

# Softwareentwicklung (SW)

## Relations und Mappings mit Spring Data

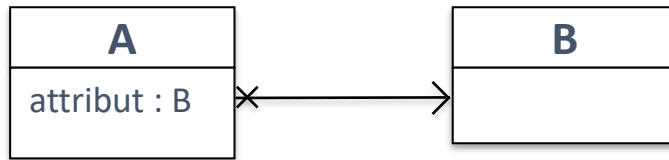
Prof. Dr. Alixandre Santana  
alixandre.santana@oth-regensburg.de

Wintersemester  
2023/2024

- Das Konzept der Direktionalität zwischen zwei Klassen zu erklären
- Die Beziehungen 1:1, N:1 und N:M und Auswirkungen zu beschreiben
- N:1-Beziehungen zu **implementieren**
- Das Konzept der Kaskadierung zu verstehen
- Die Strategien für Entity-Ableitungen zu verstehen

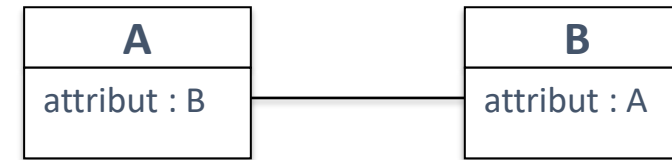
# „Direktionalität“

## Unidirektionale Relation



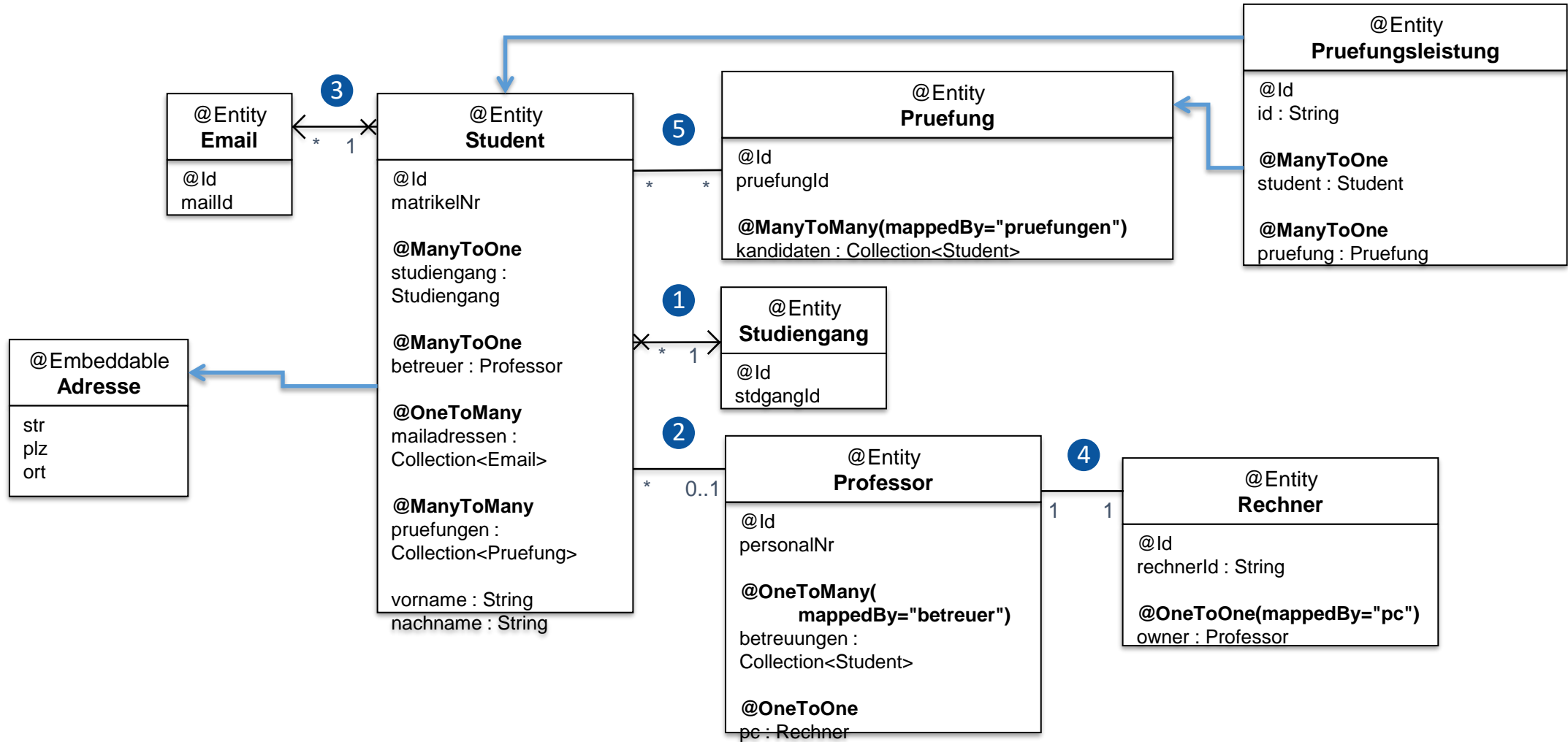
- Objekt A „kennt“ Objekt B
  - A hat ein Attribut, das auf B zeigt
- Objekt B „kennt“ Objekt A nicht
  - B hat kein Attribut, das auf A zeigt

