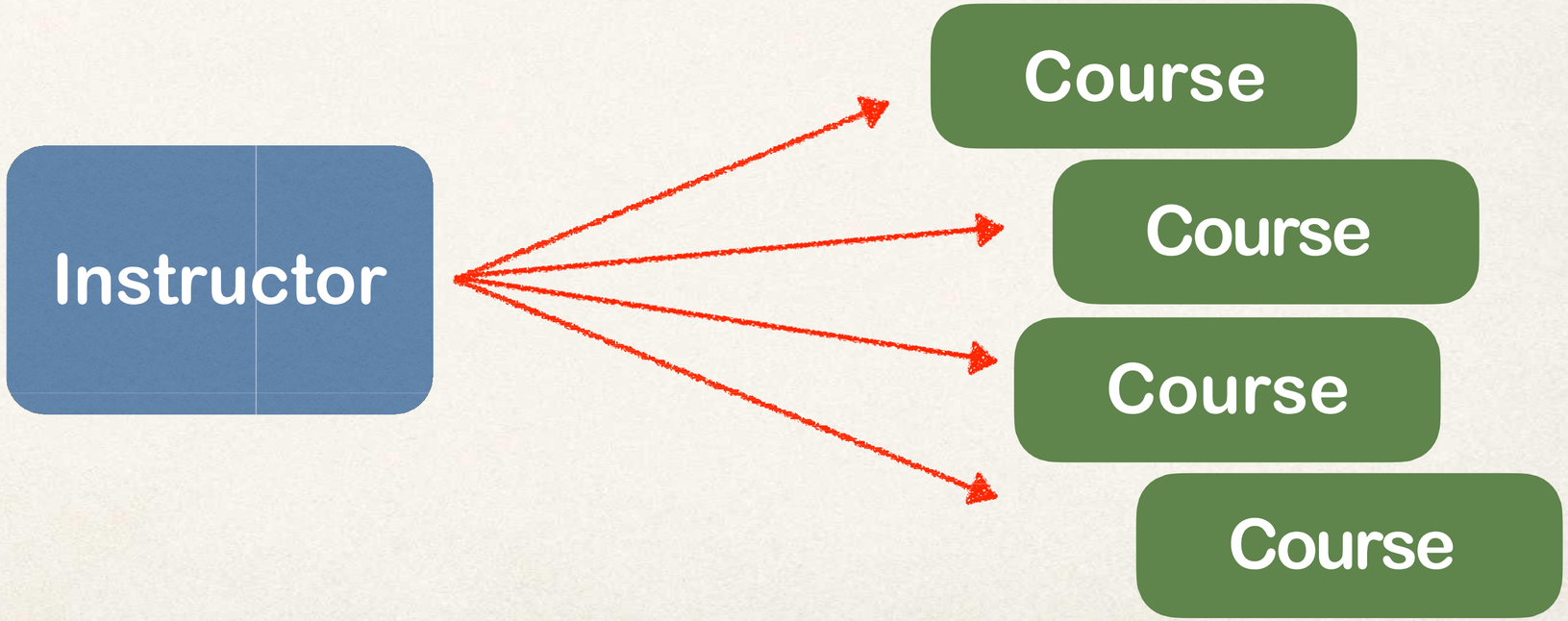
- Cascade delete depends on the use case
- Should we do cascade delete here???

Developer can configure cascading

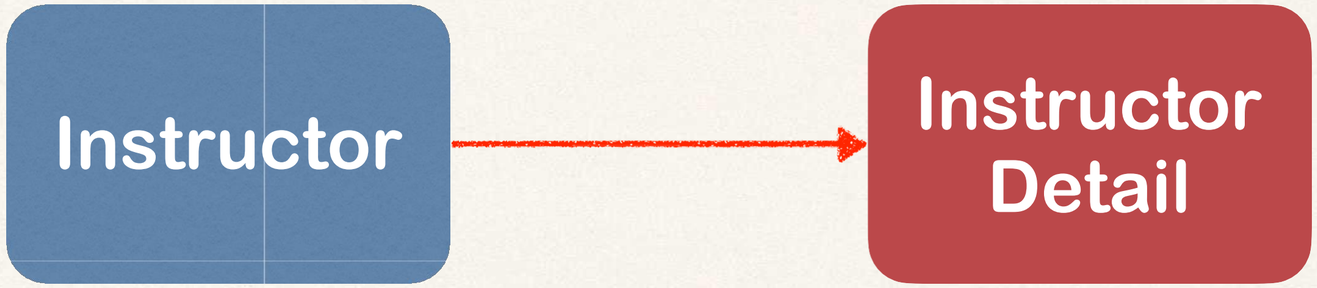


Fetch Types: Eager vs Lazy Loading

- When we fetch / retrieve data, should we retrieve EVERYTHING?
  - **Eager** will retrieve everything
  - **Lazy** will retrieve on request



Uni-Directional

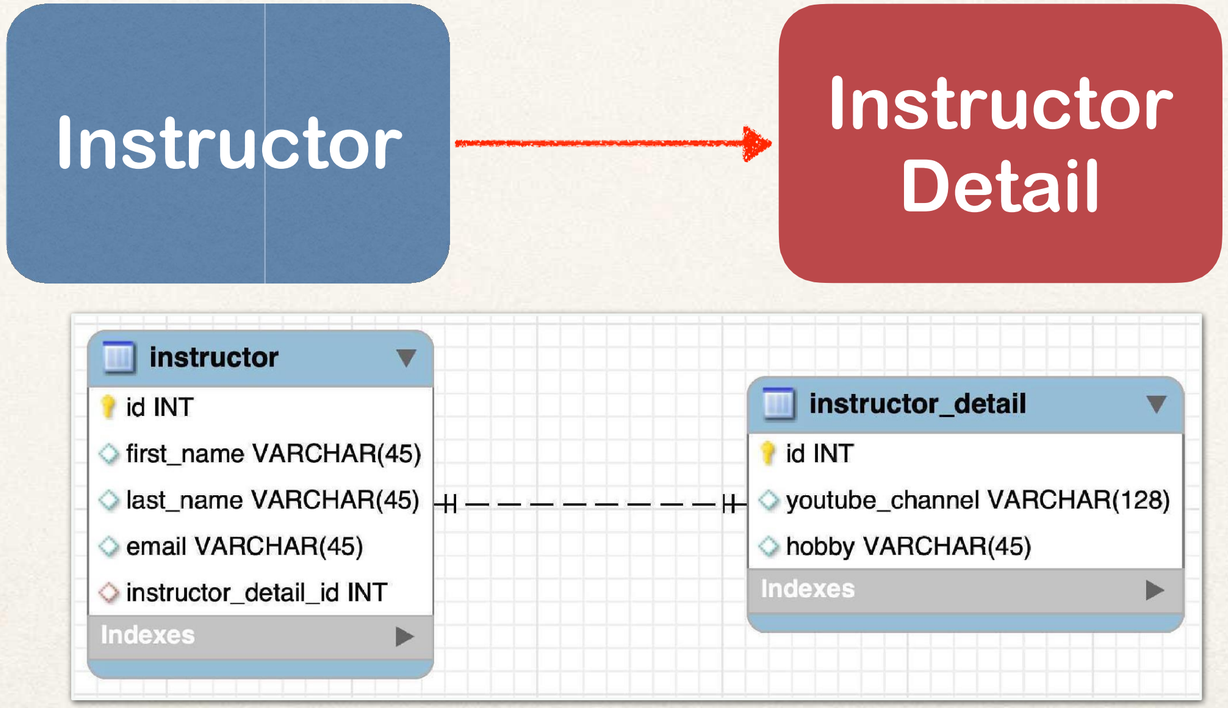


Bi-Directional

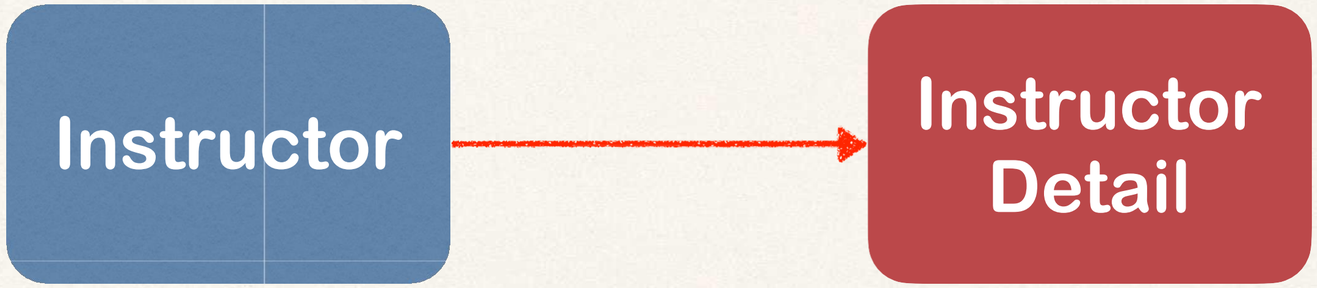
JPA / Hibernate
One-to-One

One-to-One Mapping

- An instructor can have an “instructor detail” entity
  - Similar to an “instructor profile”



Uni-Directional



Development Process: One-to-One

1. Prep Work - Define database tables
2. Create InstructorDetail class
3. Create Instructor class
4. Create Main App

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,


);

...
```

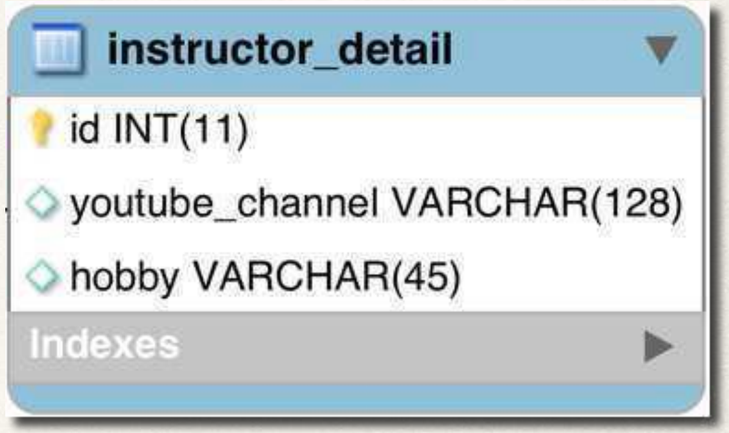


table: instructor_detail

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`)
);

...
```

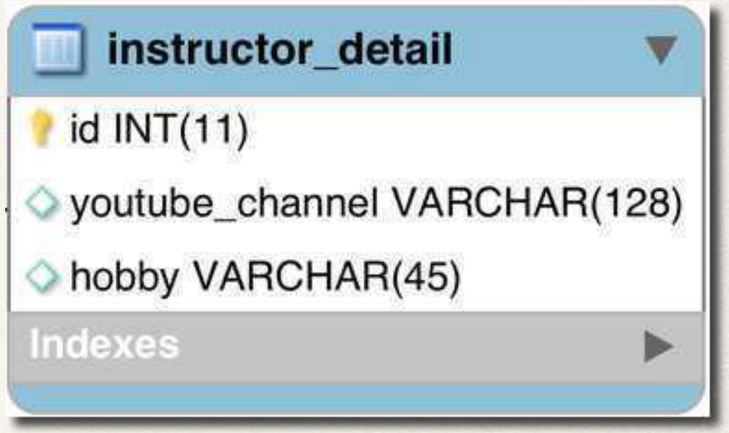


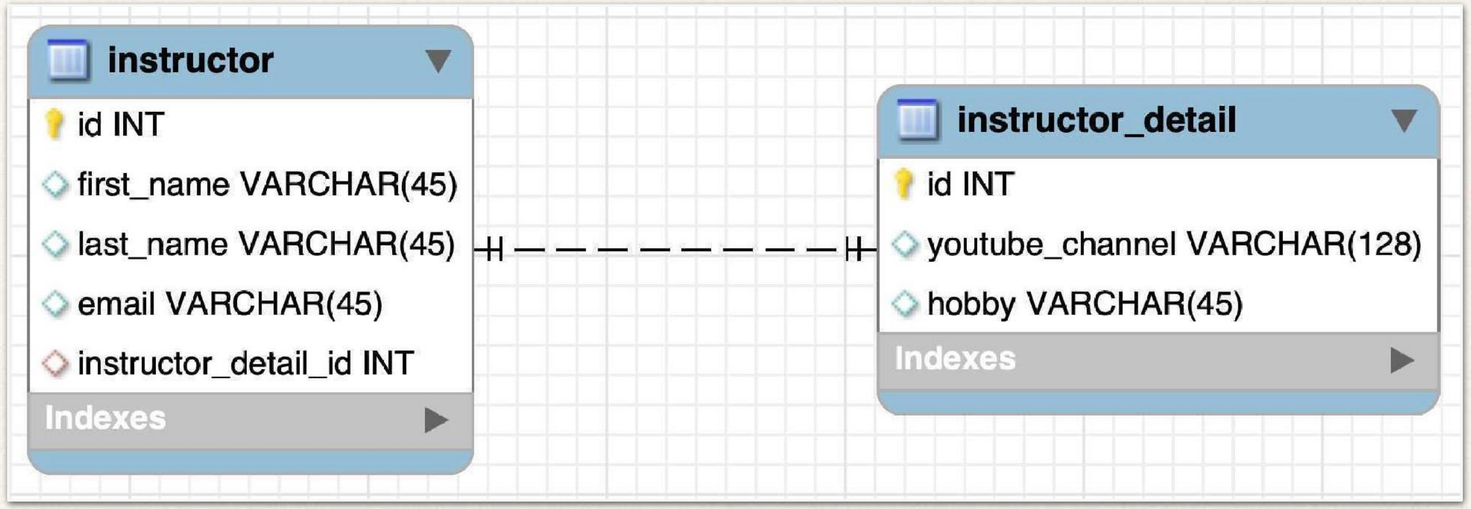
table: instructor

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`)
    ...
);
```

Foreign Key

- Link tables together
- A field in one table that refers to primary key in another table



Foreign Key Example

Table: instructor

id	first_name	last_name	instructor_detail_id
1	Chad	Darby	100
2	Madhu	Patel	200

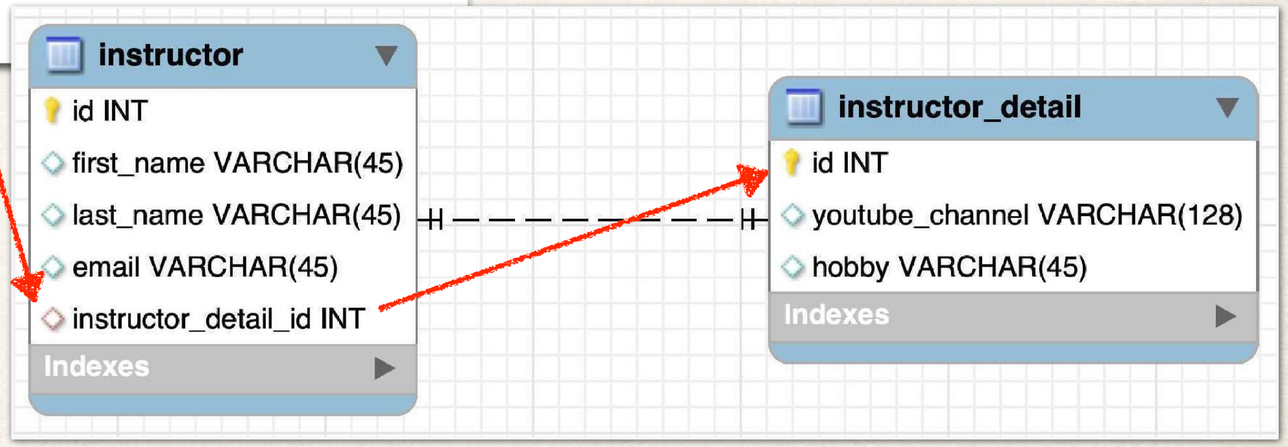
Table: instructor_detail

id	youtube_channel	hobby
100	www.luv2code.com/youtube	Luv 2 Code!!!
200	www.youtube.com	Guitar

Defining Foreign Key

File: create-db.sql

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

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

);
```



More on Foreign Key

- Main purpose is to preserve relationship between tables
  - Referential Integrity
- Prevents operations that would destroy relationship
- Ensures only valid data is inserted into the foreign key column
  - Can only contain valid reference to primary key in other table

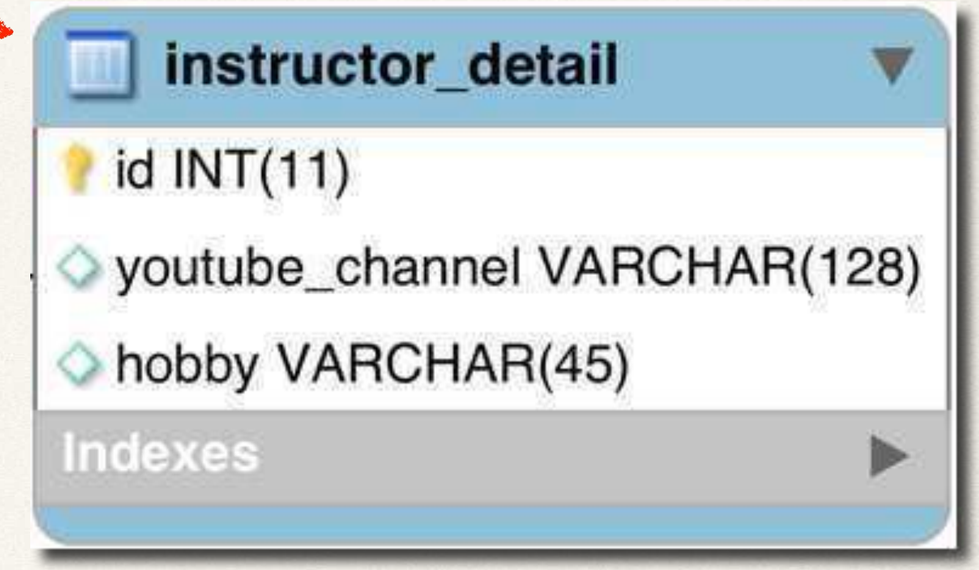
Development Process: One-to-One

Step-By-Step

1. Prep Work - Define database tables
2. Create InstructorDetail class
3. Create Instructor class
4. Create Main App

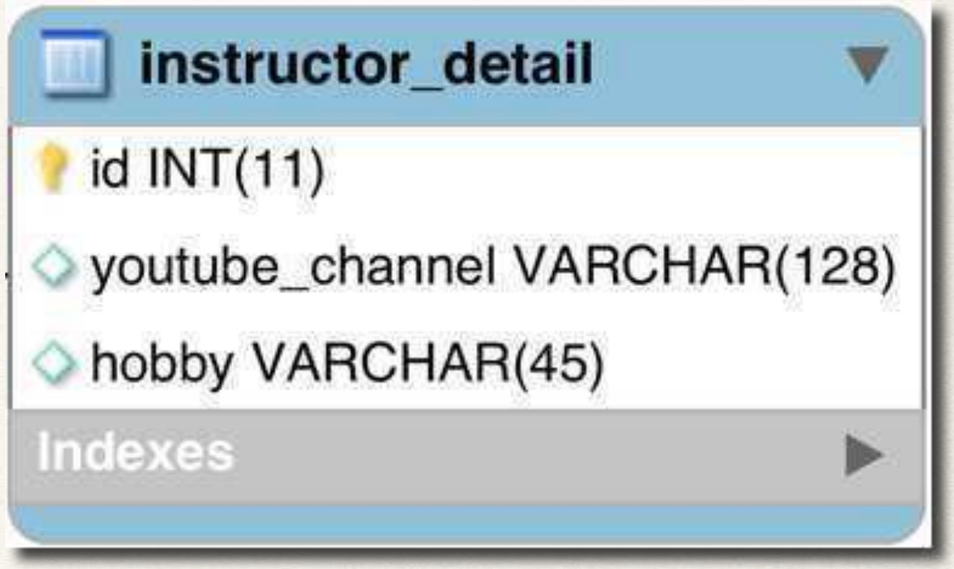
Step 2: Create InstructorDetail class

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



}
```



Step 2: Create InstructorDetail class

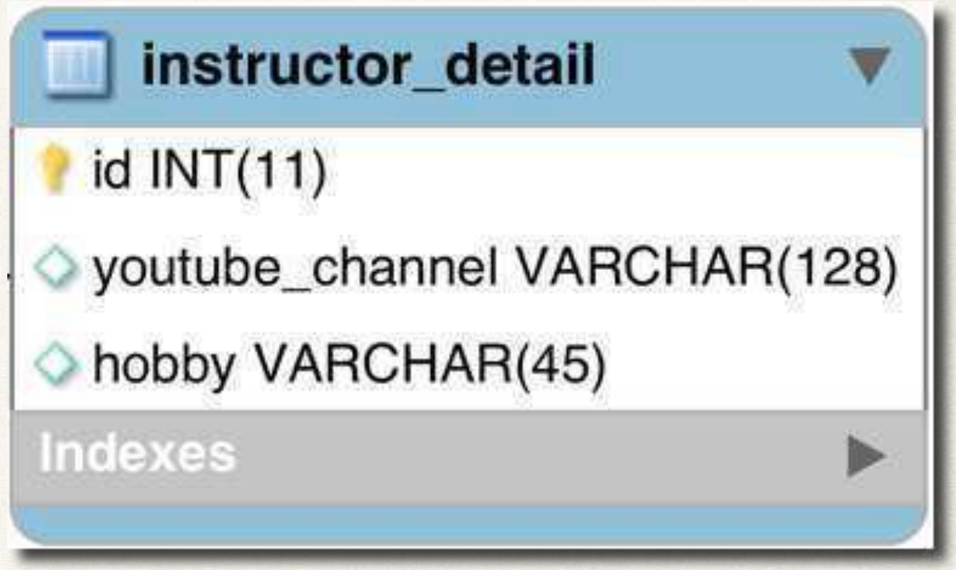
```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {

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



}
```



luv2code

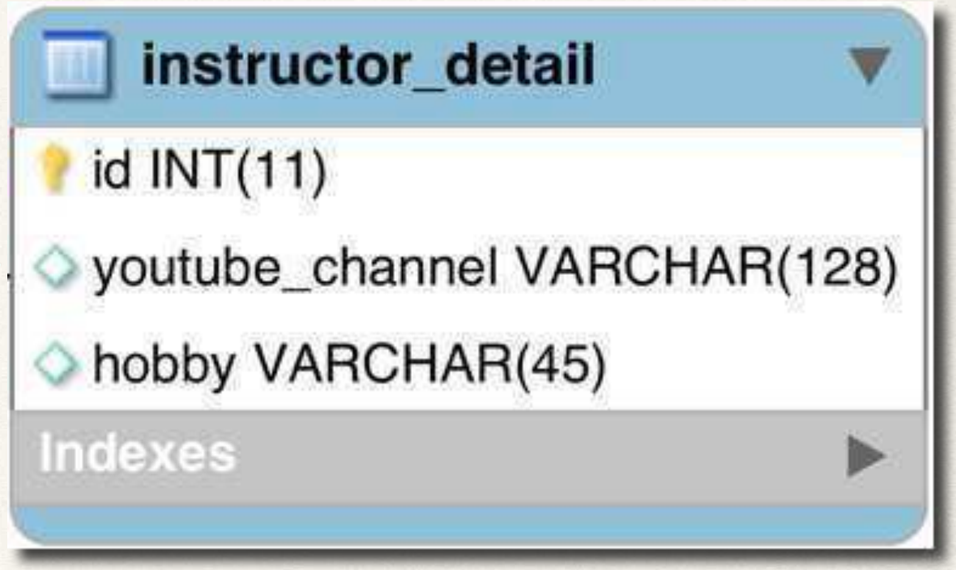
Step 2: Create InstructorDetail class

```
@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;



}
```



Step 2: Create InstructorDetail class

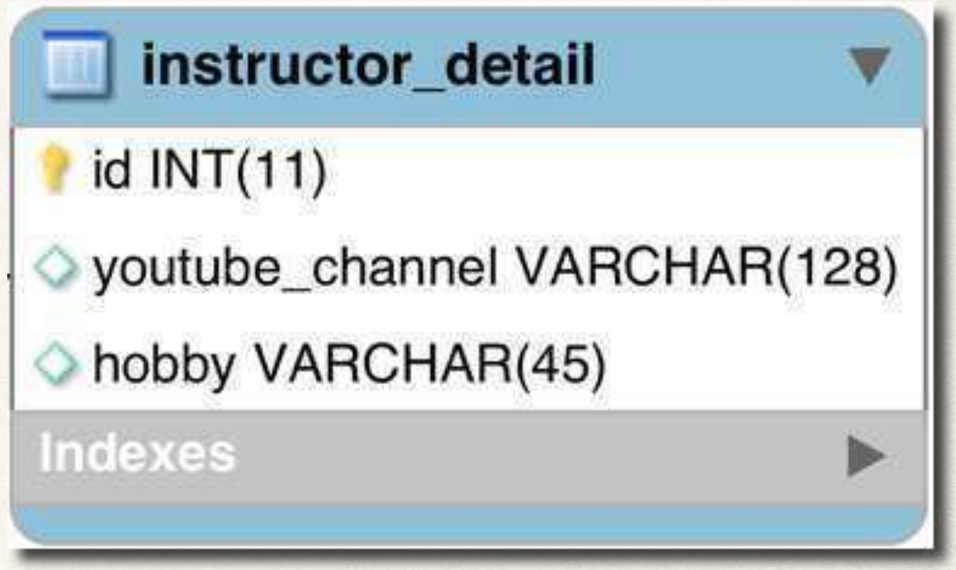
```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {

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

    @Column(name="youtube_channel")
    private String youtubeChannel;

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



}
```



luv2code

Step 2: Create InstructorDetail class

```
@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;

    // constructors

    // getters / setters
}
```



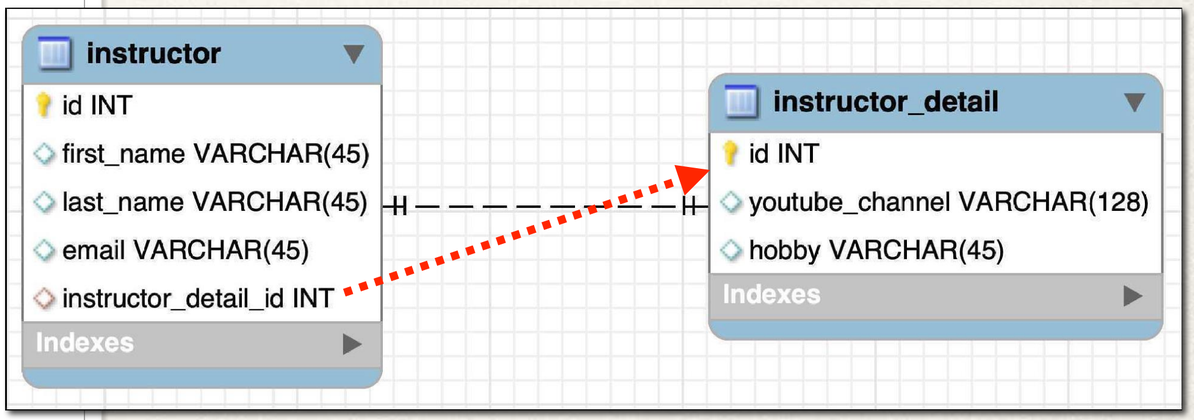
Step 3: Create Instructor class

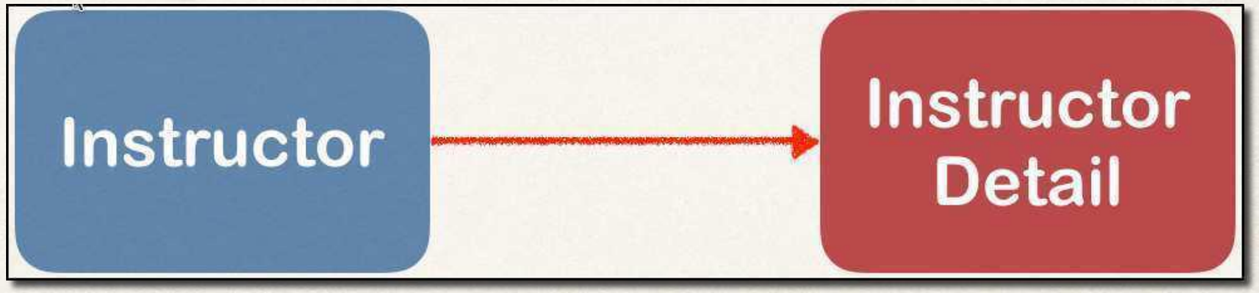
```
@Entity
@Table(name="instructor")
public class Instructor {

    @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
}
```

luv2code

Step 3: Create Instructor class - @OneToOne

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

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

    ...
    // constructors, getters / setters
}
```



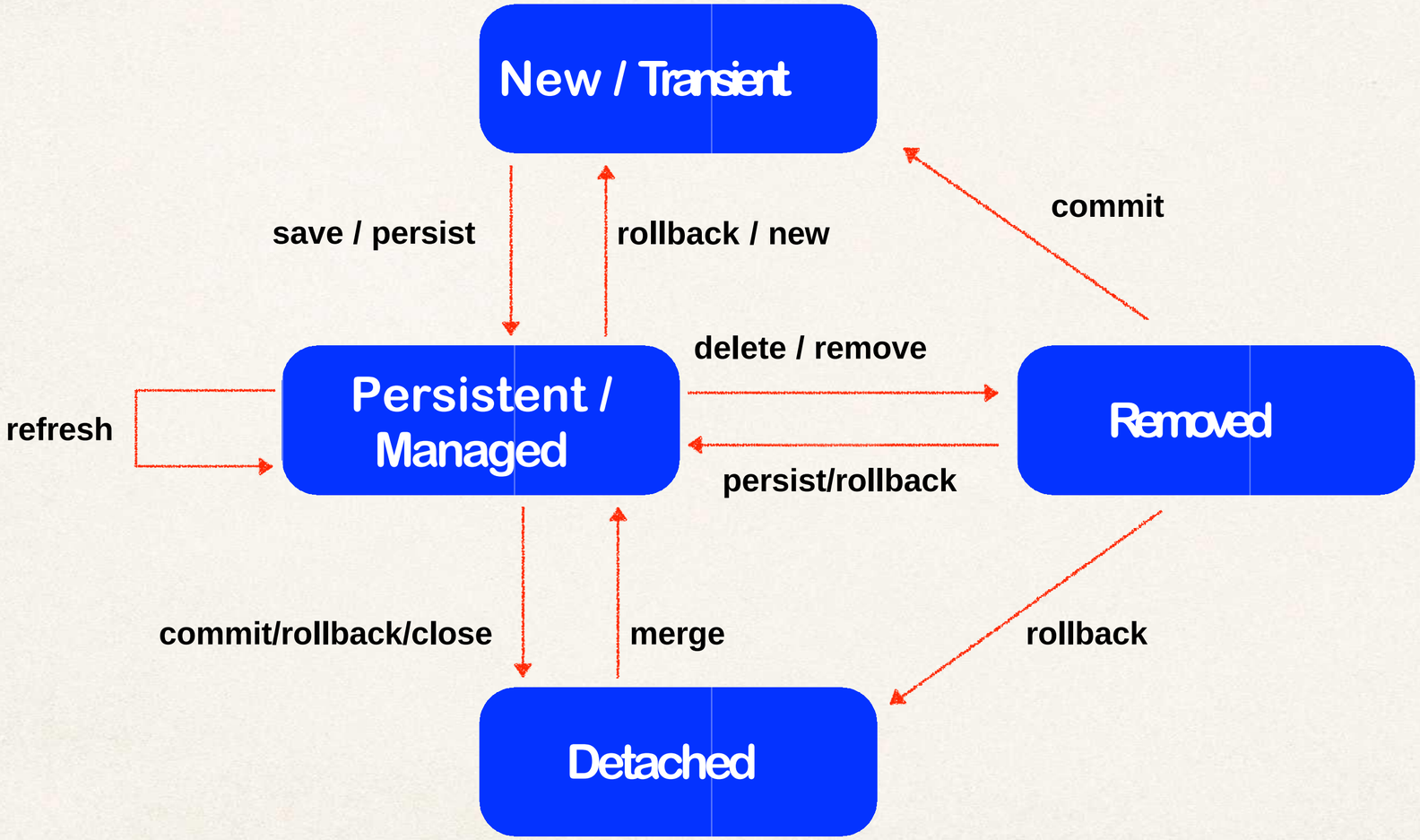


luv2code

Entity Lifecycle

Operations	Description
Detach	If entity is detached, it is not associated with a Hibernate session
Merge	If instance is detached from session, then merge will reattach to session
Persist	Transitions new instances to managed state. Next flush / commit will save in db.
Remove	Transitions managed entity to be removed. Next flush / commit will delete from db.
Refresh	Reload / synch object with data from db. Prevents stale data

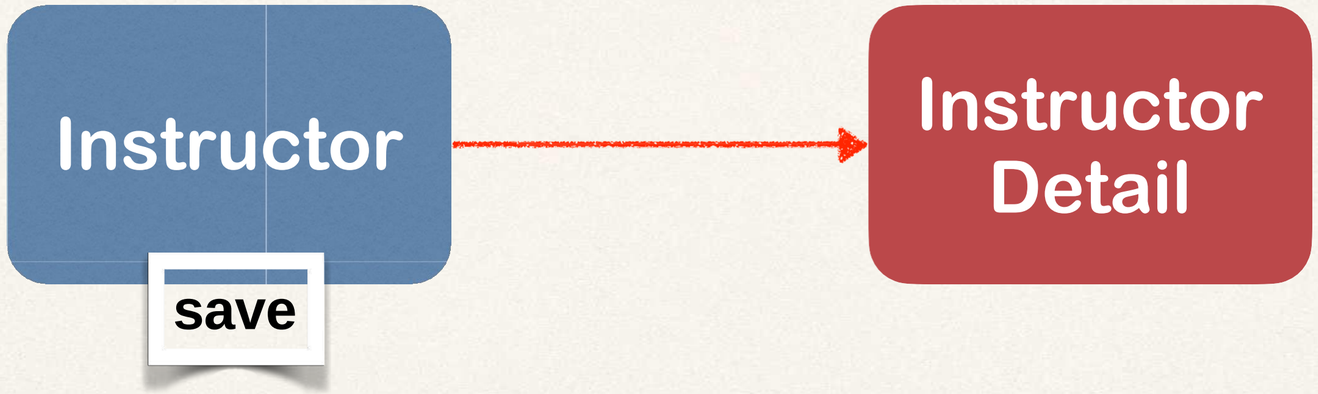
Entity Lifecycle - session method calls



Cascade

- Recall: You can **cascade** operations

- Apply the same operation to related entities



Cascade Delete

Foreign key column

Table: instructor

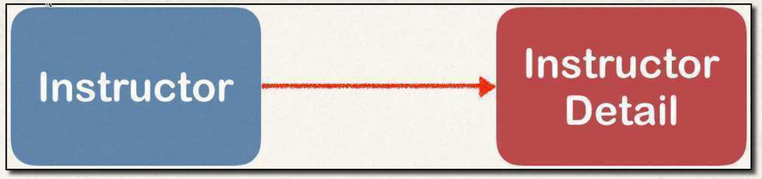
id	first_name	last_name	instructor_detail_id
1	Chad	Darby	100
2	Madhu	Patel	200

Table: instructor_detail

id	youtube_channel	hobby
100	www.luv2code.com/youtube	Luv 2 Code!!!
200	www.youtube.com	Guitar

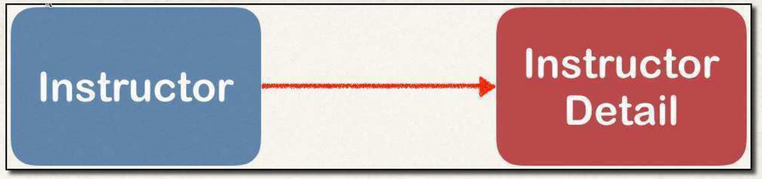
luv2code

@OneToOne - Cascade Types



Cascade Type	Description
PERSIST	If entity is persisted / saved, related entity will also be persisted
REMOVE	If entity is removed / deleted, related entity will also be deleted
REFRESH	If entity is refreshed, related entity will also be refreshed
DETACH	If entity is detached (not associated w/ session), then related entity will also be detached
MERGE	If entity is merged, then related entity will also be merged
ALL	All of above cascade types

Configure Cascade Type




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

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


    ...
    // constructors, getters / setters
}
```

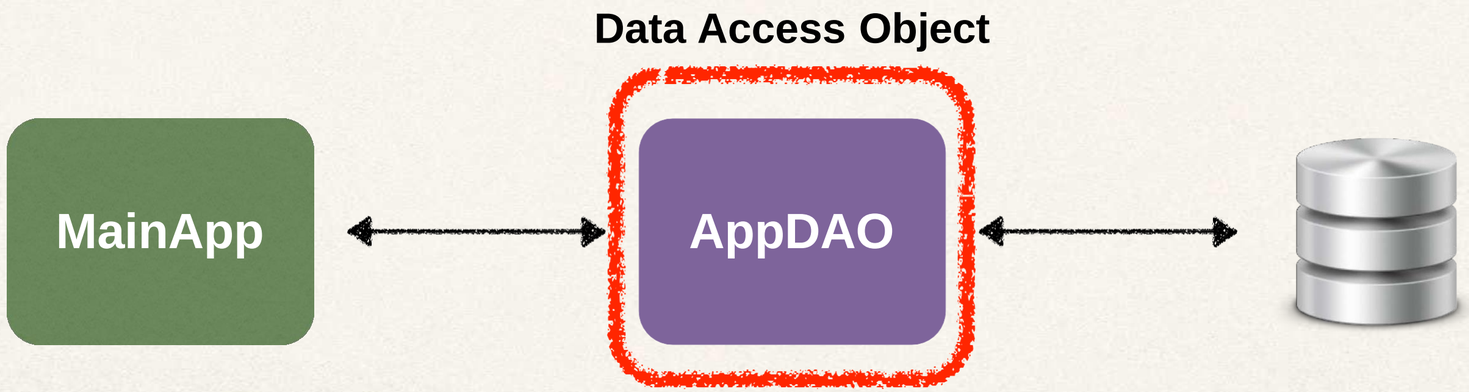
By default, no operations are cascaded.

Configure Multiple Cascade Types

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

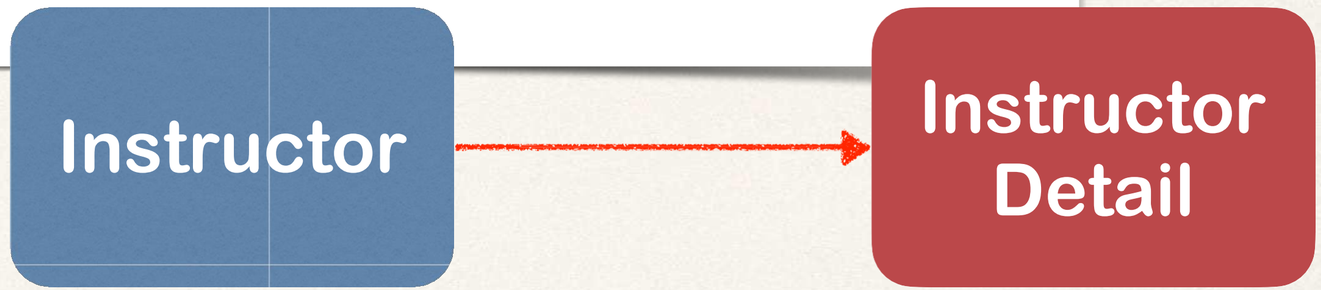
Step 4 - Creating Spring Boot - Command Line App

- We will create a Spring Boot - Command Line App
- This will allow us to focus on JPA / Hibernate
- Leverage our DAO pattern as in previous videos



Define DAO interface

```
import com.  .cruddemo.entity.Instructor;

public interface AppDAO {

    void save(Instructor theInstructor);

}
```

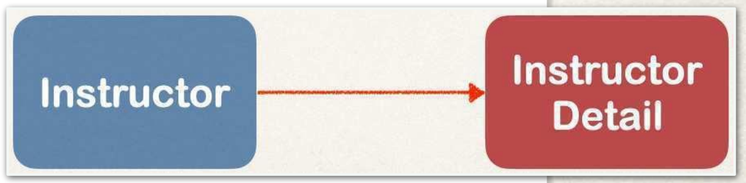


Define DAO implementation

```
import com. .cruddemo.entity.Instructor;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
public class AppDAOImpl implements AppDAO {

    // define field for entity manager
    private EntityManager entityManager;

    // inject entity manager using constructor injection
    @Autowired
    public AppDAOImpl(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @Override
    @Transactional
    public void save(Instructor theInstructor) {
        entityManager.persist(theInstructor);
    }

}
```

Inject the Entity Manager

Save the Java object

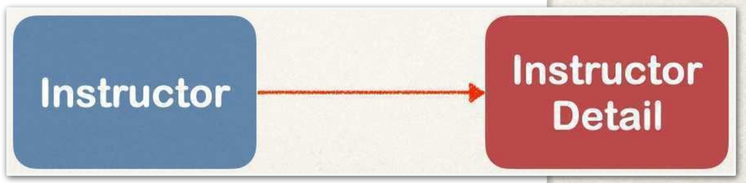


Define DAO implementation

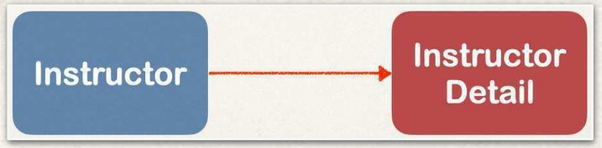
```
import com. .cruddemo.entity.Instructor;
import jakarta.persistence.EntityManager;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
public class AppDAOImpl implements AppDAO {

    // define field for entity manager
    private EntityManager entityManager;

    // inject entity manager using constructor injection
    @Autowired
    public AppDAOImpl(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @Override
    @Transactional
    public void save(Instructor theInstructor) {
        entityManager.persist(theInstructor);
    }

}
```

This will ALSO save the details object

Because of CascadeType.ALL



Update main app




```
@SpringBootApplication
public class MainApplication {

    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class, args);
    }

    @Bean
    public CommandLineRunner commandLineRunner(AppDAO appDAO) {
        return runner -> {

            createInstructor(appDAO);
    }

    ...
```

Inject the AppDAO

```
    private void createInstructor(AppDAO appDAO) {

        // create the instructor
        Instructor tempInstructor =
                new Instructor("Chad", "Darby", "darby@ .com");

        // create the instructor detail
        InstructorDetail tempInstructorDetail =
                new InstructorDetail(
                        "http:// /youtube", "Luv 2
                        code!!!");

        // associate the objects
        tempInstructor.setInstructorDetail(tempInstructorDetail);

        // save the instructor
        System.out.println("Saving instructor: " + tempInstructor);
        appDAO.save(tempInstructor);

        System.out.println("Done!");
    }
}
```

Remember:

This will ALSO save the details object

Because of CascadeType.ALL

In AppDAO, delegated to entityManager.persist(…)

JPA / Hibernate
One-to-One: Find an entity

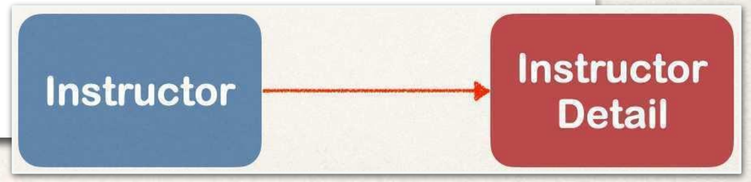
Define DAO implementation

```
@Repository
public class AppDAOImpl implements AppDAO {

    ...

    @Override
    public Instructor findInstructorById(int theId) {
        return entityManager.find(Instructor.class, theId);
    }

}
```

We'll add supporting code in the video: interface, main app

This will ALSO retrieve the instructor details object

Because of default behavior of @OneToOne fetch type is eager … more on fetch types later

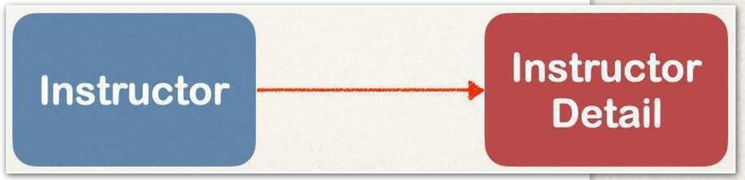


JPA / Hibernate
One-to-One: Delete an entity

Define DAO implementation

```
@Repository
public class AppDAOImpl implements AppDAO {
    ...