## Bidirektionale Relation



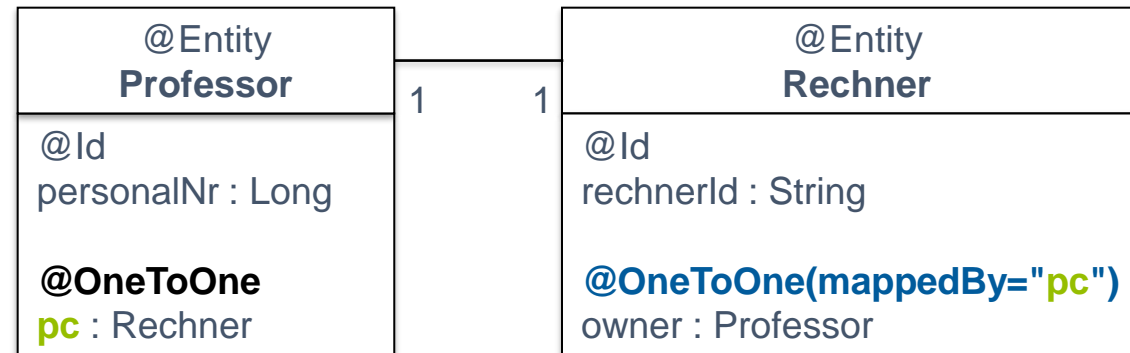
- Objekt A „kennt“ Objekt B
  - A hat ein Attribut, das auf B zeigt
- Objekt B „kennt“ auch Objekt A
  - A hat ein Attribut, das auf B zeigt
  - B hat ein Attribut, das auf A zeigt

# „Direktionalität“



# Bidirektionale 1:1-Relation

4



- Owner- und Nicht-Owner-Attribut tragen beide Annotation **@OneToOne**
- Bidirektionale Relation wird durch **mappedBy** bei Nicht-Owner erreicht

# Bidirektionale 1:1-Relation (Fremdschlüssel)

4

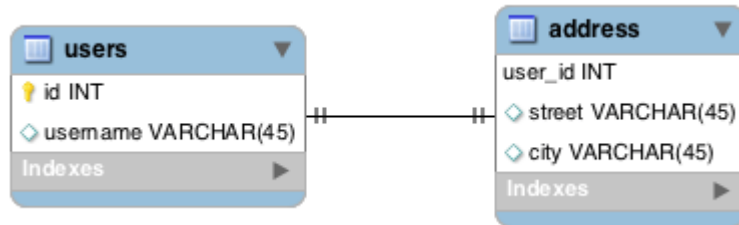
```
16 @Table(name="student")
17 public class Student implements Serializable{
18
19     /**
20      *
21      */
22     private static final long serialVersionUID = 1L;
23
24
25     @Id
26     @GeneratedValue (strategy = GenerationType.IDENTITY)
27
28     Long id;
29     @NotBlank(message = "Name is mandatory")
30     private String name;
31
32     @NotBlank(message = "Email is mandatory")
33     private String email;
34
35     @OneToOne(cascade = CascadeType.ALL)
36     @JoinColumn(name = "address_id", referencedColumnName = "id")
37     private Address address;
38
39
```

```
11 public class Address {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.IDENTITY)
15     @Column(name = "id")
16     private Long id;
17
18     private String street;
19
20     private String houseNumber;
21     private String ZPL;
22
23     @OneToOne(mappedBy = "address")
24     private Student student;
25
```