    @Override
    @Transactional
    public void deleteInstructorById(int theId) {

        // retrieve the instructor
        Instructor tempInstructor = entityManager.find(Instructor.class, theId);

        // delete the instructor
        entityManager.remove(tempInstructor);
    }

}
```

We'll add supporting code in the video: interface, main app



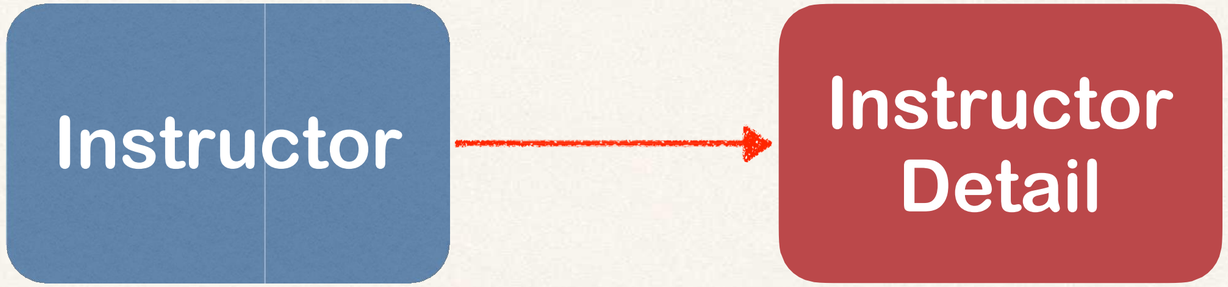
This will ALSO delete the instructor details object

Because of CascadeType.ALL

JPA / Hibernate
One-to-One: Bi-Directional

One-to-One Mapping

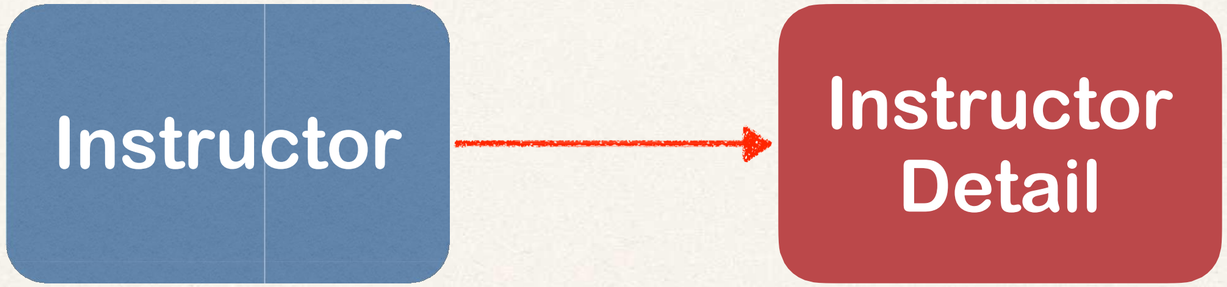
- We currently have a uni-directional mapping



New Use Case

- If we load an InstructorDetail
  - Then we'd like to get the associated Instructor

- Can't do this with current uni-directional relationship :-(



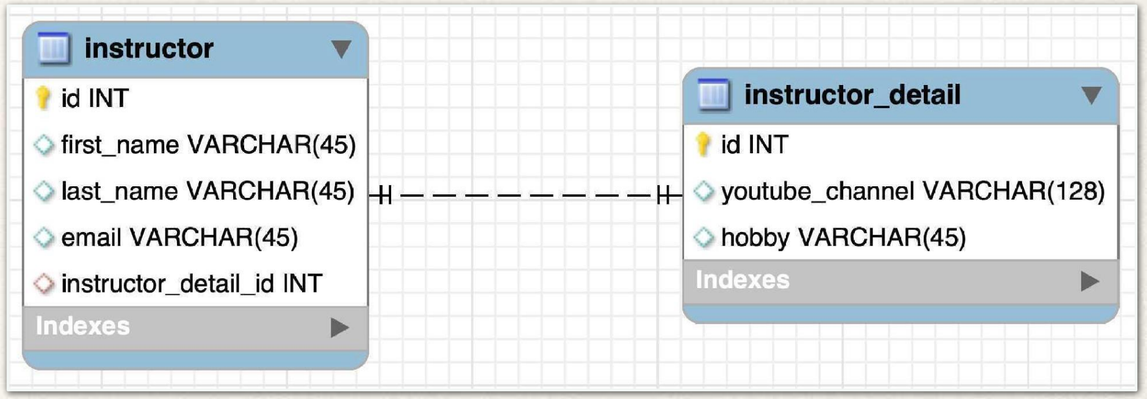
Bi-Directional

- Bi-Directional relationship is the solution

- We can start with InstructorDetail and make it back to the Instructor

Bi-Directional - The Good News

- To use Bi-Directional, we can keep the existing database schema
  - No changes required to database
- Simply update the Java code



Development Process: One-to-One (Bi-Directional)

Step-By-Step

1. Make updates to InstructorDetail class:
    1. Add new field to reference Instructor
    2. Add getter/setter methods for Instructor
    3. Add @OneToOne annotation
2. Create Main App

Step 1.1: Add new field to reference Instructor

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

    private Instructor instructor;

    ...

}
```

Step 1.2: Add getter/setter methods 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;
    }
    ...

}
```

Step 1.3: Add @OneToOne annotation

```
@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;
    }
    ...

}
```

Refers to “instructorDetail” property in “Instructor” class

More on mappedBy

```
public class InstructorDetail {
    ...

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

- **mappedBy** tells Hibernate
  - Look at the instructorDetail property in the Instructor class
  - Use information from the Instructor class @JoinColumn
  - To help find associated instructor

```
public class Instructor {
    ...

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

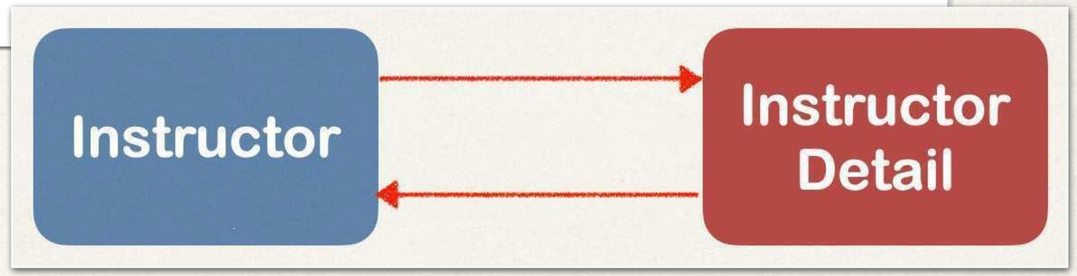
Add support for Cascading

Cascade all operations
to the associated Instructor

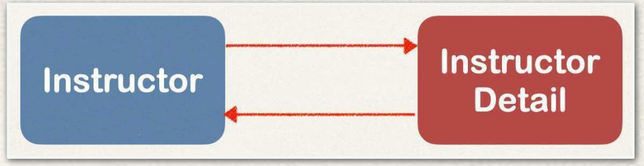
```
@Entity
@Table(name="instructor_detail")
public class InstructorDetail {
    ...

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

    public Instructor getInstructor() {
        return instructor;
    }

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

}
```

Define DAO interface

```
import com.  .cruddemo.entity.Instructor;

public interface AppDAO {

    InstructorDetail findInstructorDetailById(int theId);

}
```



Define DAO implementation



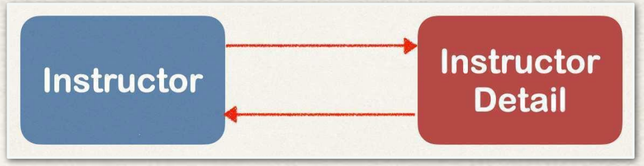

```
import com. .cruddemo.entity.InstructorDetail;
…

@Repository
public class AppDAOImpl implements AppDAO {

    // define field for entity manager
    private EntityManager entityManager;

    // inject entity manager using constructo  injection
    …

    @Override
    public InstructorDetail findInstructorDetailById(int theId) {

        return entityManager.find(InstructorDetail.class, theId);
    }

}
```

This will ALSO retrieve the instructor object

Because of default behavior of @OneToOne

Retrieve the InstructorDetail

Update main app




```
@SpringBootApplication
public class MainApplication {

    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class, args);
    }

    @Bean
    public CommandLineRunner commandLineRunner(AppDAO  appDAO) {
        return runner -> {

            findInstructorDetail(appDAO);
    }

    ...
```

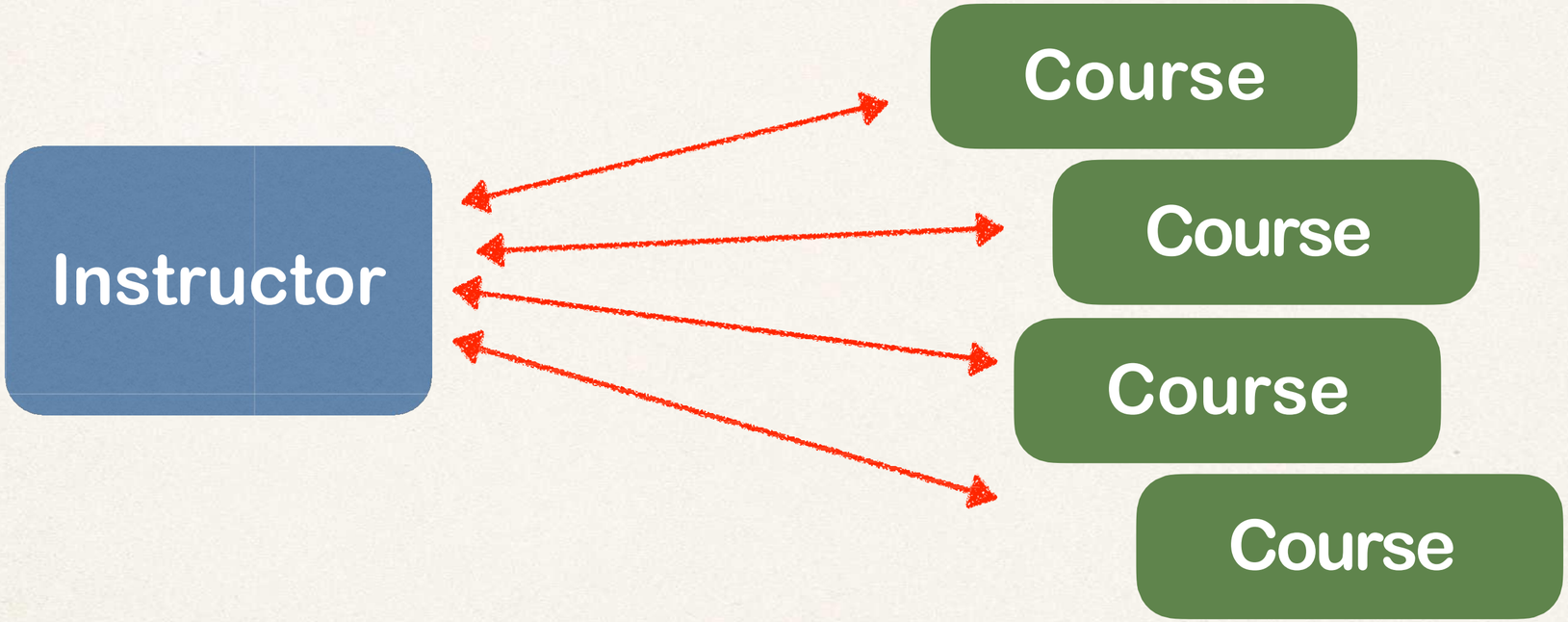
Inject the AppDAO

```
    private void findInstructorDetail(AppDAO appDAO) {

        int theId = 1;
        System.out.println("Finding instructor detail id: " + theId);

        InstructorDetail tempInstructorDetail = appDAO.findInstructorDetailById(theId);

        System.out.println("tempInstructorDetail: " + tempInstructorDetail);
        System.out.println("the associated instructor: " + tempInstructorDetail.getInstructor());

    }
}
```

JPA / Hibernate
One-to-Many

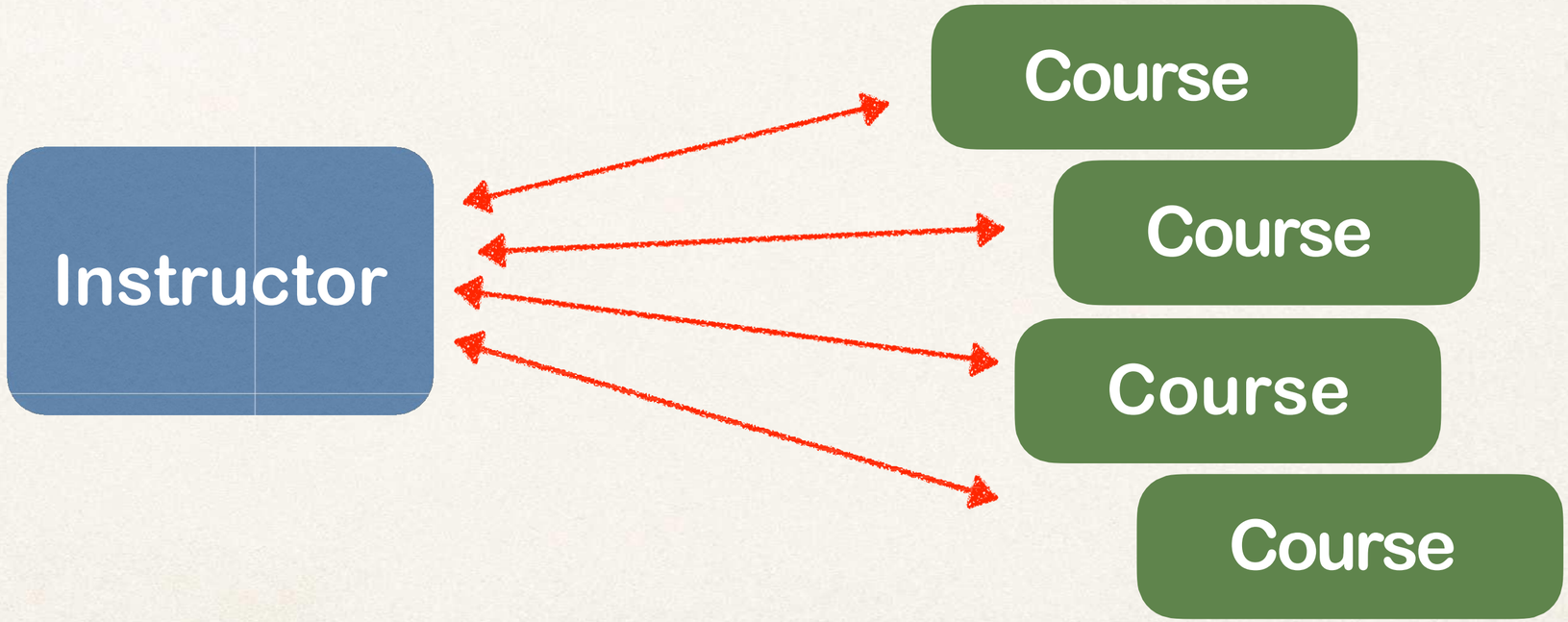
One-to-Many Mapping

- An instructor can have many courses
  - Bi-directional



Many-to-One Mapping

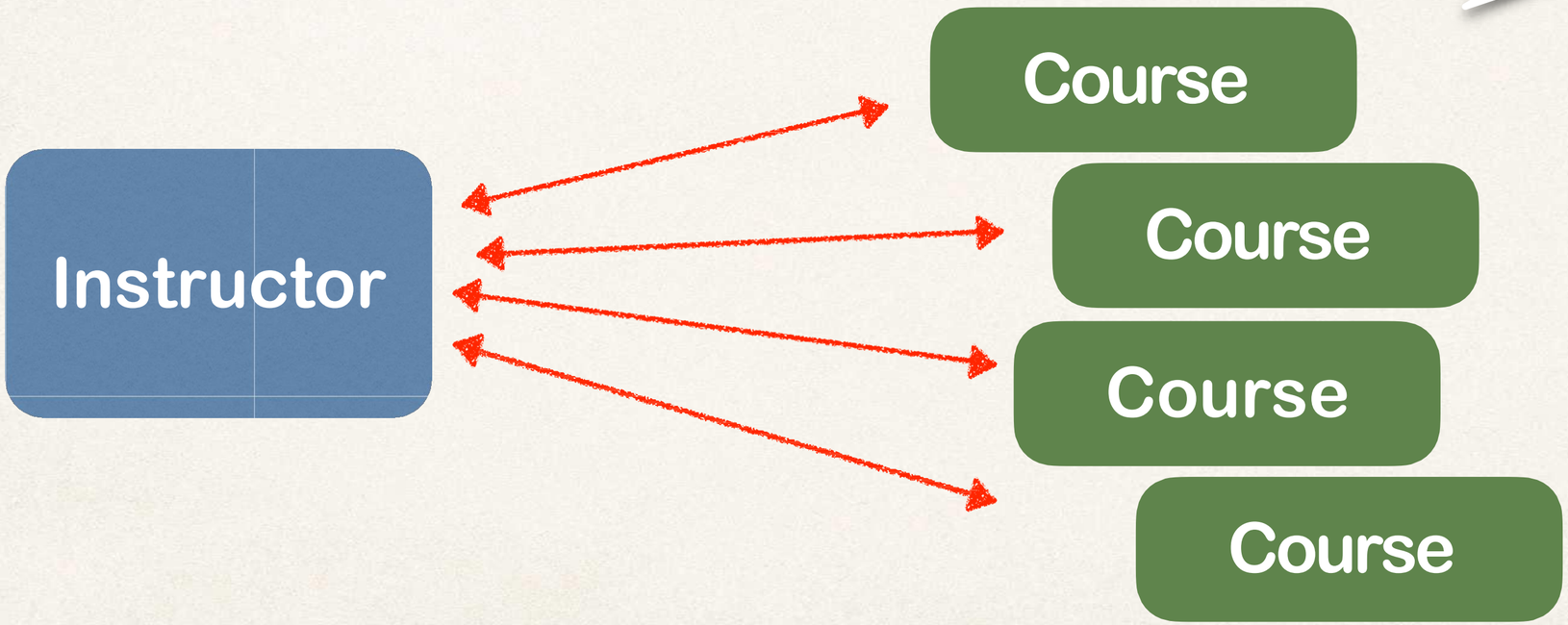
- Many courses can have one instructor
  - Inverse / opposite of One-to-Many



Real-World Project Requirement

- If you delete an instructor, DO NOT delete the courses
- If you delete a course, DO NOT delete the instructor

Do not apply cascading deletes!



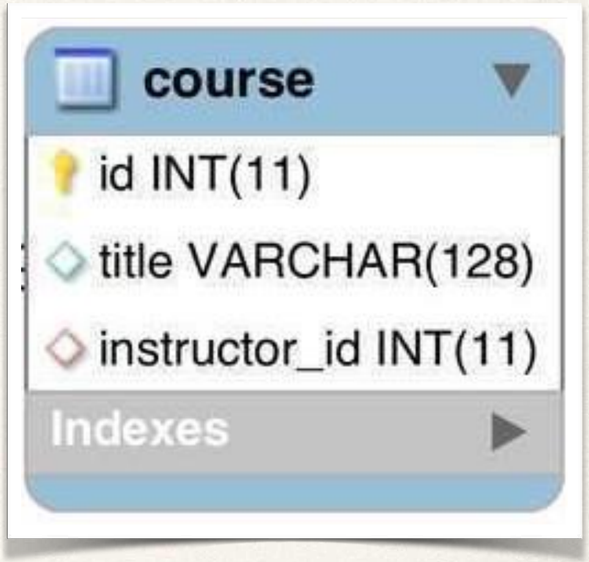
Development Process: One-to-Many

1. Prep Work - Define database tables
2. Create Course class
3. Update Instructor class
4. Create Main App

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`)
  …
);
```