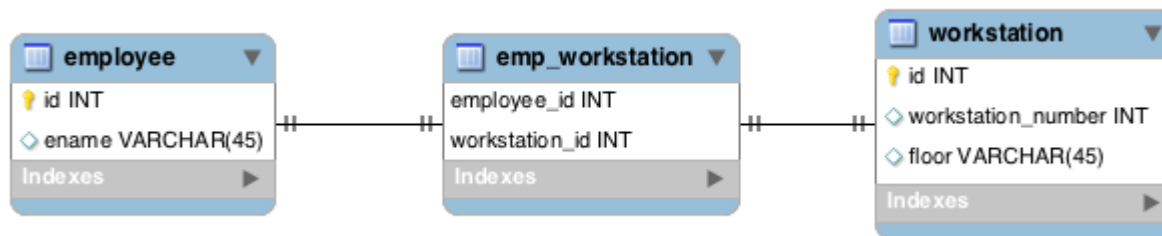
- Owner- und Nicht-Owner-Attribut tragen beide Annotation **@OneToOne**
- Bidirektionale Relation wird durch **mappedBy** bei Nicht-Owner erreicht

# Bidirektionale 1:1-Relation – Andere Möglichkeiten

- **Mit Shared Primary Key**

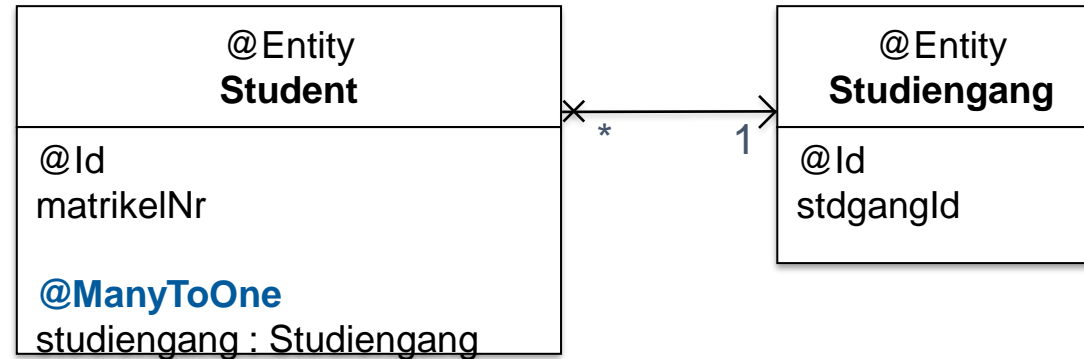


- **Mit Join Table**



# Unidirektionale n:1-Relation

1



- Owner-Attribut trägt Annotation **@ManyToOne**
- Kein Attribut beim Nicht-Owner
- Student „kennt“ seinen Studiengang (es gibt Attribut bzw. Getter)
- Studiengang kennt seine Studenten **nicht** (es gibt kein Attribut bzw. Getter)



# Unidirektionale n:1-Relation

```
18
19 @Entity
20 @Table(name="student")
21 public class Student implements Serializable{
22
23     private static final long serialVersionUID = 1L;
24
25     @Id
26     @GeneratedValue (strategy = GenerationType.IDENTITY)
27
28     Long id;
29     @NotBlank(message = "Name is mandatory")
30     private String name;
31
32     @Enumerated(EnumType.STRING)
33     private GenderEnum gender;
34
35     @NotBlank(message = "Email is mandatory")
36     private String email;
37
38
39     @OneToOne(cascade = CascadeType.ALL)
40     @JoinColumn(name = "address_id", referencedColumnName = "id")
41     private Address address;
42
43     @ManyToOne(cascade = CascadeType.PERSIST)
44     @JoinColumn(name = "course_id", referencedColumnName = "id")
45     private Course course;
46
```

- Owner-Attribut trägt Annotation **@ManyToOne**
- Kein Attribut beim Nicht-Owner
- Student „kennt“ seinen Studiengang (es gibt Attribut bzw. Getter)
- Studiengang kennt seine Studenten **nicht** (es gibt kein Attribut bzw. Getter)