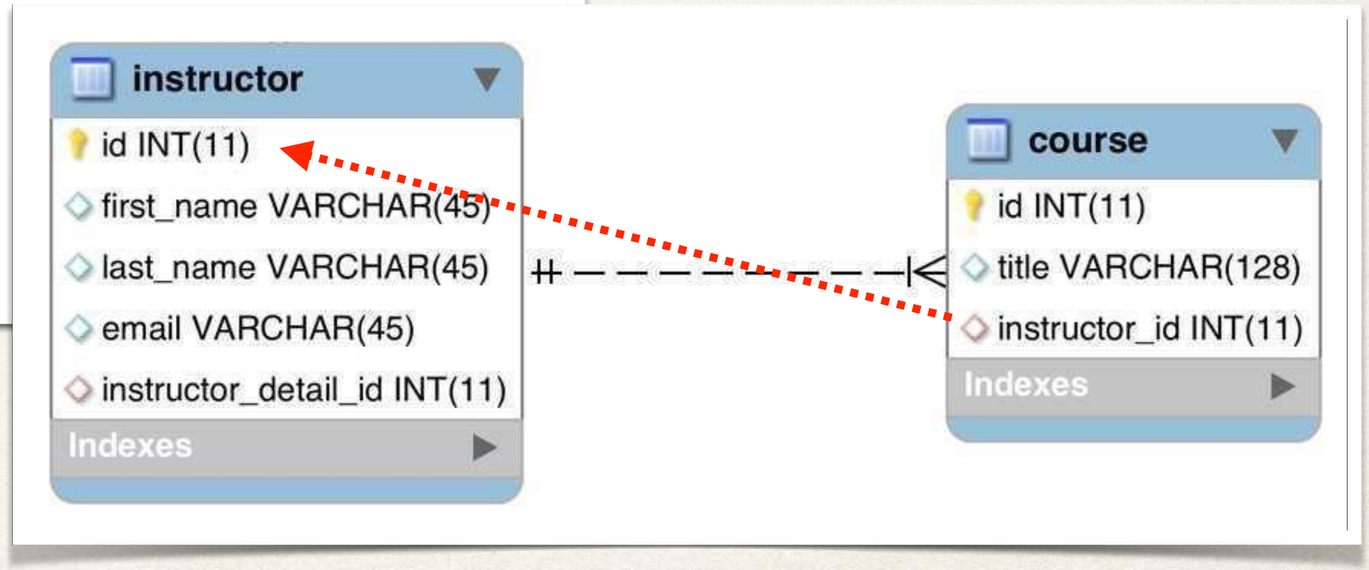
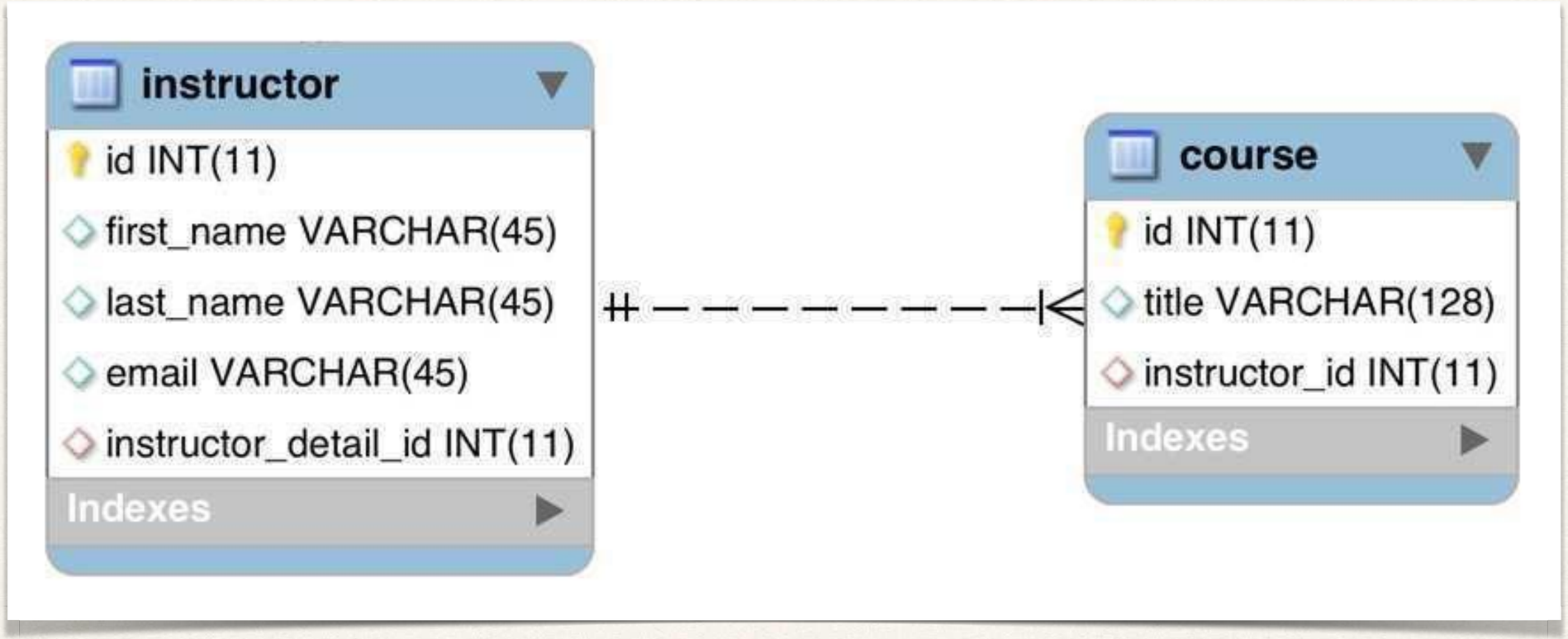


table: instructor - no changes

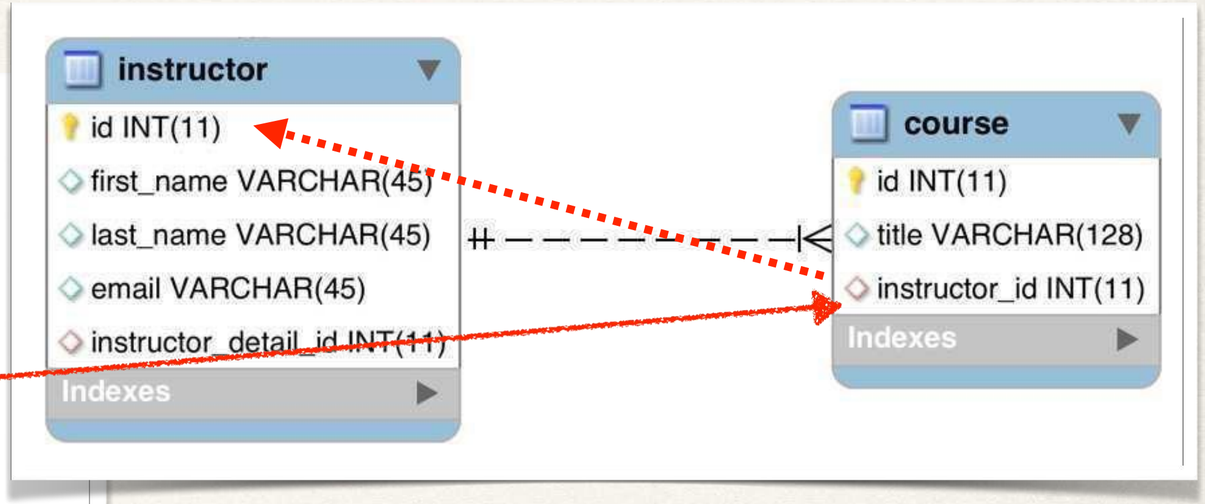


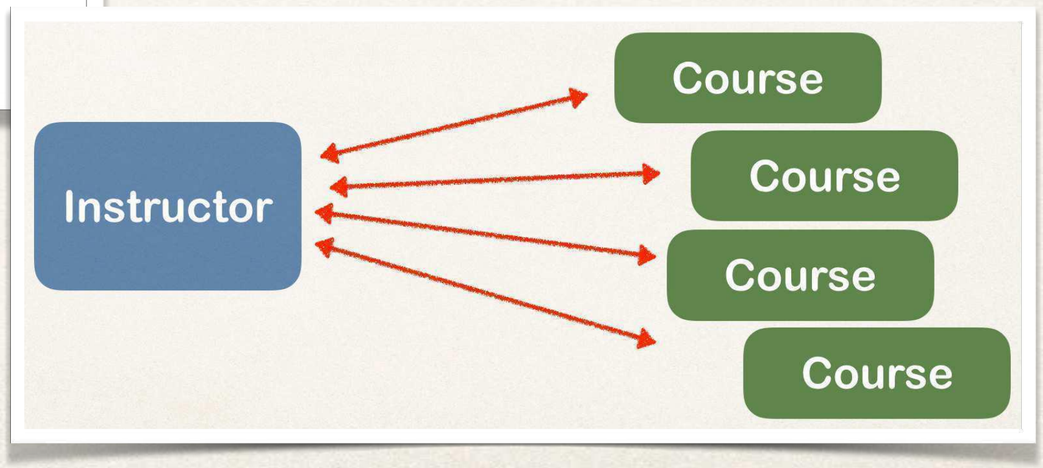
Step 2: Create Course class

```
@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
}
```

Step 2: Create Course class - @ManyToOne

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

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

    ...
    // constructors, getters / setters
}
```





Step 3: Update Instructor - reference courses

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

    private List<Course> courses;

    ...

}
```

Step 3: Update Instructor - reference courses

```
@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;
    }
    ...

}
```

Add @OneToMany annotation

```
@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;
    }

    ...

}
```

Refers to “instructor” property in “Course” class

More: mappedBy

```
public class Instructor {
    ...

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

- **mappedBy** tells Hibernate
  - Look at the instructor property in the Course class
  - Use information from the Course class @JoinColumn
  - To help find associated courses for instructor

```
public class Course {
    ...

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

Add support for Cascading

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

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



    ...

}
```

Do not apply cascading deletes!

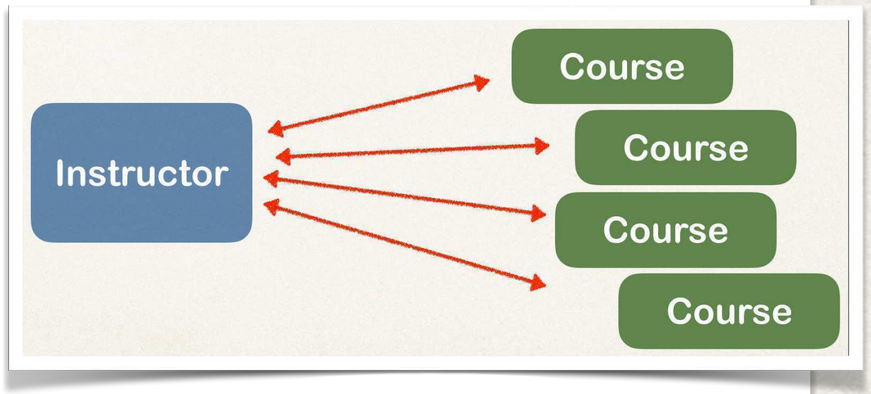
Add support for Cascading

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

    @ManyToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE
                        CascadeType.DETACH, CascadeType.REFRESH})
    @JoinColumn(name="instructor_id")
    private Instructor instructor;


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

Do not apply cascading deletes!

Add convenience methods for bi-directional

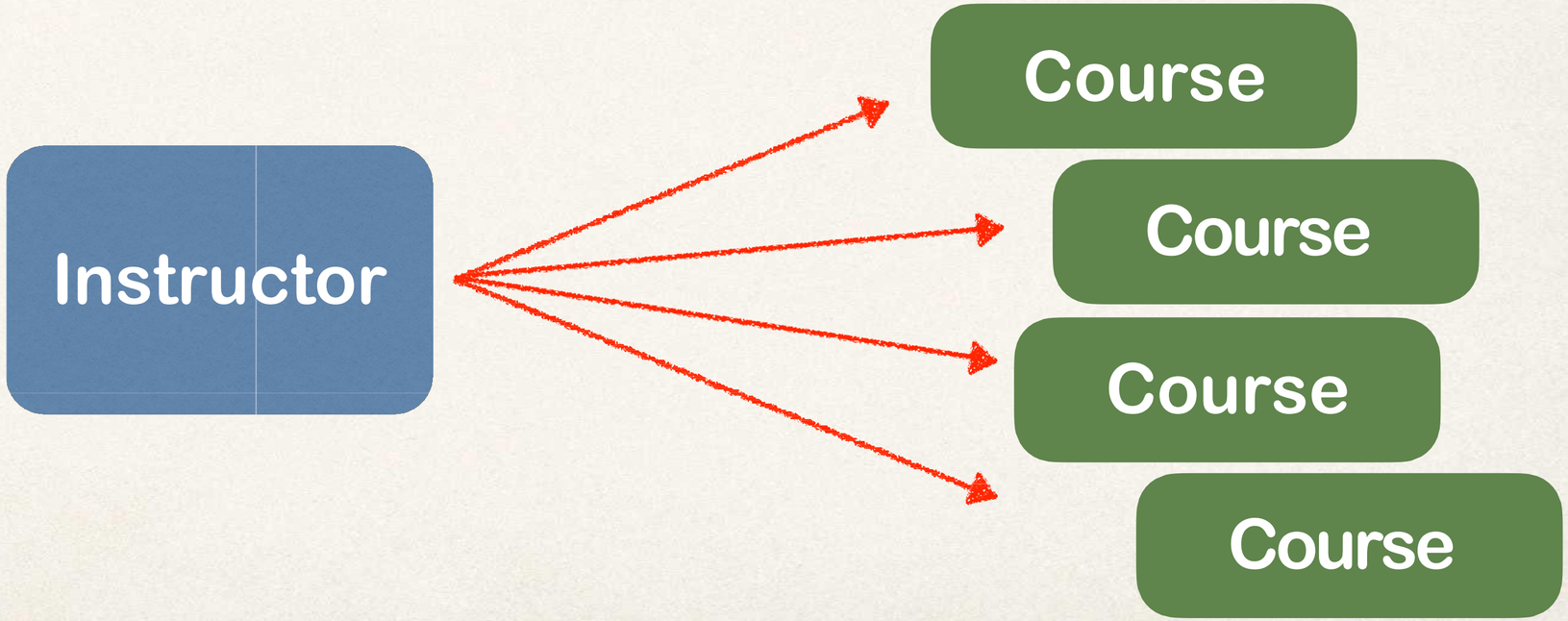
```
@Entity
@Table(name="instructor")
public class Instructor {
    ...
    // add convenience methods for bi-directional relationship

    public void add(Course tempCourse) {

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

        courses.add(tempCourse);

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



Fetch Types: Eager vs Lazy

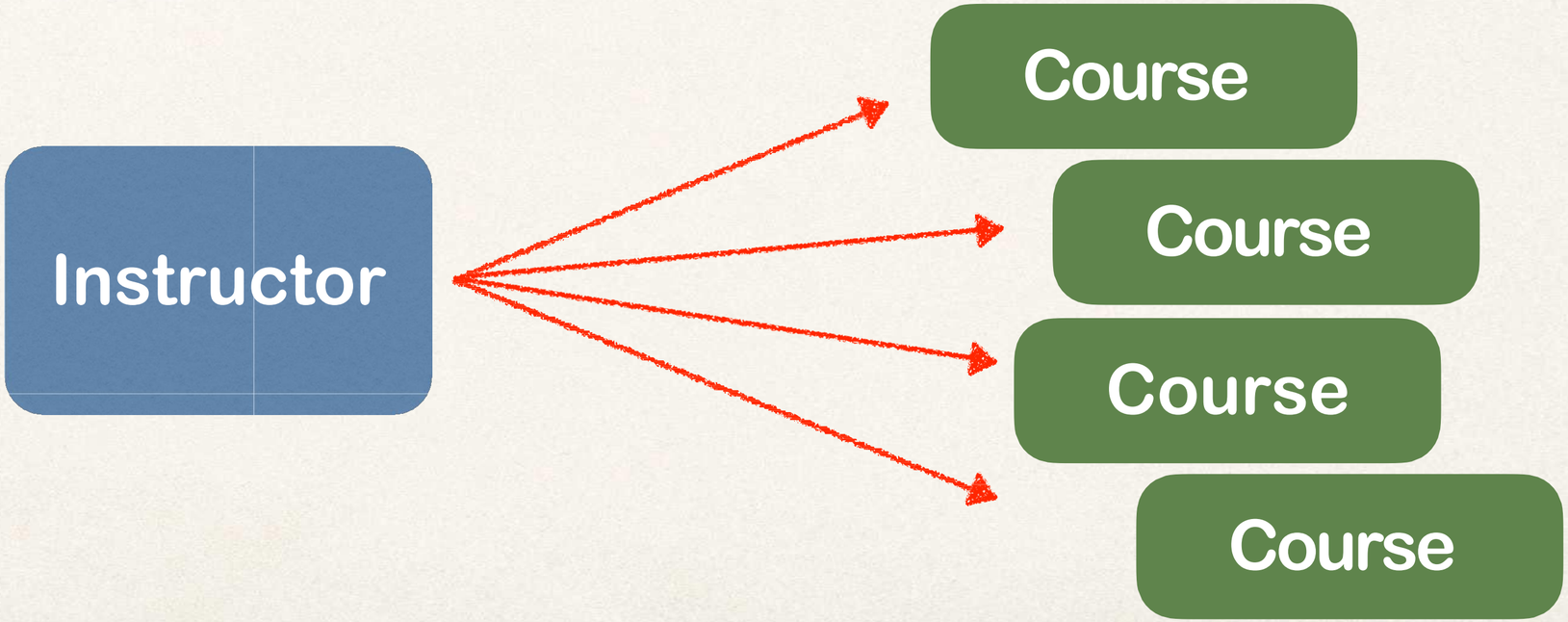
Fetch Types: Eager vs Lazy Loading

- When we fetch / retrieve data, should we retrieve EVERYTHING?
  - **Eager** will retrieve everything
  - **Lazy** will retrieve on request



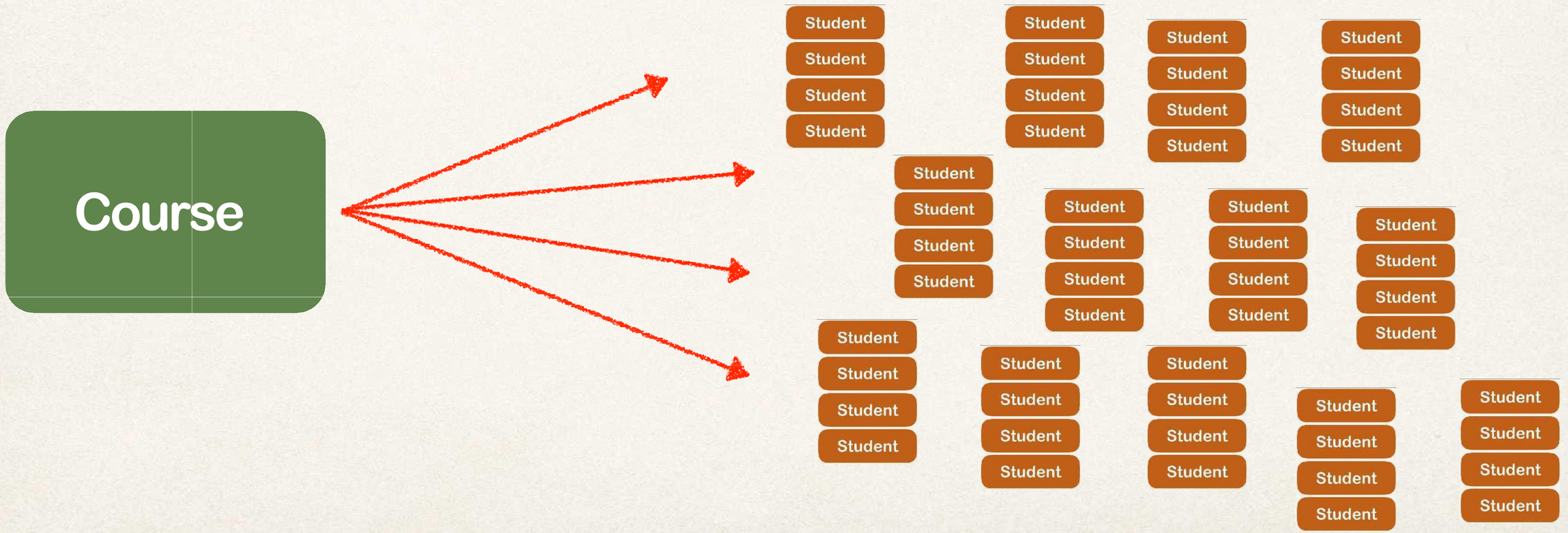
Eager Loading

- Eager loading will load all dependent entities
  - Load instructor and all of their courses at once



Eager Loading

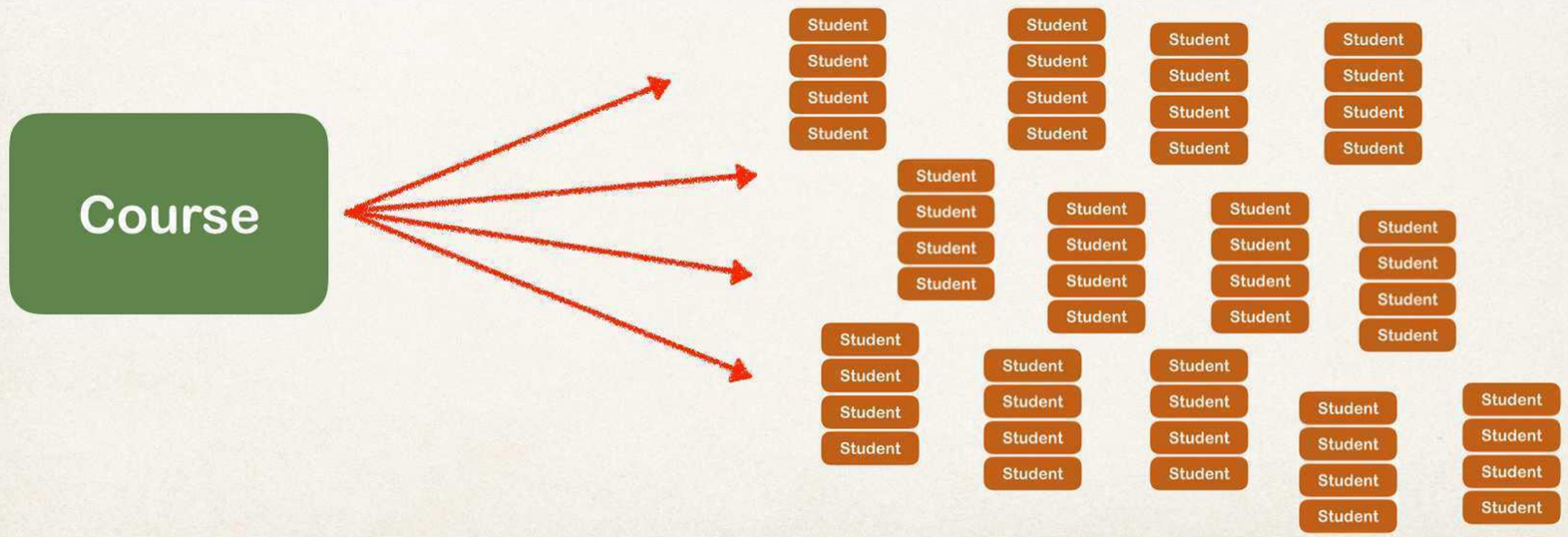
- What about course and students?
  - Could easily turn into a performance nightmare …



Eager Loading

- In our app, if we are searching for a course by keyword
  - Only want a list of matching courses

- Eager loading would still **load all students** for **each course** …. not good!



Best Practice

Only load data when absolutely needed

Prefer Lazy loading
instead of
Eager loading

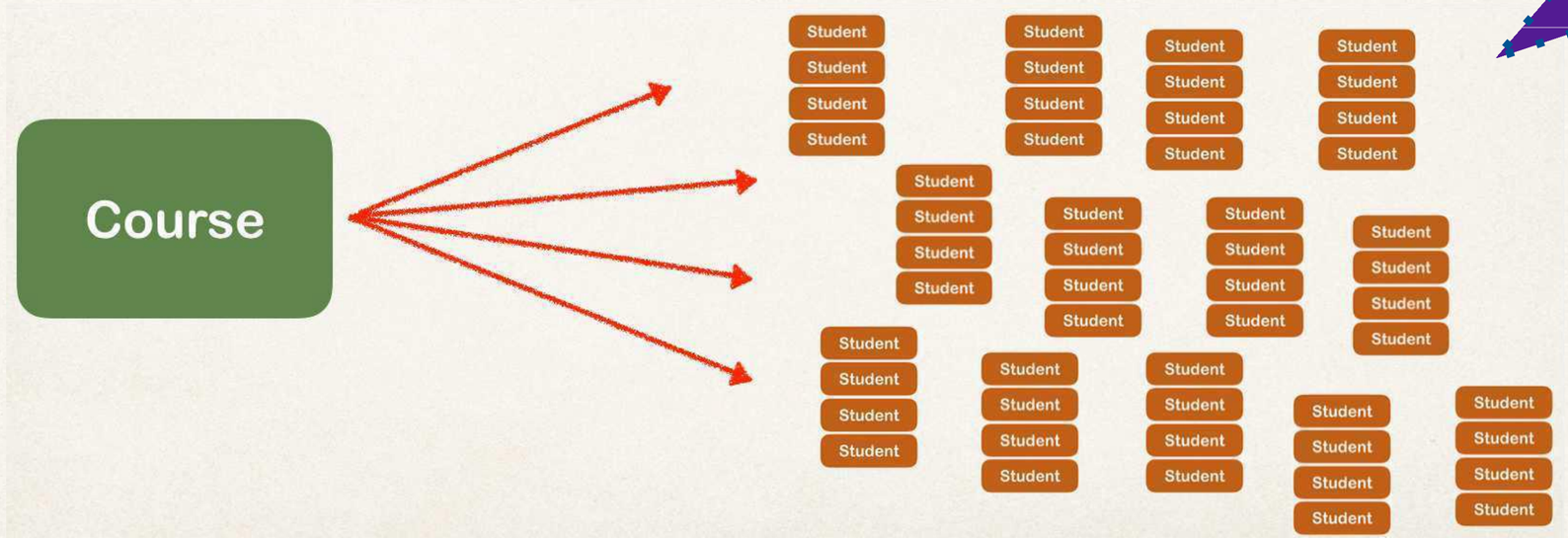
Lazy Loading

- Lazy loading will load the main entity first
  - Load dependent entities on demand (lazy)

Lazy Loading is preferred

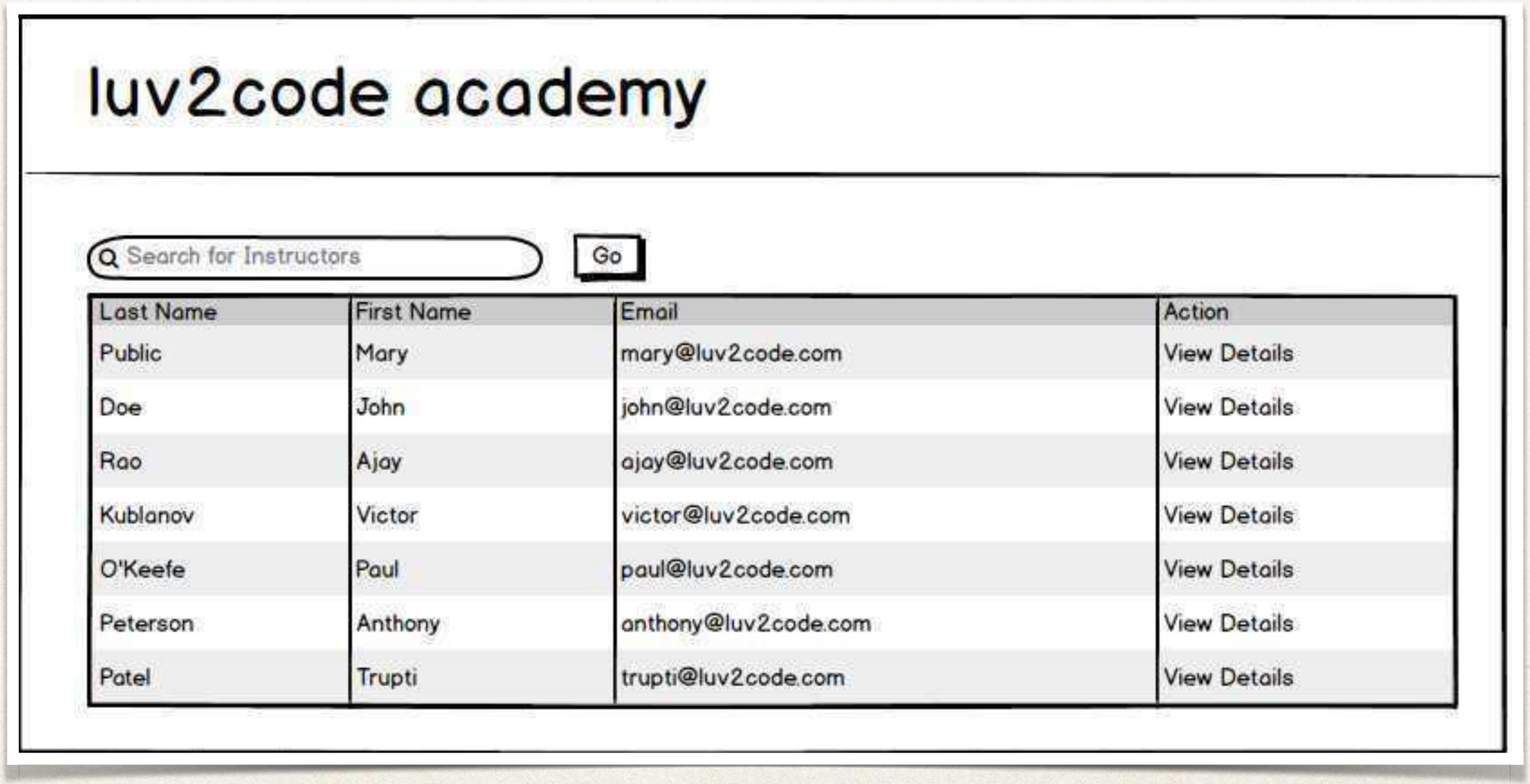
Load on demand (lazy)

Load course first



Real-World Use Case

- Search for instructors



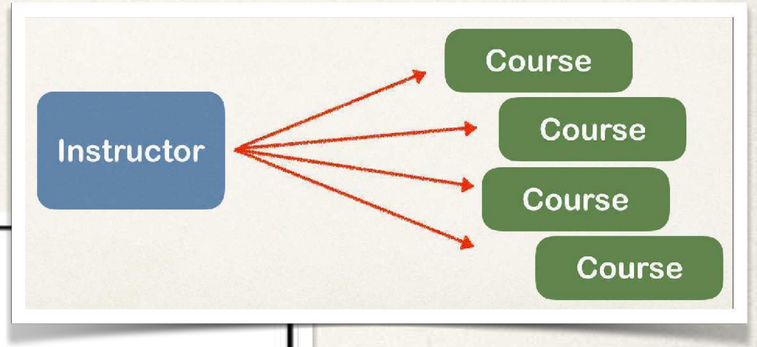
Real-World Use Case

- In Master view, use lazy loading

- In Detail view, retrieve the entity and necessary dependent entities

Real-World Use Case - Master View

- In Master view, use lazy loading for search results
- Only load instructors … not their courses



luv2code academy

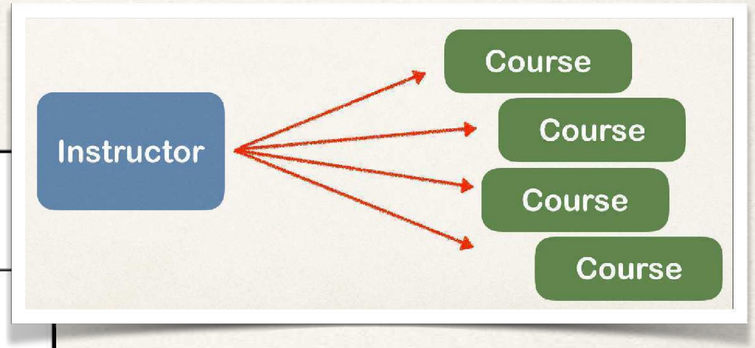
Search for Instructors

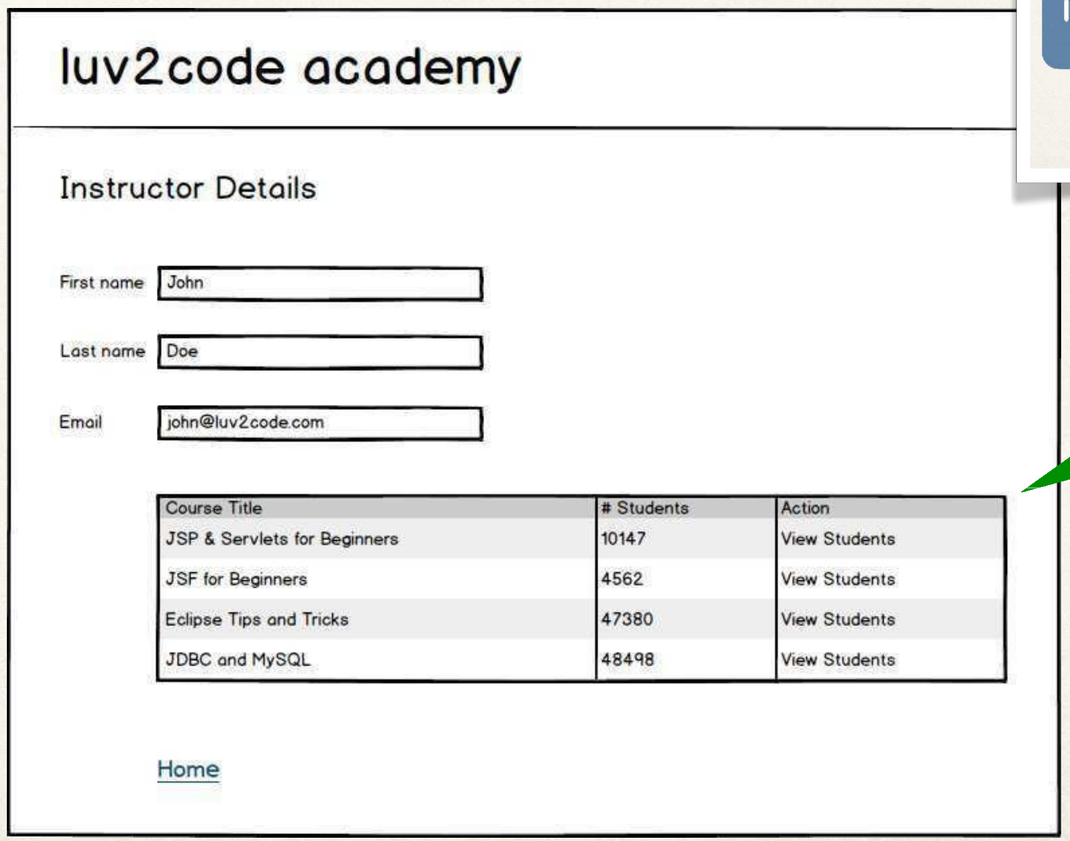
Go

Last Name	First Name	Email	Action
Public	Mary	mary@luv2code.com	View Details
Doe	John	john@luv2code.com	View Details
Rao	Ajay	ajay@luv2code.com	View Details
Kublanov	Victor	victor@luv2code.com	View Details
O'Keefe	Paul	paul@luv2code.com	View Details
Peterson	Anthony	anthony@luv2code.com	View Details
Patel	Trupti	trupti@luv2code.com	View Details

Real-World Use Case - Detail View

- In Detail view, retrieve the entity and necessary dependent entities
- Load instructor AND their courses

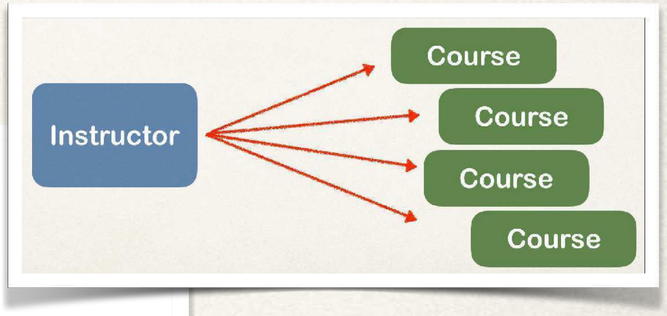




List of courses

Fetch Type

- When you define the mapping relationship
  - You can specify the fetch type: EAGER or LAZY




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

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

   ...

}
```

Default Fetch Types

Mapping	Default Fetch Type
@OneToOne	FetchType.EAGER
@OneToMany	FetchType.LAZY
@ManyToOne	FetchType.EAGER
@ManyToMany	FetchType.LAZY

Overriding Default Fetch Type

- Specifying the fetch type, overrides the defaults

Mapping	Default Fetch Type
@ManyToOne	FetchType.EAGER

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

More about Lazy Loading

- When you lazy load, the data is only retrieved on demand

- However, this requires an open Hibernate session
  - need an connection to database to retrieve data

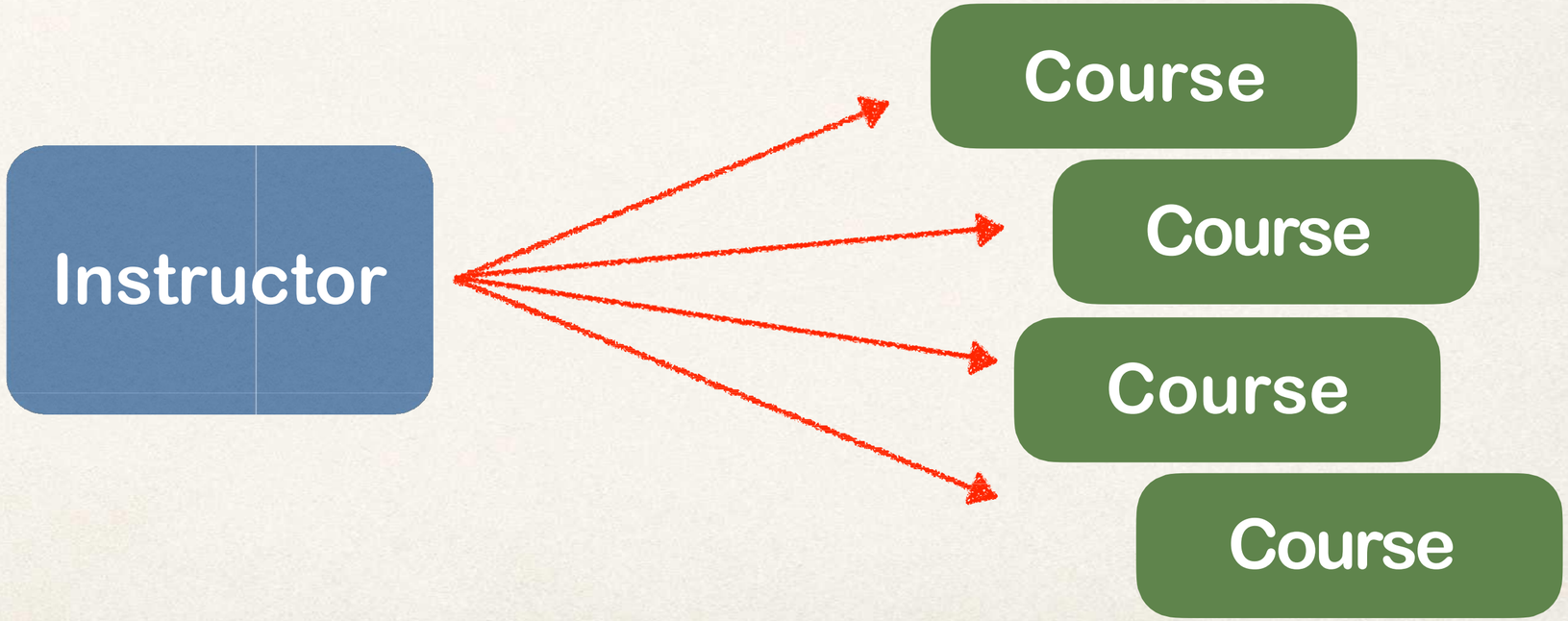
More about Lazy Loading

- If the Hibernate session is closed
  - And you attempt to retrieve lazy data
  - Hibernate will throw an exception

Lazy: Find Courses

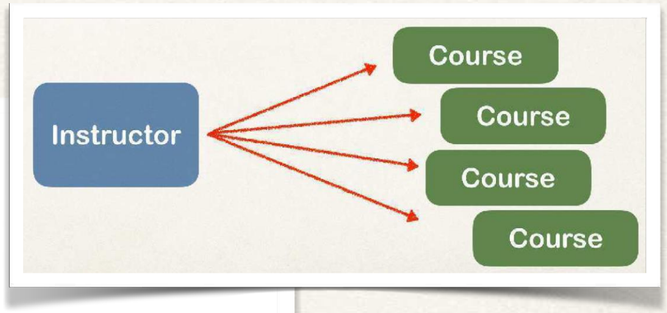
Previous Solution: Eager

- **Eager** will retrieve everything … all of the courses for an instructor
- But we may not want this ALL of the time
- We'd like the option to load courses as needed …



Fetch Type

- Change the fetch type back to LAZY




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

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

   ...

}
```

FetchType for @OneToMany defaults to lazy …
But I will explicitly list it for readability

Add new method to find courses for instructor

File: AppDAOImpl.java

```
@Override
public List<Course> findCoursesByInstructorId(int theId) {

    // create query
    TypedQuery<Course> query = entityManager.createQuery("from Course where instructor.id = :data", Course.class);
    query.setParameter("data", theId);

    // execute query
    List<Course> courses = query.getResultList();

    return courses;
}
```

File: CruddemoApplication.java

```
private void findCoursesForInstructor(AppDAO appDAO) {

    int theId = 1;

    // find the instructor
    Instructor tempInstructor = appDAO.findInstructorById(theId);
    System.out.println("tempInstructor: " + tempInstructor);

    // find courses for instructor
    List<Course> courses = appDAO.findCoursesByInstructorId(theId);

    // associate the objects
    tempInstructor.setCourses(courses);

    System.out.println("the associated courses: " + tempInstructor.getCourses());
}
```

Since fetch type for courses is lazy

This will retrieve the instructor WITHOUT courses

Lazy: Find Instructor with Courses

Previous Solution: Find Courses for Instructor

- Previous solution was OK … but …
- Required an extra query
- I wish we could have a new method that would
  - Get instructor AND courses … in a single query
  - Also keep the LAZY option available … don't change fetch type

Add new method to find instructor with courses

File: AppDAOImpl.java

```
@Override
public Instructor findInstructorByIdJoinFetch(int theId) {

    // create query
    TypedQuery<Instructor> query = entityManager.createQuery(
                                        "select i from Instructor i "
                                            + "JOIN FETCH i.courses "
                                            + "where i.id = :data", Instructor.class);

    query.setParameter("data", theId);

    // execute query
    Instructor instructor = query.getSingleResult();

    return instructor;
}
```

Even with Instructor
@OneToMany(fetchType=LAZY)

This code will still retrieve Instructor AND Courses

The JOIN FETCH is similar to EAGER loading

This code will still retrieve Instructor AND Courses

File: CruddemoApplication.java

```
private void findInstructorWithCoursesJoinFetch(AppDAO appDAO) {

    int theId = 1;

    // find the instructor
    System.out.println("Finding instructor id: " + theId);
    Instructor tempInstructor = appDAO.findInstructorByIdJoinFetch(theId);

    System.out.println("tempInstructor: " + tempInstructor);
    System.out.println("the associated courses: " + tempInstructor.getCourses());

    System.out.println("Done!");
}
```

We have options now

- If you only need Instructor … and no courses, then call
  - appDAO.findInstructorById(…)
- If you need Instructor AND courses, then call
  - appDAO.findInstructorByIdJoinFetch(…)
- This gives us flexibility instead of having EAGER fetch hard-coded

@OneToMany: Update Instructor

Update Instructor

- Find an instructor by ID
- Change the instructor's data by calling setter method(s)
- Update the instructor using the DAO

Add new DAO method to update instructor

File: AppDAOImpl.java

```
@Override
@Transactional
public void update(Instructor  tempInstructor)  {
    entityManager.merge(tempInstructor);
}
```

merge(…) will update an existing entity

Main app

File: CruddemoApplication.java

```
private void updateInstructor(AppDAO appDAO) {

  int theId = 1;

  System.out.println("Finding instructor id: " + theId);
  Instructor tempInstructor = appDAO.findInstructorById(theId);

  System.out.println("Updating instructor id: " + theId);
  tempInstructor.setLastName("TESTER");

  appDAO.update(tempInstructor);

  System.out.println("Done");
}
```

Change instructor's data

Call DAO method to update database

@OneToMany: Update Course

Update Course

- Find a course by ID
- Change the course's data by calling setter method(s)
- Update the course using the DAO

Add new DAO method to update course

File: AppDAOImpl.java

```
@Override
@Transactional
public void update(Course  tempCourse)  {
    entityManager.merge(tempCourse);
}
```

merge(…) will update an existing entity

Main app

File: CruddemoApplication.java

```
private void updateCourse(AppDAO appDAO) {

  int theId = 10;

  System.out.println("Finding course id: " + theId);
  Course tempCourse = appDAO.findCourseById(theId);

  System.out.println("Updating course id: " + theId);
  tempCourse.setTitle("Enjoy the Simple Things");

  appDAO.update(tempCourse);

  System.out.println("Done");
}
```

Change course's data

Call DAO method
to update database

@OneToMany: Delete Instructor

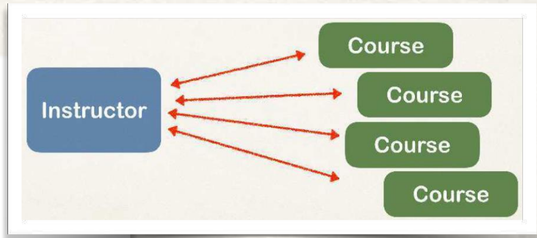
Delete instructor

- Find an instructor by ID
- Break association of all instructor's courses
- Delete the instructor

Add new DAO method to delete instructor

File: AppDAOImpl.java

```
@Override
@Transactional
public void deleteInstructorById(int  theId) {

    // retrieve the instructor
    Instructor tempInstructor = entityManager.find(Instructor.class, theId);

    List<Course> courses = tempInstructor.getCourses();

    // break associations of all courses for instructor
    for (Course tempCourse : courses) {
        tempCourse.setInstructor(null);
    }

    // delete the instructor
    entityManager.remove(tempInstructor);
}
```



Remove the instructor from the courses

We only delete the instructor … not the associated course based on our cascade types

Error message

- If you don't remove instructor from courses … **constraint violation**

```
Caused by: java.sql.SQLIntegrityConstraintViolationException:

Cannot delete or update a parent row: a foreign key constraint fails

(`hb-03-one-to-many`.`course`,
CONSTRAINT `FK_INSTRUCTOR` FOREIGN KEY (`instructor_id`) REFERENCES `instructor` (`id`))
```

- An instructor can not be deleted if it is referenced by a course
- You must remove the instructor from the course first

Main app

File: CruddemoApplication.java

```
private void deleteInstructor(AppDAO  appDAO)  {

  int theId = 1;
  System.out.println("Deleting instructor id: " + theId);

  appDAO.deleteInstructorById(theId);

  System.out.println("Done!");
}
```

@OneToMany: Delete Course

Delete course

- Delete the course by ID

Add new DAO method to delete course

File: AppDAOImpl.java

```
@Override
@Transactional
public void deleteCourseById(int  theId) {

    // retrieve the course
    Course tempCourse = entityManager.find(Course.class,  theId);

    // delete the course
    entityManager.remove(tempCourse);
}
```