# Unidirektionale n:1-Relation : Implementation mit Enum (1)

```

1 package de.othr.persistenceproject.utils;
2
3 public enum GenderEnum {
4
5     MALE ("M") , FEMALE("F"), OTHER ("O");
6
7     private final String displayValue;
8
9
10
11     private GenderEnum(String displayValue) {
12         this.displayValue = displayValue;
13     }
14
15     public String getDisplayValue() {
16         return displayValue;
17     }
18
19 }
20

```

```

2
3 import java.io.Serializable;
4
18
19 @Entity
20 @Table(name="student")
21 public class Student implements Serializable{
22
23     private static final long serialVersionUID = 1L;
24
25     @Id
26     @GeneratedValue (strategy = GenerationType.IDENTITY)
27
28     Long id;
29     @NotBlank(message = "Name is mandatory")
30     private String name;
31
32     @Enumerated(EnumType.STRING)
33     private GenderEnum gender;
34

```

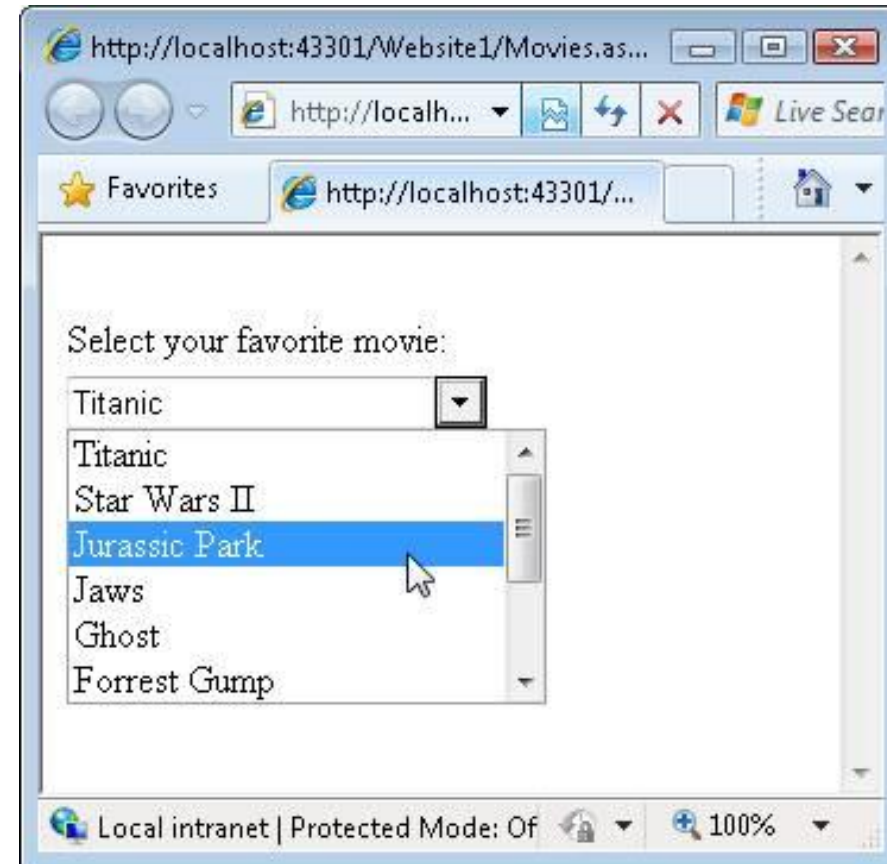
```

<select name="gender" class="form-control" id="gender" placeholder="Gender">
<option th:each="genderOpt : ${T(de.othr.persistenceproject.utils.GenderEnum).values()}"
th:field="*{gender}"
th:value="${genderOpt}"
th:text="${genderOpt.displayValue}"
th:classappend="${#fields.hasErrors('gender')} ? 'is-invalid'" />
</option>
</select>

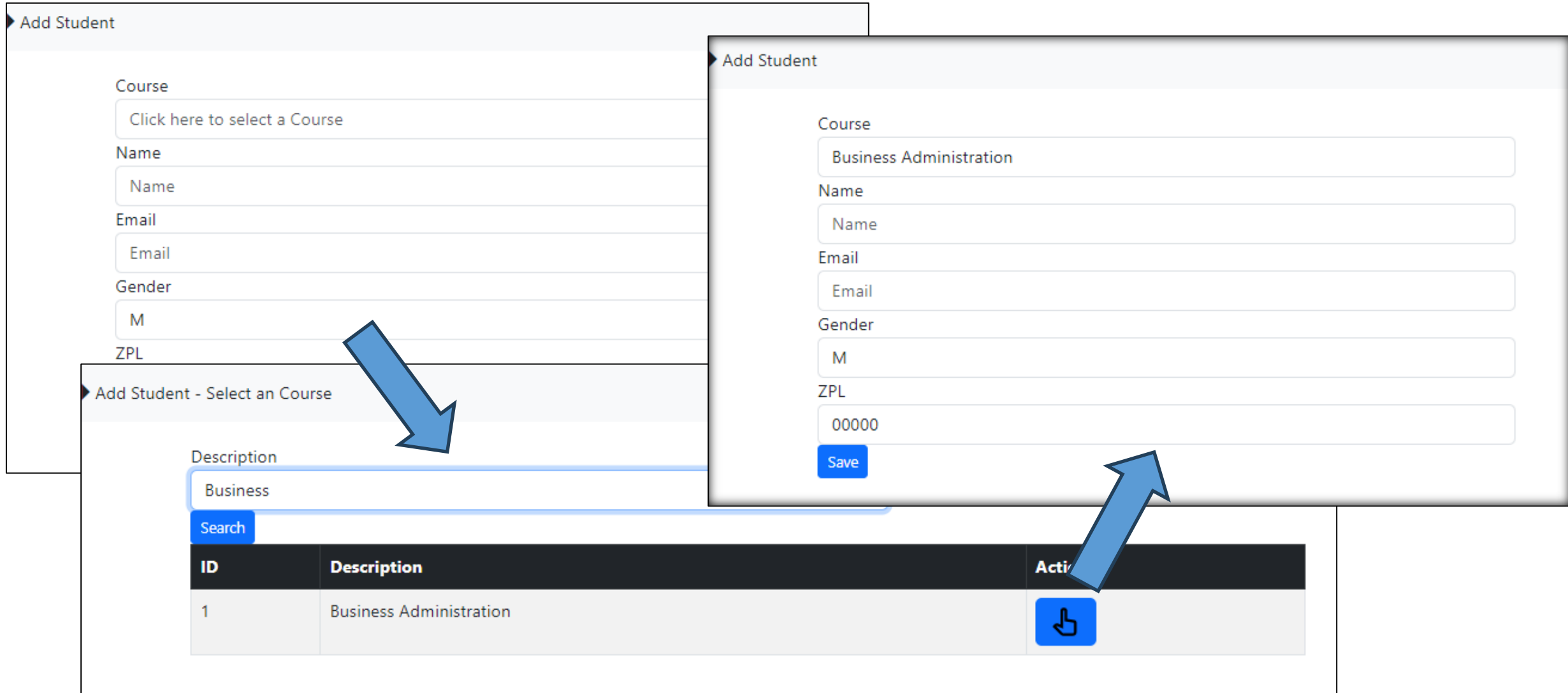
```