Main app

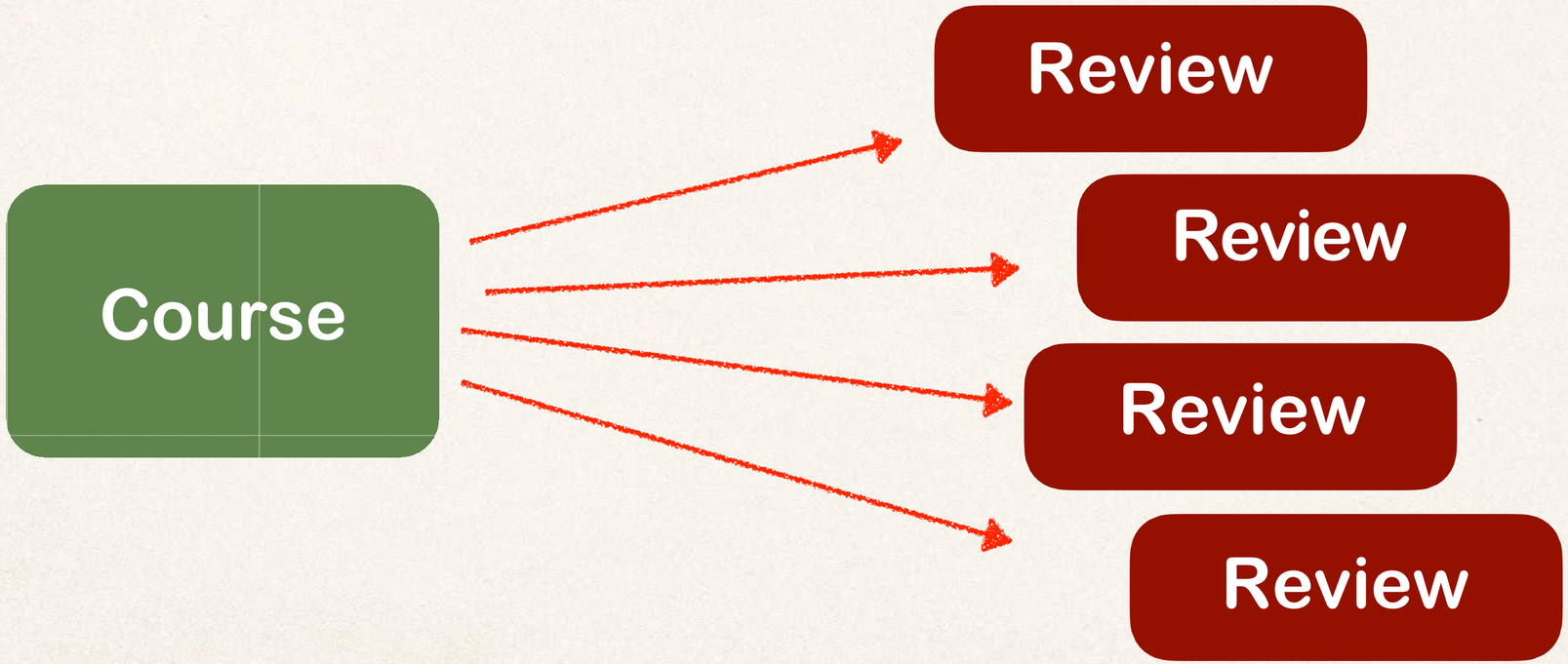
File: CruddemoApplication.java

```
private void deleteCourseById(AppDAO  appDAO)  {

  int theId = 10;
  System.out.println("Deleting course id: " + theId);

  appDAO.deleteCourseById(theId);

  System.out.println("Done!");

}
```

@OneToMany: Uni-Directional

One-to-Many Mapping

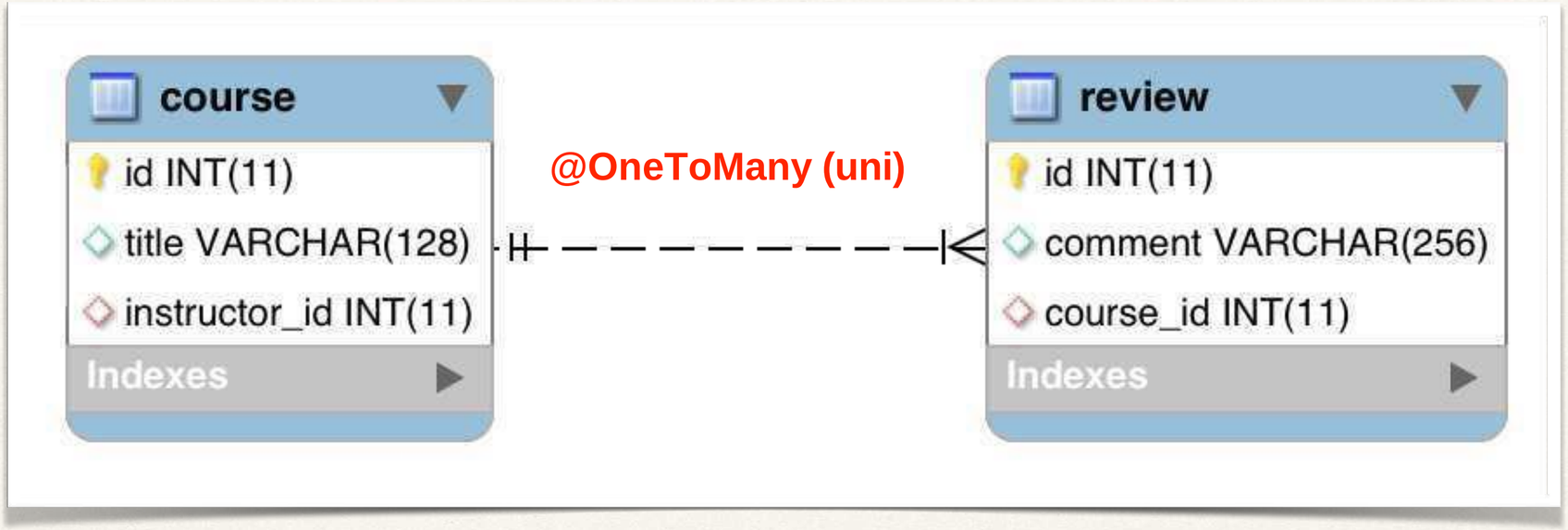
- A course can have many reviews
  - Uni-directional



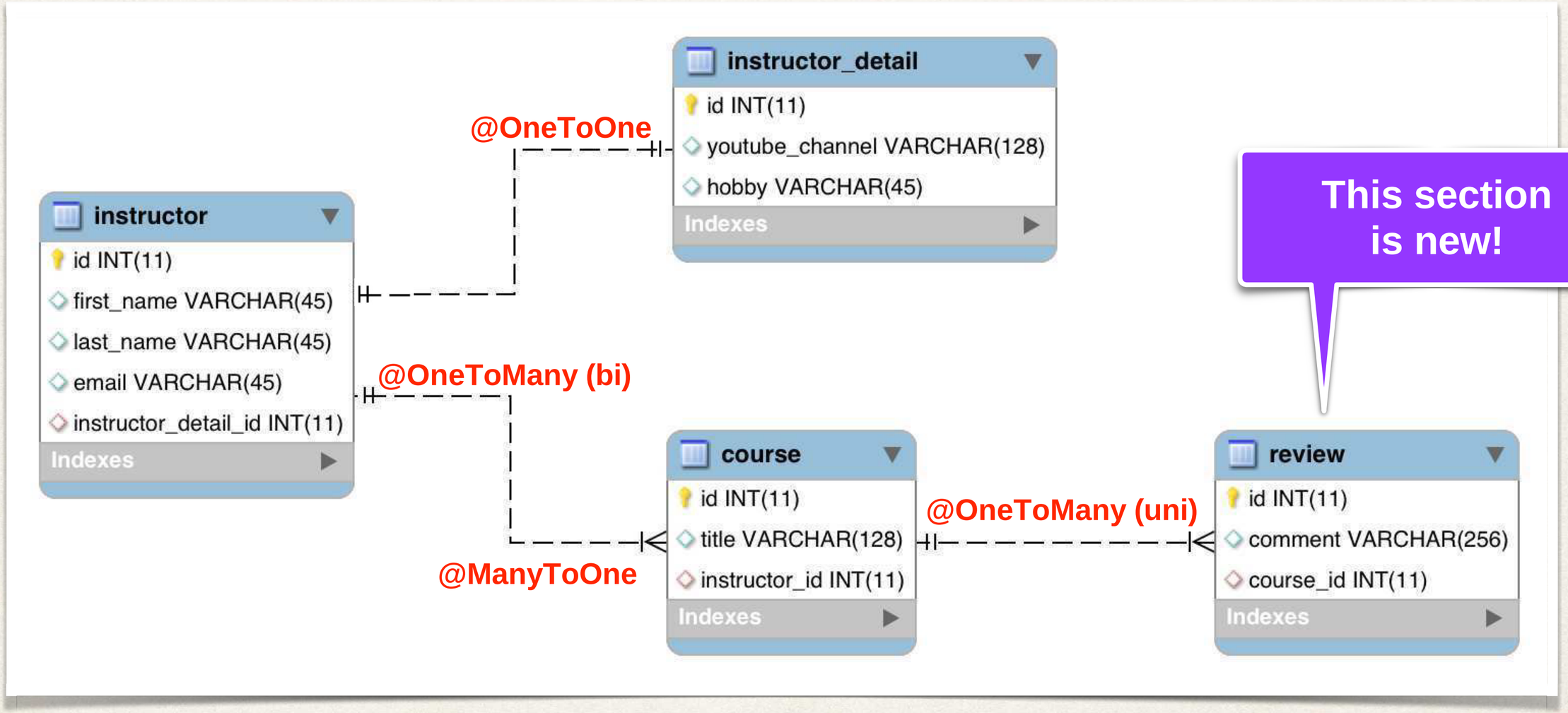
Real-World Project Requirement

- If you delete a course, also delete the reviews
- Reviews without a course … have no meaning

@OneToMany



Look Mom … our project is growing!



Development Process: One-to-Many

Step-By-Step

1. Prep Work - Define database tables
2. Create **Review** class
3. Update **Course** class

table: review

File: create-db.sql

```
CREATE TABLE `review` (

  `id` int(11) NOT NULL AUTO_INCREMENT,
  `comment` varchar(256) DEFAULT NULL,
  `course_id` int(11) DEFAULT NULL,

   ...
);
```

comment:
"Wow … this course is awesome!"

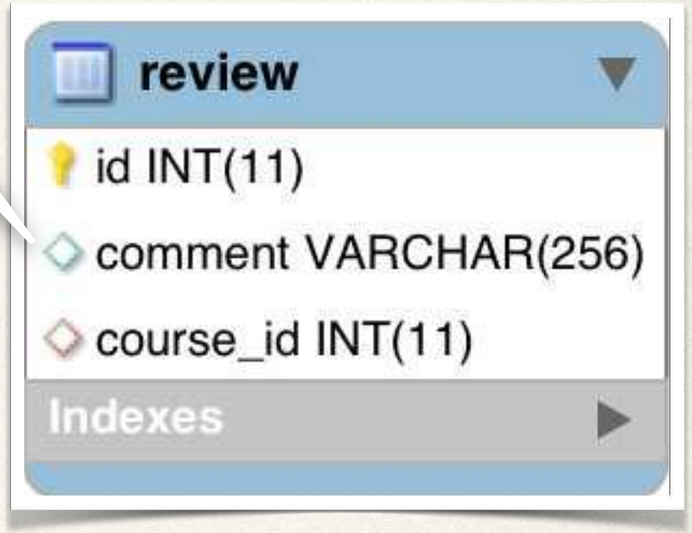


table: review - foreign key

File: create-db.sql

```
CREATE TABLE `review` (
  …
  KEY `FK_COURSE_ID_idx` (`course_id`),
  CONSTRAINT `FK_COURSE`
  FOREIGN KEY (`course_id`)
  REFERENCES `course` (`id`)
  …
);
```

Table

Column

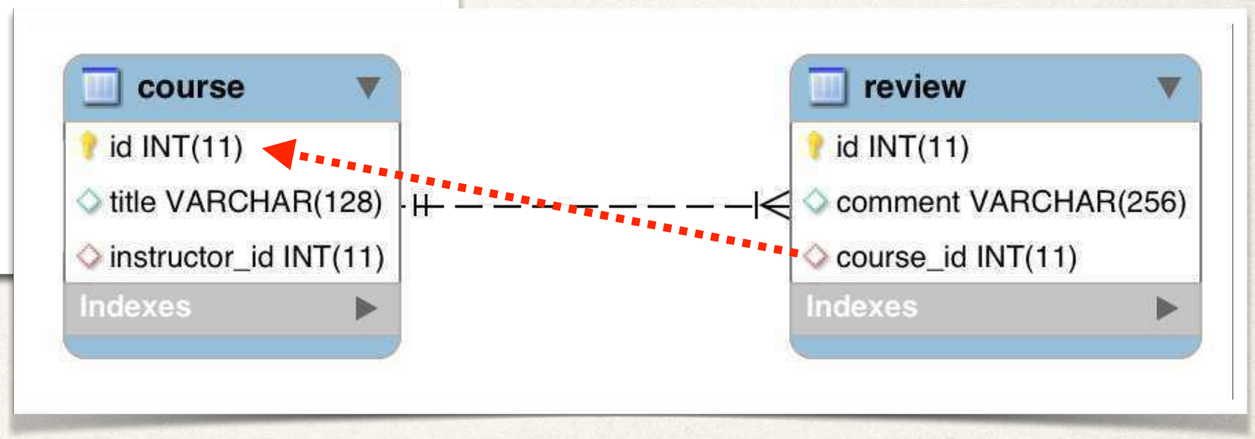
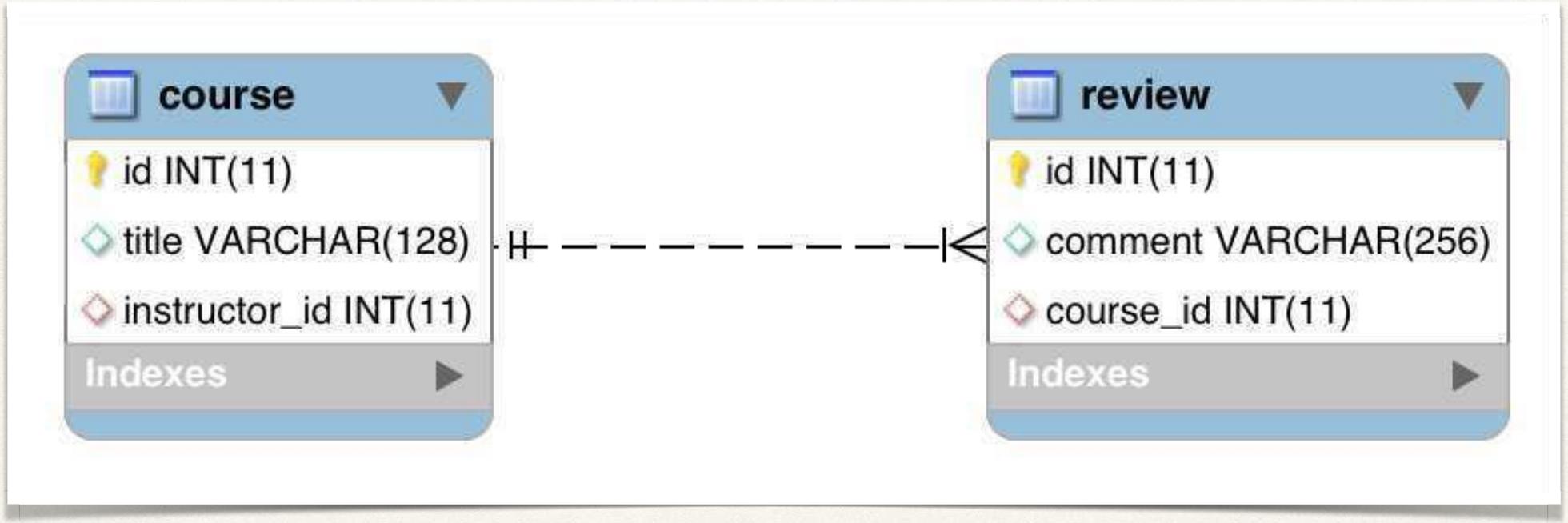
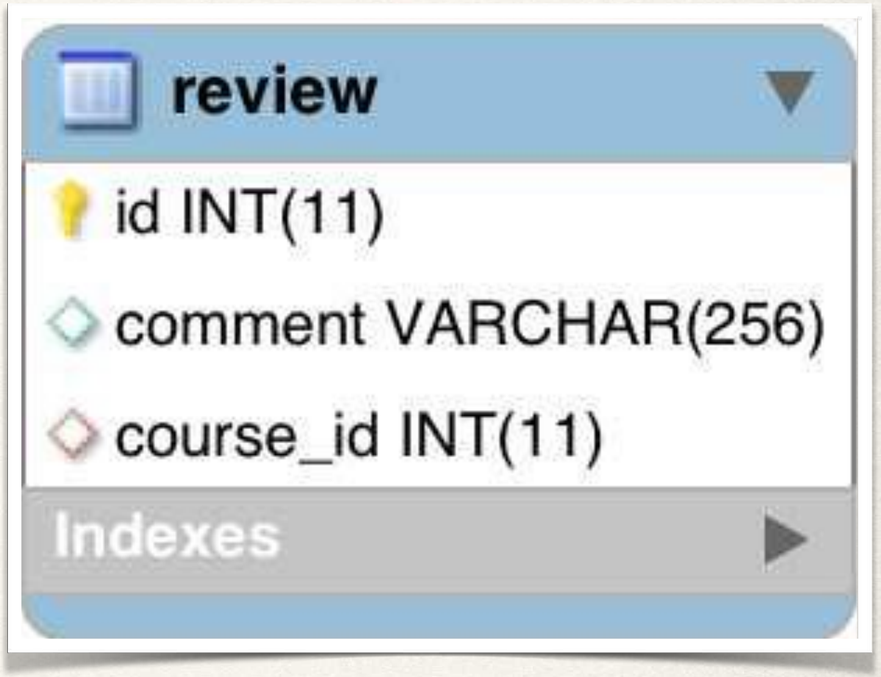


table: course - no changes



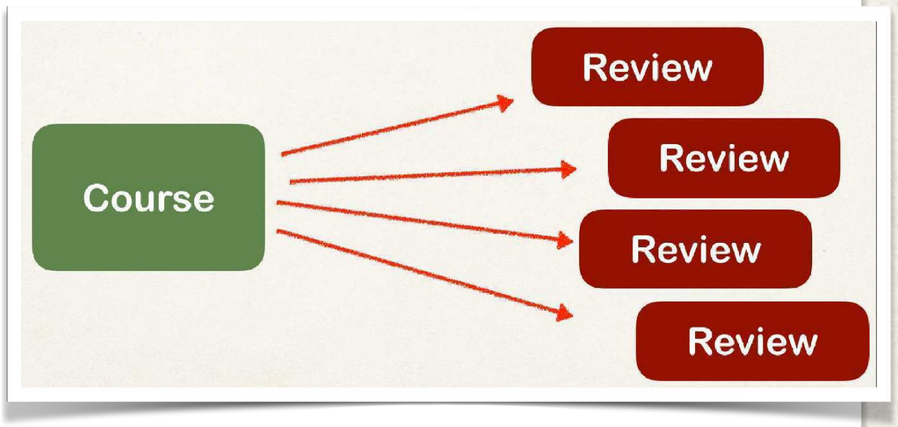
Step 2: Create Review class

```
@Entity
@Table(name="review")
public class Review {

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

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


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



Step 3: Update Course - reference reviews

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


    private List<Review> reviews;

    // getter / setters
    ...

}
```



Add @OneToMany annotation

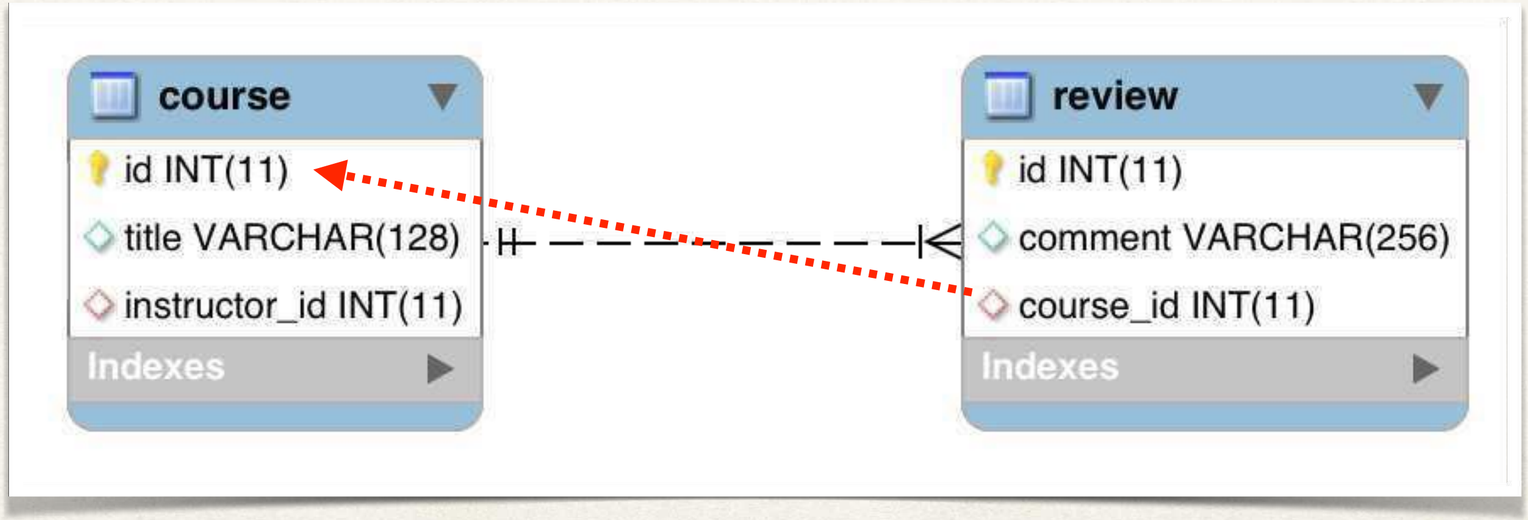
```
@Entity
@Table(name="course")
public class Course {
    ...

    @OneToMany
    @JoinColumn(name="course_id")
    private List<Review> reviews;

    // getter / setters
    ...

}
```

Refers to “course_id” column in “review” table

More: @JoinColumn

- In this scenario, **@JoinColumn** tells Hibernate
  - Look at the **course_id** column in the **review** table
  - Use this information to help find associated reviews for a course

```
public class Course {
    ...

    @OneToMany
    @JoinColumn(name="course_id")
    private List<Review> reviews;
```



Add support for Cascading

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

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="course_id")
    private List<Review> reviews;

    ...