# Unidirektionale n:1-Relation : **Implementation mit <select> (2)**

- Select the favorite movie  
for a Student using a <select> List directly from a  
Table in the Database.



# Unidirektionale n:1-Relation : Implementation mit Search Form (3)



The diagram illustrates the implementation of a unidirectional n:1 relationship between a Student and a Course. It shows three main components:

- Add Student Form:** A form for adding a new student. It includes fields for Course, Name, Email, Gender, and ZPL. The Course field is a dropdown menu with the text "Click here to select a Course".
- Add Student - Select an Course:** A search form for selecting a course. It includes a text input field for the course description (containing "Business"), a "Search" button, and a table of results.
- Course Table:** A table with columns ID, Description, and Action. It contains one row with ID 1 and Description "Business Administration". The Action column has a button with a hand icon.

Arrows indicate the flow of data: from the "Add Student" form to the "Select an Course" form, and from the "Select an Course" form to the "Add Student" form.

**Add Student Form:**

Course: Click here to select a Course

Name: Name

Email: Email


Gender: M

ZPL: ZPL

**Add Student - Select an Course:**

Description: Business

Search

ID	Description	Action
1	Business Administration	

# Bidirektionale n:1-Relation

2



- Owner-Attribut trägt Annotation `@ManyToOne`
- Nicht-Owner-Attribut trägt Annotation `@OneToMany`
- Bei **bidirektionalen** Relationen gilt immer:
  - Annotation auf **Nicht**-Owner-Seite hat immer `mappedBy="..."`
  - `mappedBy` verweist immer auf den Attributnamen der Owner-Seite
- Student kennt seinen Betreuer (Attribut `betreuer`)
- Betreuer kennt die betreuten Studenten (`Collection<Student>`)

# Auswirkung Implementierung Nicht-Owner

- Da die Relation bidirektional ist, enthält der Nicht-Owner eine Collection seiner Owner (Klasse Student ist Owner)
  - Beispiel: der Professor (Nicht-Owner) hat eine Liste betreuter Studenten

```
private List<Student> betreungen;  
  
public List<Student> getBetreuungen(){  
    return this.betreuungen;  
}
```

- Wird die Owner-Liste durch andere Objekte über einen Getter gelesen, so könnte diese Liste an anderer Stelle verändert werden, ohne dass die Änderung persistiert wird (d. h. n würde zu n+1 oder n-1)
  - Hinzufügen/Löschen in Owner-Liste bedürfte auch Änderungen an Ownern
  - ➔ Die Liste der Owner muss vor Weitergabe „read-only“ gesetzt werden:

```
public List<Student> getBetreuungen() {  
    return Collections.unmodifiableList(this.betreuungen);  
}
```

# Auswirkung Löschen eines Nicht-Owners

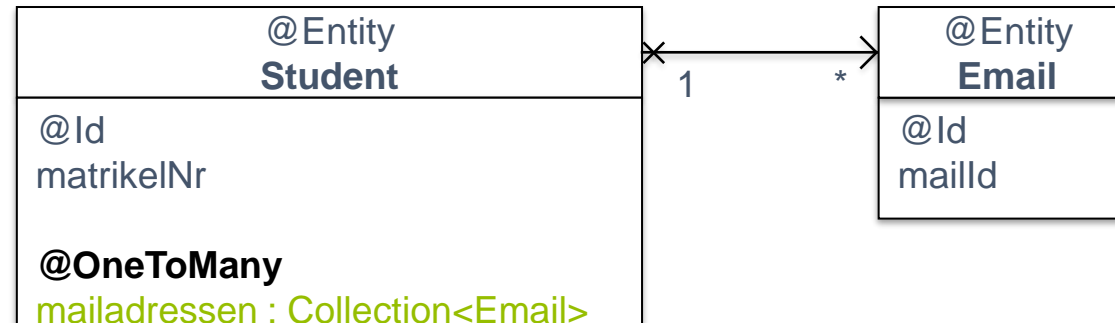
- Bei der Definition von Fremdschlüsselbeziehungen werden auch entsprechende „Constraints“<sup>1</sup> in der Datenbank angelegt
- Bei einem `remove()` auf einen Nicht-Owner, auf den in einem Owner noch referenziert wird, kommt es zu einer `IntegrityConstraintViolationException`
- **Beispiel:** Hat ein Student (Owner) eine Referenz auf einen Professor (Nicht-Owner) als Betreuer, kann der Professor erst dann gelöscht werden (`remove`), wenn die Referenz auf ihn im Studenten-Objekt gelöscht wurde (`null` gesetzt):

```
@Autowired private ProfessorRepository repo; @Autowired private StudentRepository studentRepo;
public void professorLoeschen(Long personalNr) {
    Option<Professor> professor = repo.find(personalNr);
    professor.ifPresent( prof -> {
        if(prof.getBetreuungen()!=null){
            if(prof.getBetreuungen().size()>=1){ // d.h. der Prof. wird bei mind. 1 Studenten referenziert
                List<Student> betreuungen = prof.getBetreuungen();
                for(Student betreuung : betreuungen){
                    betreuung.setBetreuer(null); // Betreuung beim Student muss zuerst gelöscht werden...
                    studentRepo.save(betreuung); // ... und diese Änderung am Student persistiert werden
                }
            }
        }
    })
    repo.delete(prof); // erst dann kann der Professor durch den Entity-Manager entfernt werden
}
```

<sup>1</sup>Ausnahme z. B. MySQL Engine „MyISAM“ erlaubt keine „Foreign Key Constraints“

# Unidirektionale 1:n-Relation

3

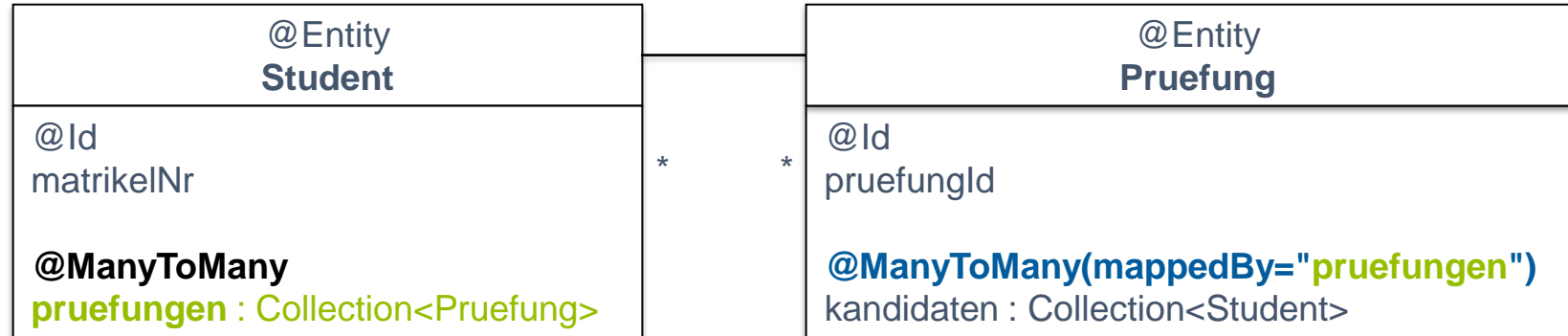


- Owner-Attribut trägt Annotation **@OneToMany**
- Kein Attribut beim Nicht-Owner
- Student kennt seine Email-Adresse
- Über Email kann nicht auf den Owner-Student zugegriffen werden



# Bidirektionale n:m-Relation

5



- Owner- und Nicht-Owner-Attribut tragen beide Annotation **@ManyToMany**
- Bidirektionale Relation wird durch **mappedBy** bei Nicht-Owner erreicht
- Eine n:m-Relation bedarf einer eigenen Join-Tabelle, diese wird automatisch vom Persistence-Manager erzeugt und verwaltet
  - Änderungen an der Collection von Nicht-Owner im Owner-Objekt (hier: **pruefungen**) führen bei **persist()** auf den Owner automatisch zum Update auf die Join-Tabelle
  - Die Objekte in den Collections im Speicher (von Owner und Nicht-Owner, hier: **pruefungen** und **kandidaten**) werden durch **persist()** **nicht** geändert
  - **Problem:** Die Klasse **Pruefung** gibt eine „read-only“-Liste von Studenten zurück...

# Auswirkungen auf Nicht-Owner-Code

...deshalb braucht der Nicht-Owner eigene Methoden, um einzelne Owner-Objekte hinzuzufügen oder zu entfernen

- **Beispiel:** Die Klasse Professor bekommt zusätzliche Methoden, z. B.:  
addKandidat(Student student) und removeKandidat(Student student)

```
// Der Nicht-Owner
@Entity
public class Pruefung implements Serializable
// ...
    public void addKandidat(Student student){
        if(!kandidaten.contains(student))
            kandidaten.add(student);
    }

    public void removeKandidat(Student student){
        if(kandidaten.contains(student))
            kandidaten.remove(student);
    }

    public List<Student> getKandidaten() {
        return Collections.unmodifiableList(kandidaten);
    }
}
```

```
public class PruefungService {
    @Autowired private StudentRepository studentRepo;
    @Autowired private PruefungRepository pruefungRepo;
    // ...
    @Transactional(TxType.REQUIRED)
    public void pruefungAnmelden(String pruefId, Long matrNr){
        Student student = studentRepo.findById(matrNr).get();
        List<Pruefung> pruefungen = student.getPruefungen();
        Pruefung p = pruefungRepo.findById(pruefungId).get();

        if(student!=null && pruefungen!=null && p!=null){
            if(!pruefungen.contains(p)){
                pruefungen.add(p);           // im Owner hinzufügen...
                p.addKandidat(student);      // ... und im Nicht-Owner
                studentRepo.save(student);   // ändert nur Join-
                                           // Tabelle
            }
        }
    }
}
```

**Weitere Umsetzungsstrategien sehen wir nächste Woche!**

# Kaskadierung

- **Grundsatz:** Persistente Entites dürfen in deren Relationsbeziehung-en nur mit persistenten Entity-Objekten gespeichert werden
- Zusammengehörige (referenzierte) Entities müssen einzeln persistiert werden...
  - ... „von innen nach außen“, d. h. Nicht-Owner zuerst, dann der Owner