}
```

Cascade all operations including deletes!

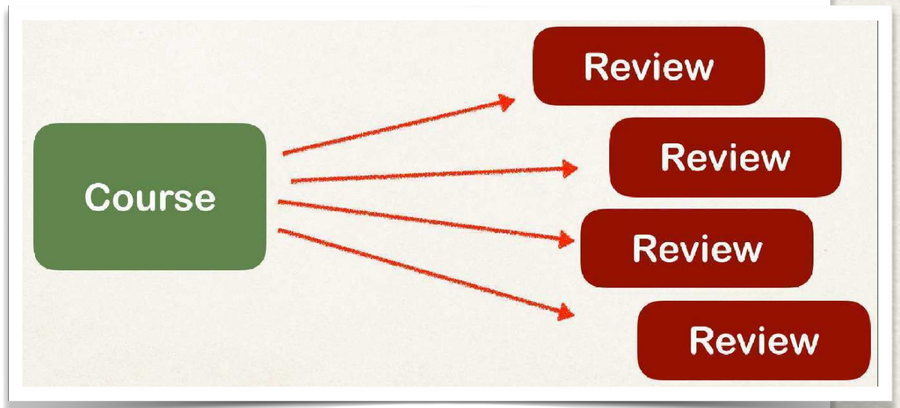
Add support for Lazy loading

Lazy load the reviews

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

    @OneToMany(fetch=FetchType.LAZY, cascade=CascadeType.ALL)
    @JoinColumn(name="course_id")
    private List<Review> reviews;

    ...

}
```

Add convenience method for adding review

```
@Entity
@Table(name="course")
public class Course {
    ...
    // add convenience methods for adding reviews

    public void add(Review tempReview) {

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

        reviews.add(tempReview);

    }
    ...
}
```

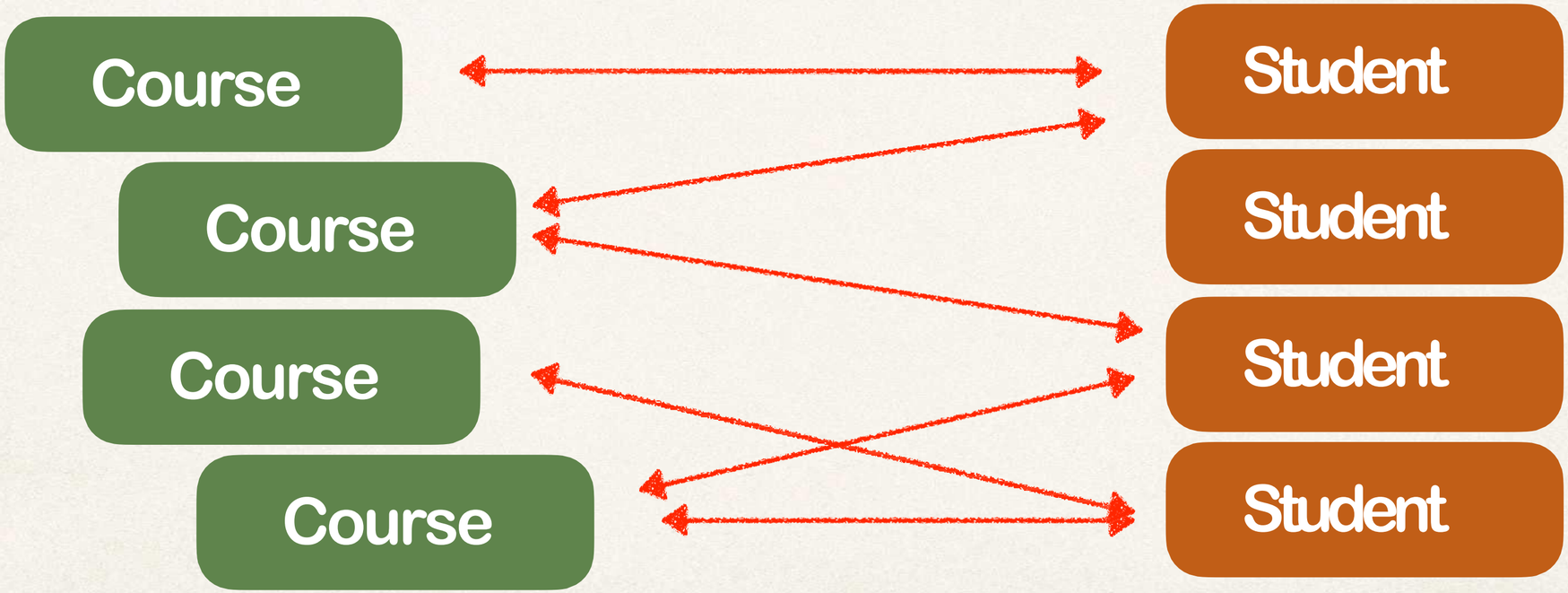


luv2code

@ManyToMany

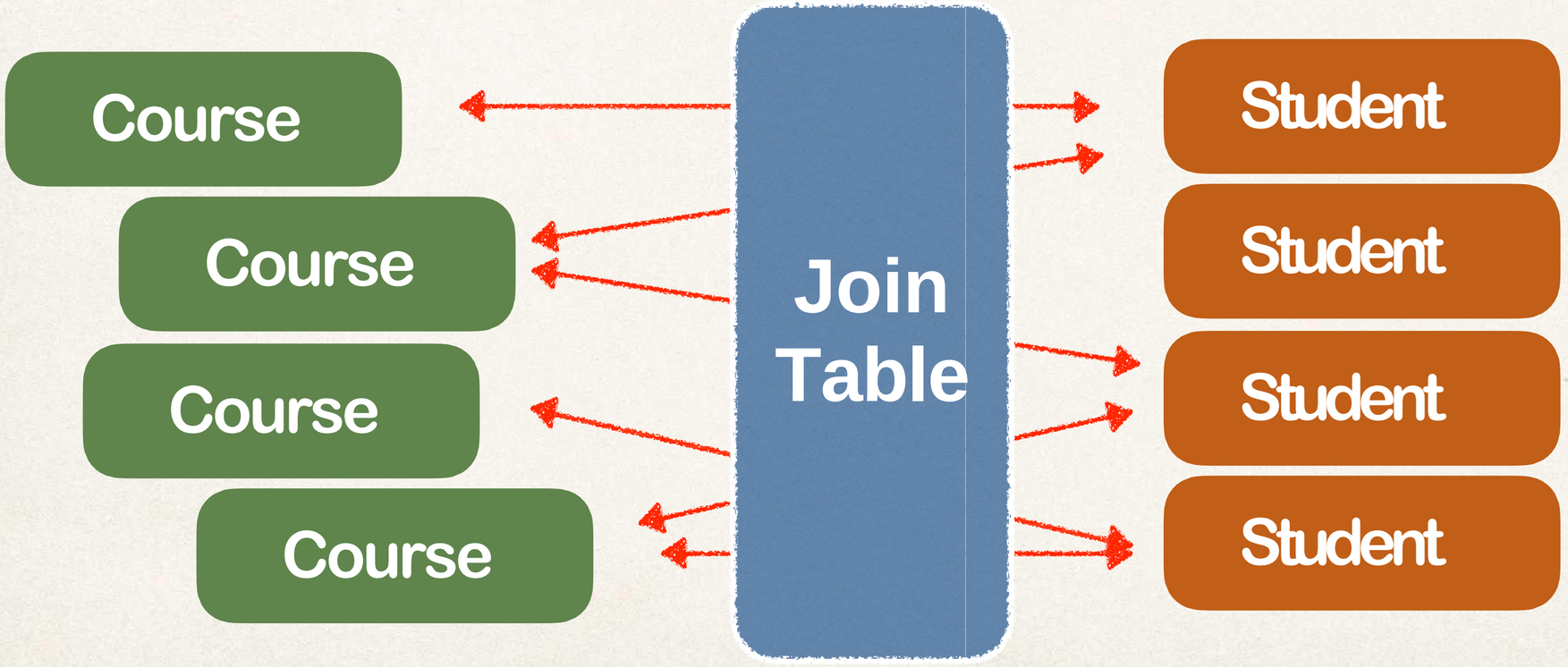
Many-to-Many Mapping

- A course can have many students
- A student can have many courses



Keep track of relationships

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

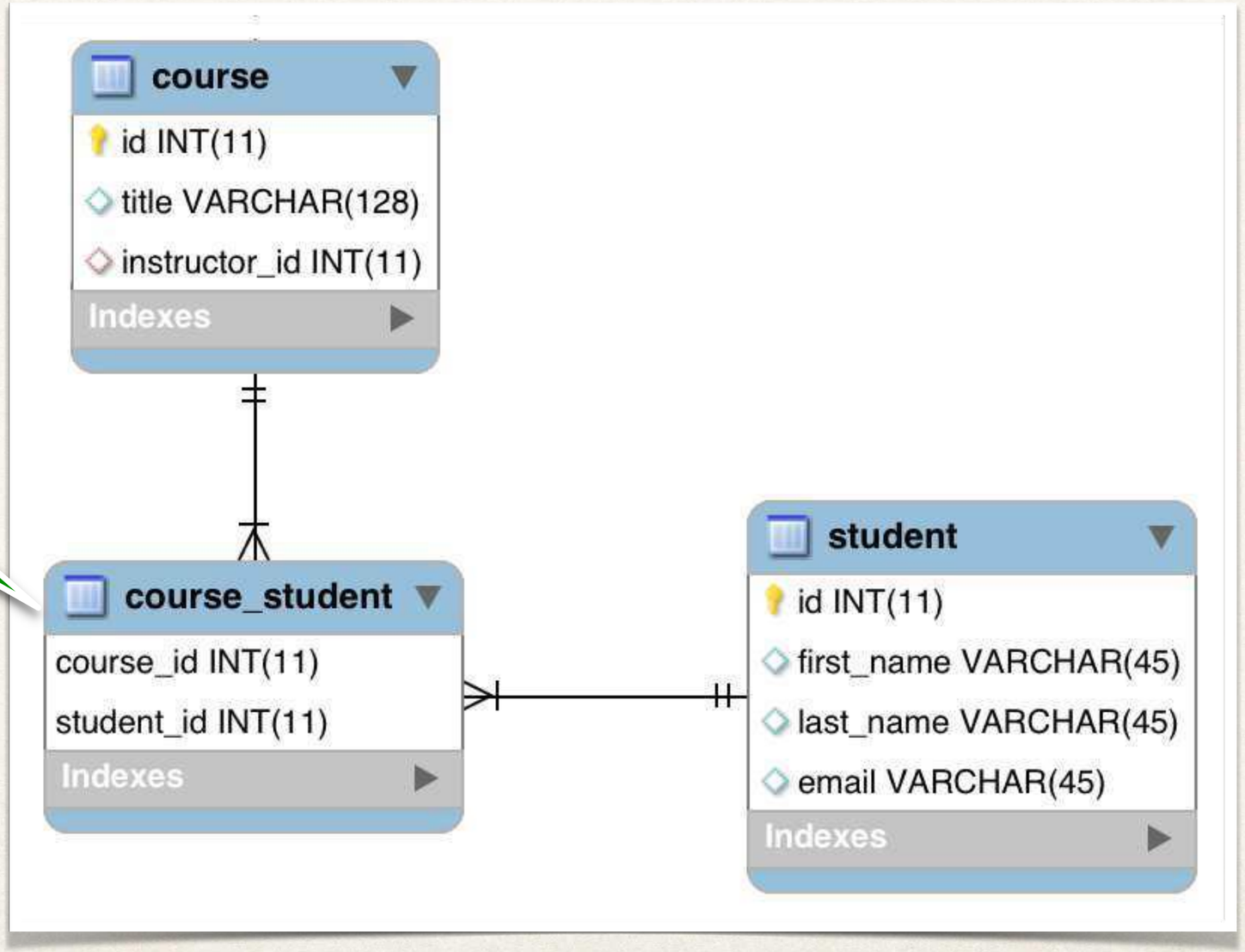


Join Table

A table that provides a mapping between two tables.

It has foreign keys for each table
to define the mapping relationship.

@ManyToMany

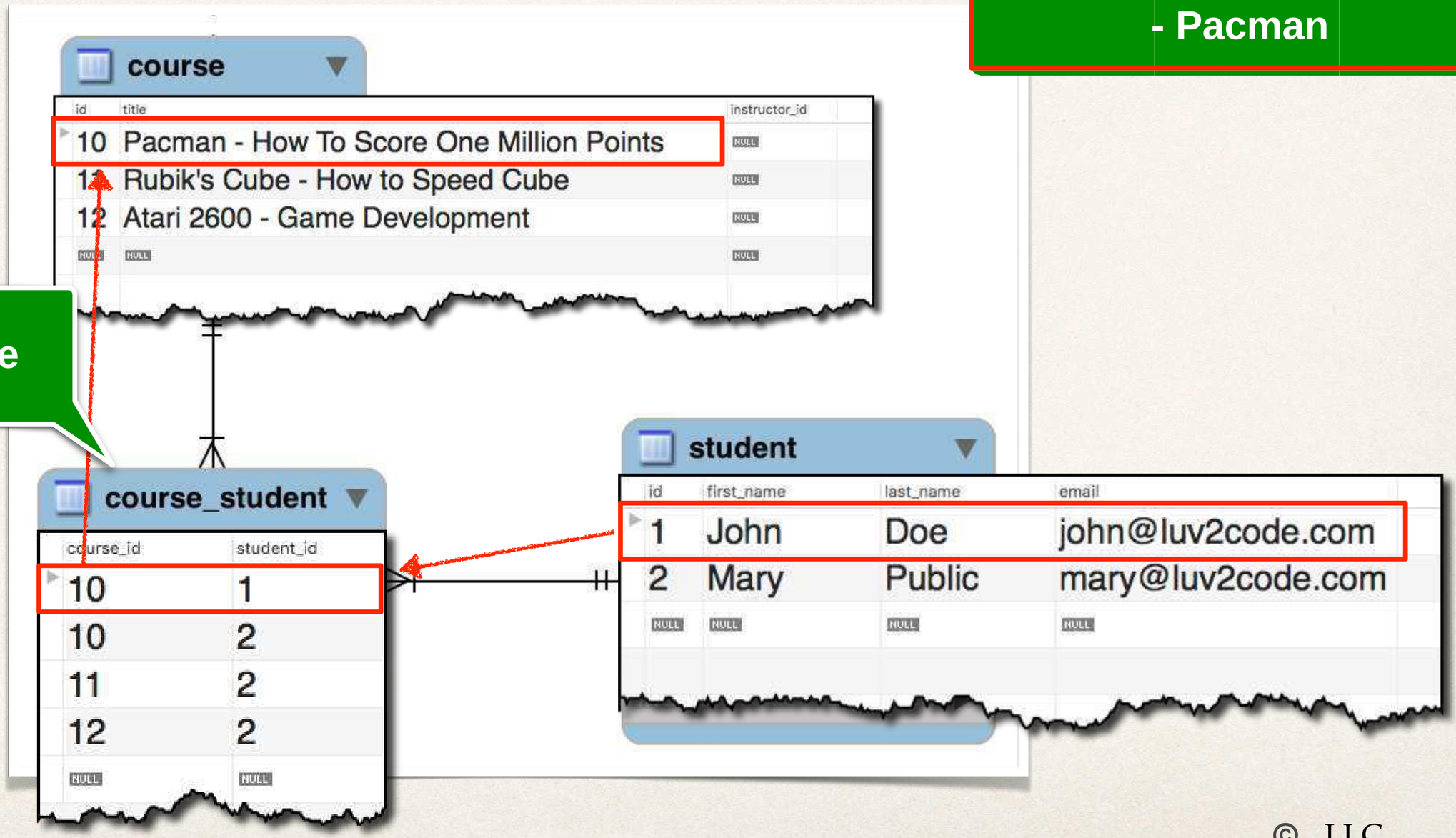


Join Table

Join Table Example

Find John's Courses

- Pacman

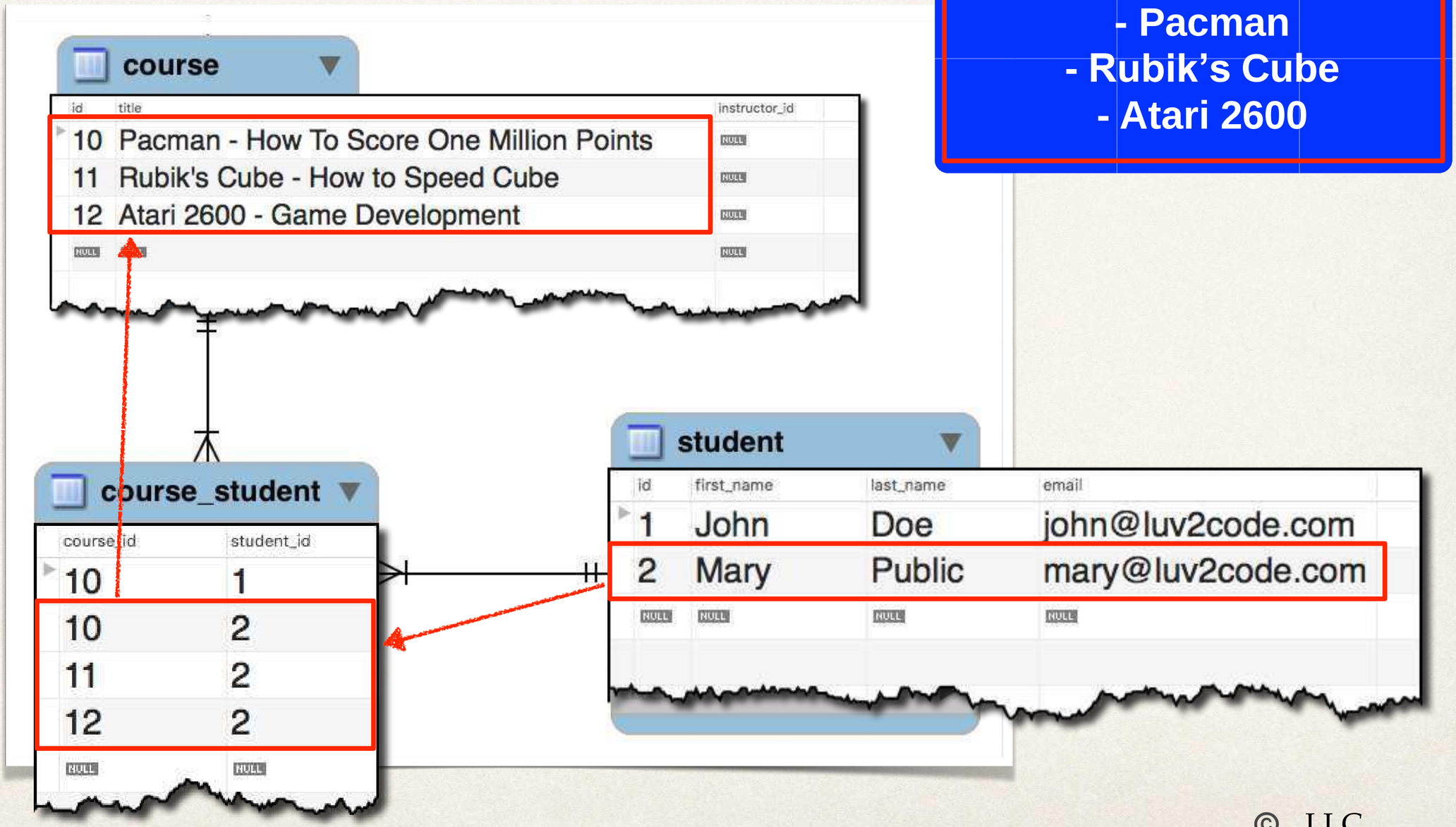


Join Table

Join Table Example

Find Mary's Courses

- Pacman
- Rubik's Cube
- Atari 2600



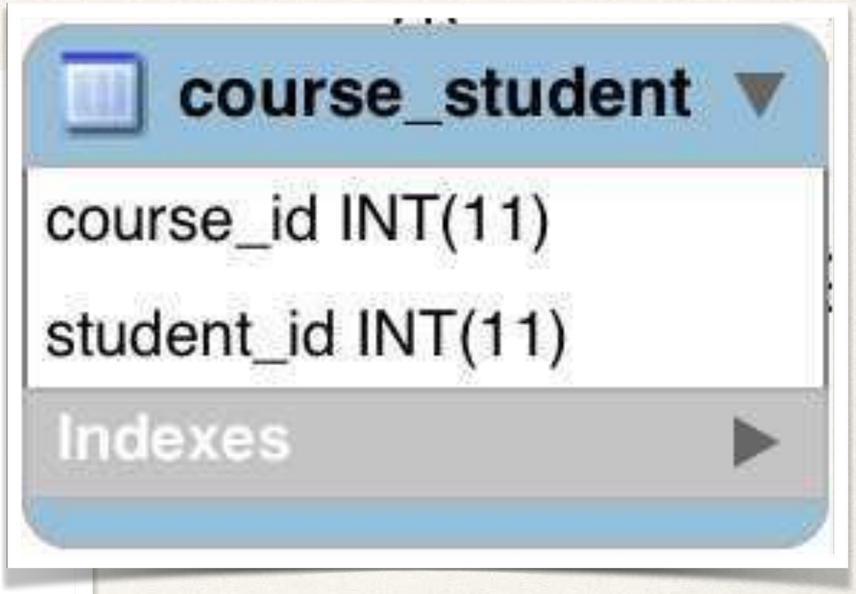
Development Process: Many-to-Many

1. Prep Work - Define database tables
2. Update **Course** class
3. Update **Student** class

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`),

    ...
);
```

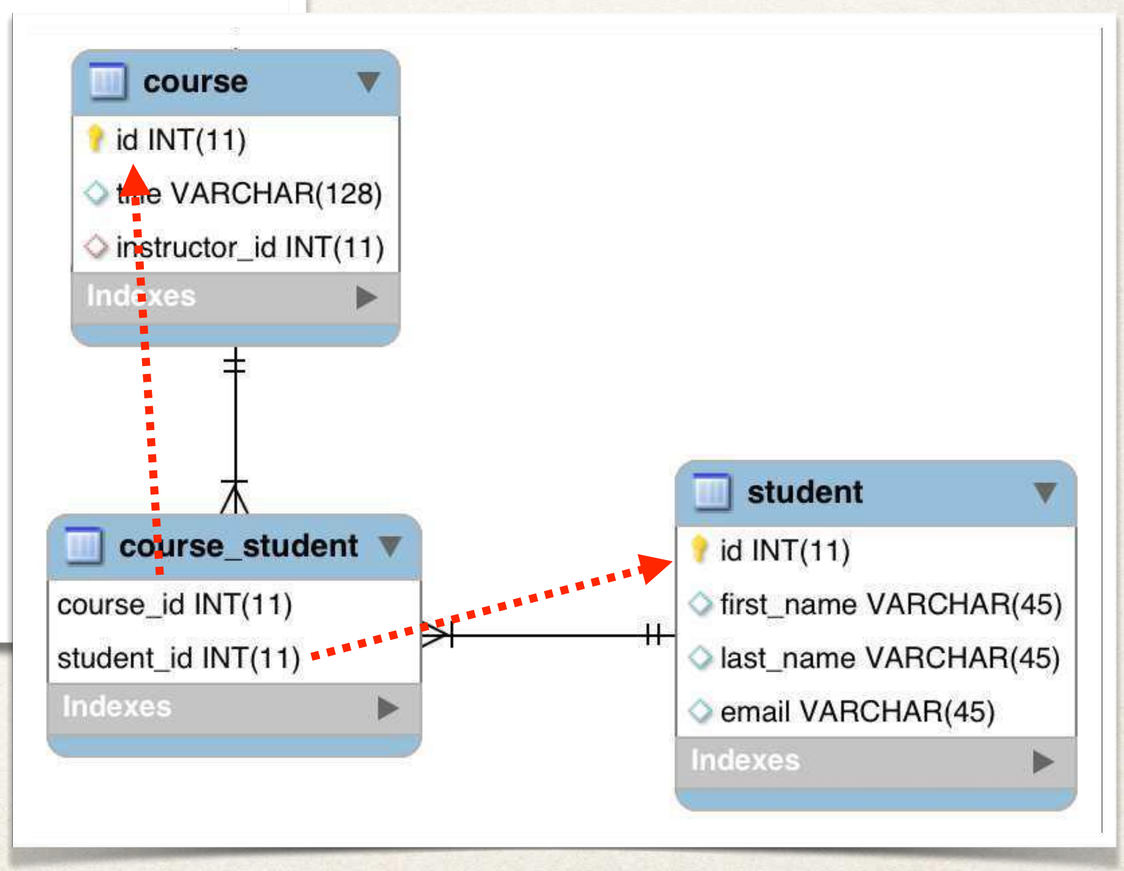


join table: course_student - foreign keys