@Entity **Rechner**

```
rechnerRepo.save(rechner);
prof.setRechner(rechner);
profRepo.save(prof);    // Reihenfolge!!
```

- Der Parameter @...To... (**cascade={CascadeType...}**) in Relationsannotationen erlaubt eine automatische Anwendung von Methoden des Entity-Managers wie `persist()`, `merge()` u. a. auf alle referenzierten Entity-Objekte (d. h. auf Nicht-Owner)
- Welche Methoden auf referenzierte Objekte angewandt werden kann durch die Enumeration `CascadeType`<sup>1</sup> festgelegt werden

```
@Entity
public class Professor {
    // ...
    @OneToOne(cascade = {CascadeType.PERSIST})
    private Rechner pc;
}
```

<sup>1</sup>Möglich sind folgende Attribute: ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH

# Kaskadierung

- **Grundsatz:** Persistente Entites dürfen in deren Relationsbeziehung-en nur mit persistenten Entity-Objekten gespeichert werden

All JPA-specific cascade operations are represented by the `javax.persistence.CascadeType` enum containing entries:

- ALL
- PERSIST
- MERGE
- REMOVE
- ....

# Abbildung von Entity-Ableitungen

Eine Entity kann von einer anderen abgeleitet werden. Abgeleitete Entities tragen die Annotation `@Inheritance(strategy=...)`, die drei Strategien zur Abbildung in der DB haben kann:

- `InheritanceType.SINGLE_TABLE`

DTYPE	matrikelNr	name	fach
Student	123456	Max Muster	<NULL>
Tutor	273645	Susanne Schnell	Mathe

```
@Entity
@Inheritance(strategy=InheritanceType....)
public class Student{
    @Id protected Long matrikelNr;
    protected String name;
}
```

```
@Entity
public class Tutor extends Student{
    private String fach;
}
```

- `InheritanceType.TABLE_PER_CLASS`

STUDENT	
matrikelNr	name
123456	Max Muster

TUTOR		
matrikelNr	name	fach
273645	Susanne Schnell	Mathe

- `InheritanceType.JOINED`

STUDENT		
DTYPE	matrikelNr	name
Student	123456	Max Muster
Tutor	273645	Susanne Schnell

TUTOR	
matrikelNr	fach
273645	Mathe

Sehen Sie: <https://stackabuse.com/guide-to-jpa-with-hibernate-inheritance-mapping/>

# Tabellenbeziehungen

## Einfache Tabelle / Relation:

*Tabelle / Relation*

STUDENT		
matrikelNr	name	vorname
123456	Muster	Max
273645	Schnell	Susanne
...	...	...

*Attribut / Spalte*

*Schema*

*einzelne Tupel / Zeilen*

## Verbindung über Fremdschlüsselbeziehung (1:n oder n:1):

STUDENT		
matrikelNr	name	vorname
123456	Muster	Max
273645	Schnell	Susanne

ADRESSE		
zu_matrikelNr	strasse	ort
123456	Unistraße 10	Regensburg
273645	Neupfarrplatz 21	Regensburg

## Verbindung über Join-Tabelle (n:m):

STUDENT		
matrikelNr	name	vorname
123456	Muster	Max
273645	Schnell	Susanne

JOIN_ST_PR	
matrikelNr	pruefungID
123456	P1
123456	P2
273645	P2

PRUEFUNG	
pruefungID	bezeichnung
P1	Programmieren 1
P2	Software-Engineering

# Entity-Lifecycle-Callbacks

- JPA Callback-Methoden, die vor oder nach Tätigkeiten des Containers (Entity-Managers) ausgeführt werden können
- Methoden ohne Parameter und mit Rückgabotyp void (in der jeweiligen Entity-Klasse) können eine oder mehrere der folgenden Annotationen erhalten; sie werden jeweils vom Entity-Manager aufgerufen (entsprechend ihrer Bedeutung):
  - `@PrePersist` und `@PostPersist`
  - `@PostLoad`
  - `@PreUpdate` und `@PostUpdate`
  - `@PreRemove` und `@PostRemove`

```
@Entity
public class Student{
    @Id private Long matrikelNr;
    private String name;

    @PrePersist
    @PreUpdate
    private void pruefeName(){
        if(this.name.length()<2)
            throw new IllegalArgumentException("zu kurz");
    }
}
```