```
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`)
  …
);
```

Table

Column

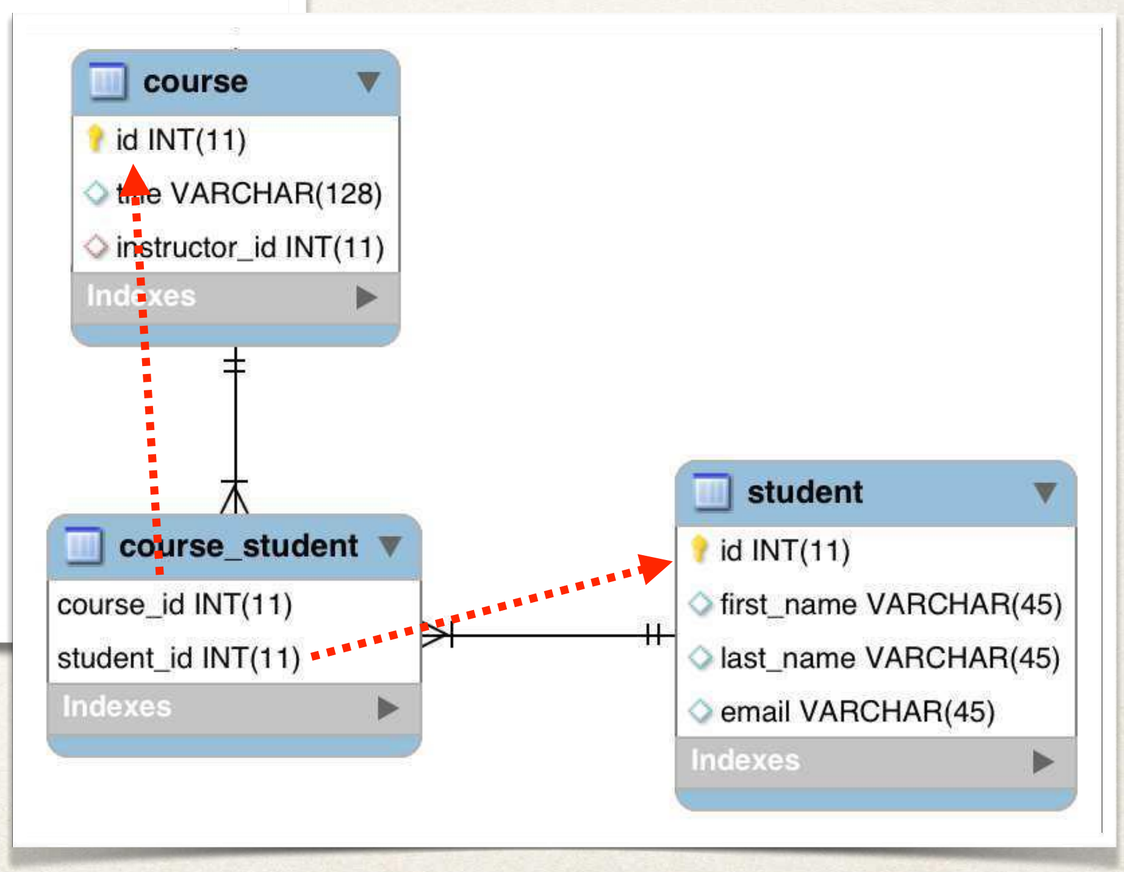


join table: course_student - foreign keys

```
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`)
  …
);
```

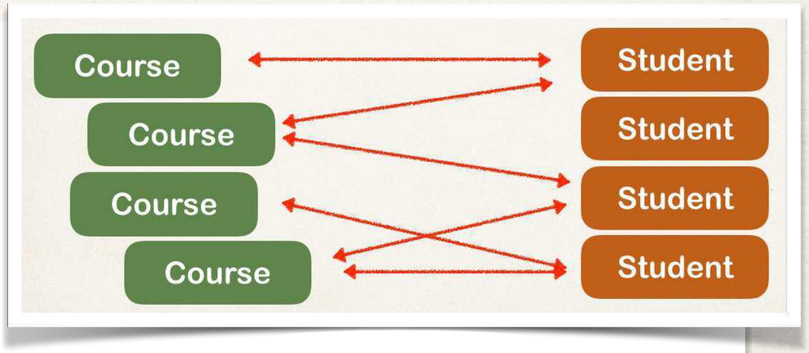
Table

Column



Step 2: Update Course - reference students

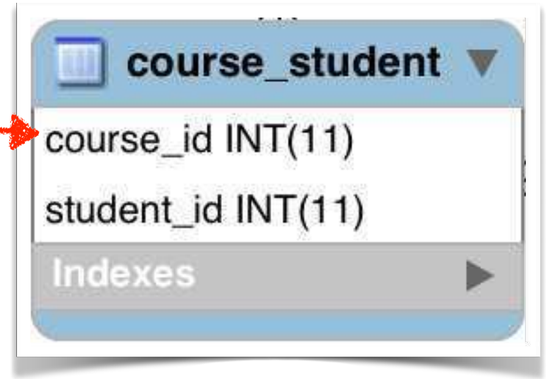
```
@Entity
@Table(name="course")
public class Course {
    ...


    private List<Student> students;

    // getter / setters
    ...

}
```



Add @ManyToMany annotation

```
@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
    ...

}
```

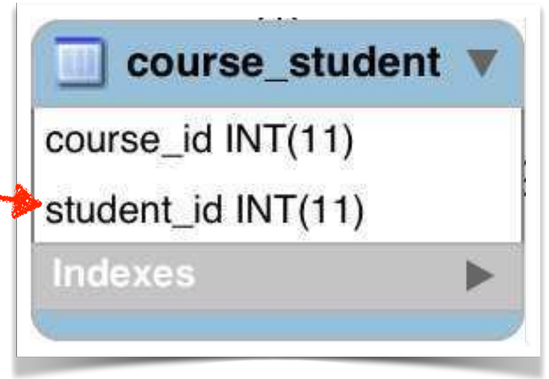
Refers to “course_id” column in “course_student” join table



Add @ManyToMany annotation

```
@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
    …

}
```

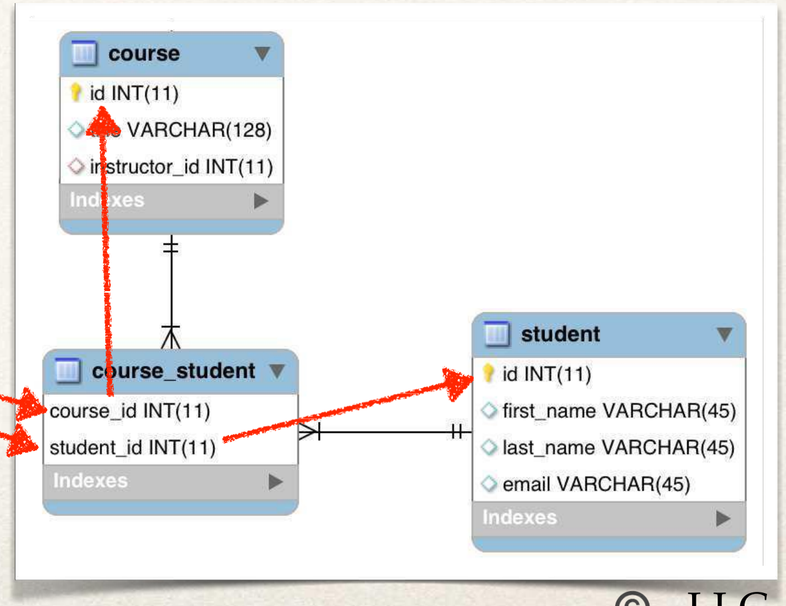
Refers to “student_id” column in “course_student” join table



More: @JoinTable

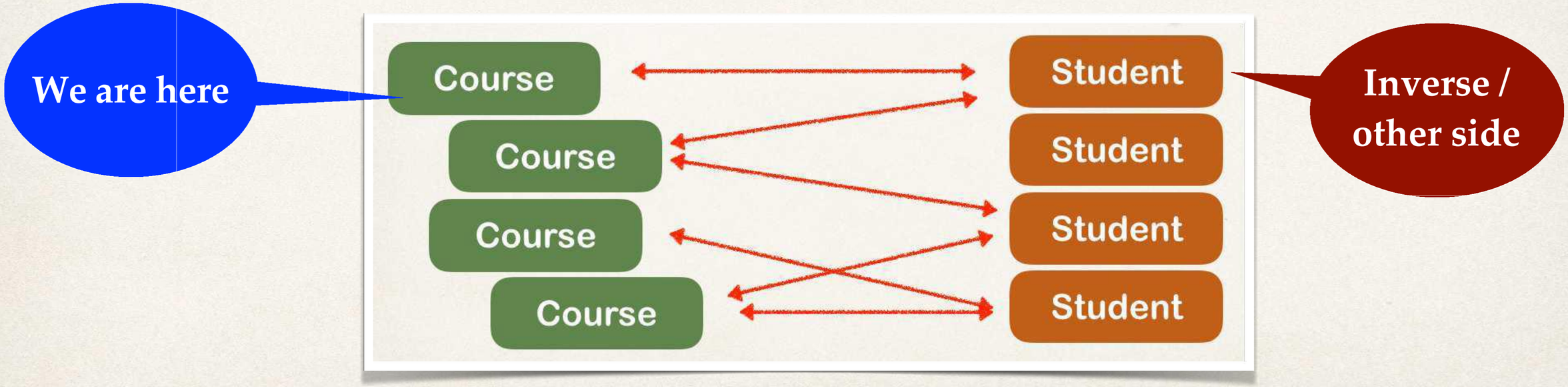
- **@JoinTable** tells Hibernate
  - Look at the **course_id** column in the **course_student** table
  - For other side (inverse), look at the **student_id** column in the **course_student** table
  - Use this information to find relationship between **course** and **students**

```
public class Course {

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

}
```



More on “inverse”

- In this context, we are defining the relationship in the **Course** class
- The **Student** class is on the “other side” … so it is considered the “inverse”
- “Inverse” refers to the “other side” of the relationship

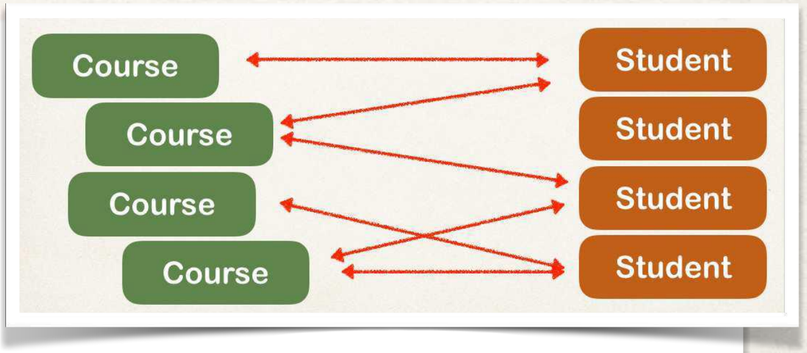


Now, let’s do a similar thing for Student

just going the other way …

Step 3: Update Student - reference courses

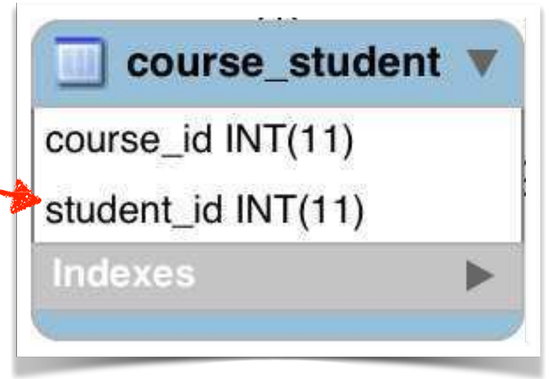
```
@Entity
@Table(name="student")
public class Student {
    ...


    private List<Course> courses;

    // getter / setters
    ...

}
```



Add @ManyToMany annotation

```
@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
    ...

}
```

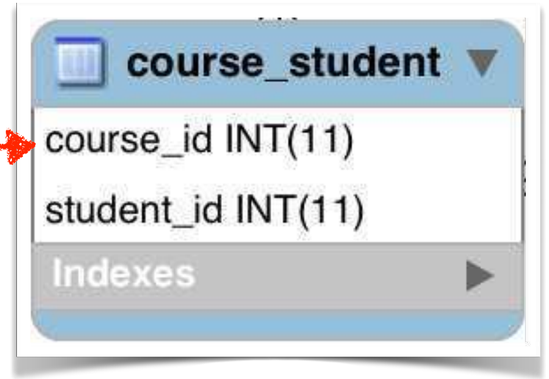
Refers to “student_id” column in “course_student” join table



Add @ManyToMany annotation

```
@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
    ...

}
```

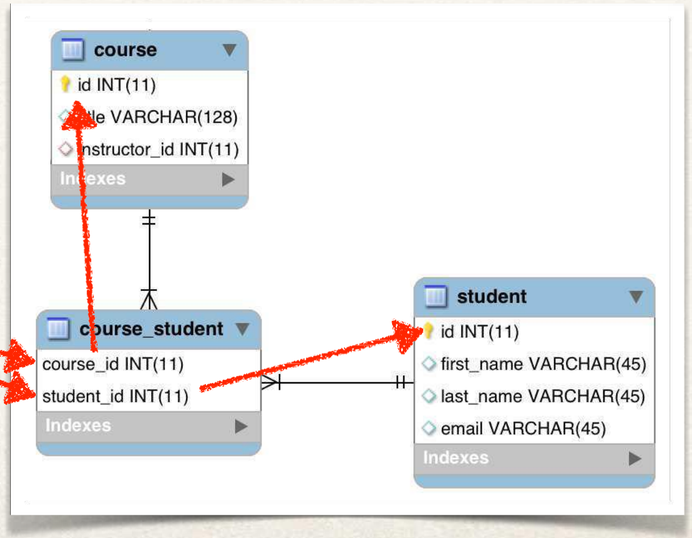
Refers to “course_id” column in “course_student” join table



More: @JoinTable

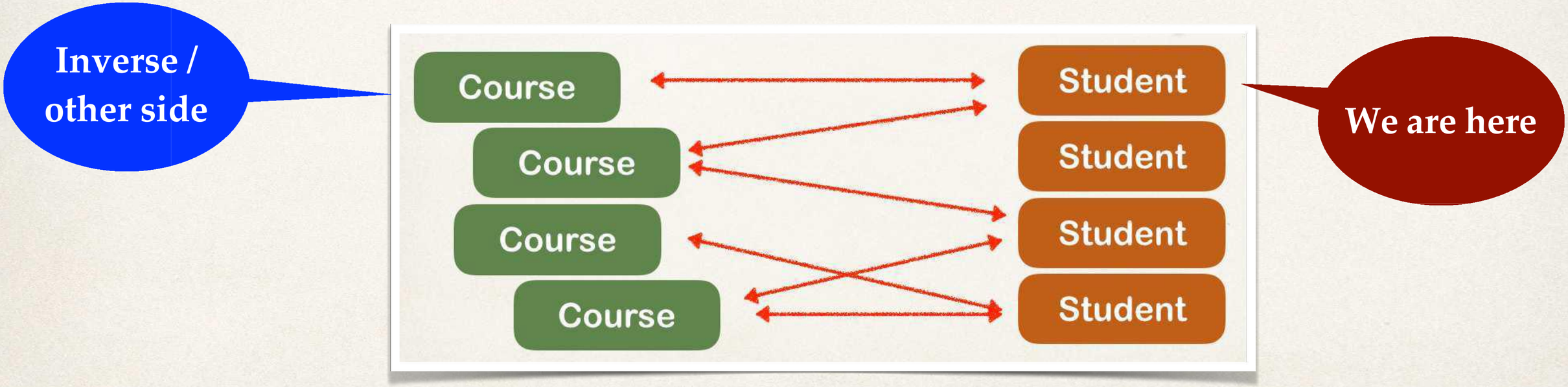
- **@JoinTable** tells Hibernate
  - Look at the **student_id** column in the **course_student** table
  - For other side (inverse), look at the **course_id** column in the **course_student** table
  - Use this information to find relationship between **student** and **courses**

```
public class Student {

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

}
```



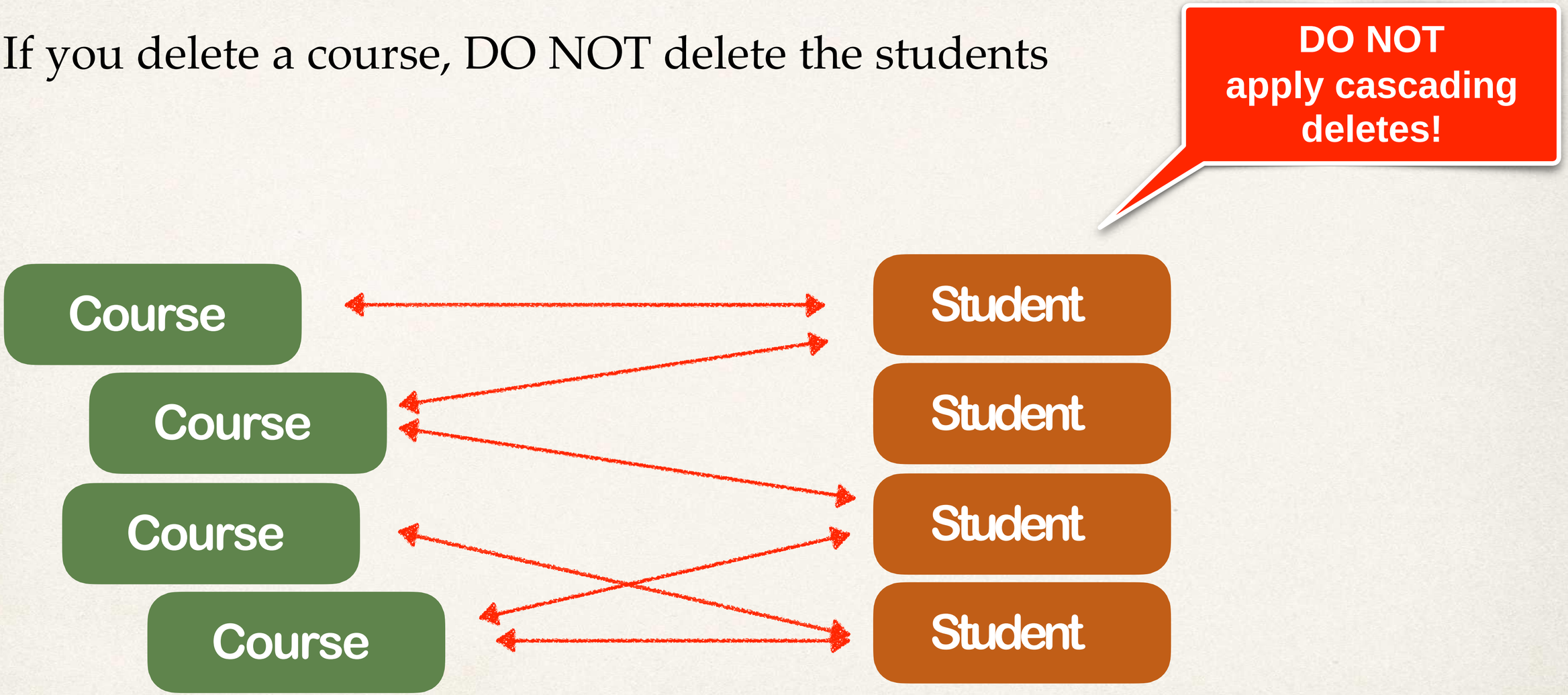
More on “inverse”

- In this context, we are defining the relationship in the **Student** class
- The **Course** class is on the “other side” … so it is considered the “inverse”
- “Inverse” refers to the “other side” of the relationship



Real-World Project Requirement

- If you delete a course, DO NOT delete the students



Other features

- In the next set of videos, we'll add support for other features
- **Lazy Loading** of students and courses
- **Cascading** to handle cascading saves … but NOT deletes
  - If we delete a course, DO NOT delete students
  - If we delete a student, DO NOT delete courses