Java Persistance API

Genres de relations

- UML associations et agrégations
  - One-to-one
  - One-to-many
  - Many-to-one
  - Many-to-many
- UML compositions
  - One-to-one
  - One-to-many

Relation unidirectionnelle

```
@Entity
public class Order implements Serializable {
  @Id
  private int id;
  private String orderName;

  @OneToOne
  private Shipment shipment;

  ...
  public Shipment getShipment() {
    return shipment;
  }
  public void setShipment(Shipment shipment) {
    this.shipment = shipment;
  }
}
```

La table Ordre à une clé étrangère vers la table Shipment

L’attribut cascade

- L’exemple précèdent peut être récrit :

```
@OneToOne(cascade={CascadeType.PERSIST})
private Shipment shipment;
```

Le comportement *cascade* est précisé par l'attribut cascade, disponible sur les annotations:

@OneToOne,@OneToMany et @ManyToMany.

La valeur de cet attribut est une énumération de type CascadeType.

En plus desvaleurs DETACH, MERGE, PERSIST, REMOVE, REFRESH, cette énumération définit la valeur ALL, qui correspond à toutes les valeurs à la fois

Relation bidirectionnelle

```
@Entity
public class Shipment implements Serializable {
  @Id
  private int id;
  private String city;
  private String zipcode;

  @OneToOne(mappedBy="shipment")
  private Order order;

  public Order getOrder() { return order; }
  public void setOrder(Order order) { this.order = order; }
}
```

Table Order has foreign key to table Shipment because Order is **owner** of the relationship

Many-to-one unidirectionnelle

```
@Entity
public class Employee implements Serializable {
   ...
   @ManyToOne
   private Company company;

   public Company getCompany() {
     return this.company;
   }
   public void setCompany(Company company){
     this.company = company;
   }
}
```

Table Employee contains a foreign key to table Company.

Many-to-one bidirectionnlle

```
@Entity
public class Company implements Serializable {
   ...
   @OneToMany(mappedBy="company")
   private Collection<Employee> employees;

   public Collection<Employee> getEmployees() {
     return this.employees;
   }
   public void setEmployees(Collection<Employee> employees){
     this.employees = employees;
   }
}
```

Many-to-many Unidirectionnelle

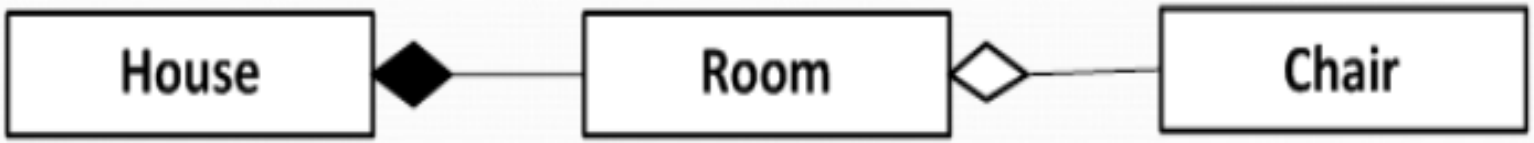
```
@Entity
public class Student implements Serializable {
  @ManyToMany
  @JoinTable(name="STUDENT_COURSE")
  private Collection<Course> courses;

  public Collection<Course> getCourses() {
    return courses;
  }
  public void setCourses(Collection<Course> courses) {
    this.courses = courses;
  }
}
```

Many-to-many bidirectionnelle

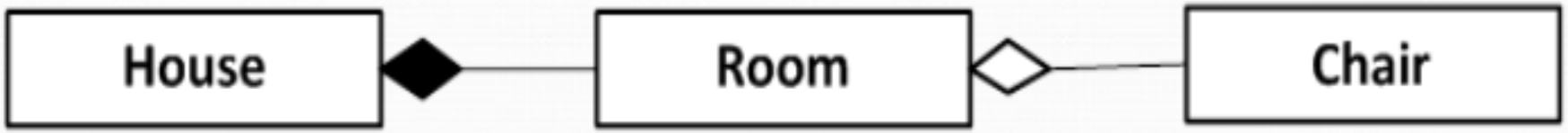
```
@Entity
public class Course implements Serializable {
  @ManyToMany(mappedBy="courses")
  private Collection<Student> students;

  public Collection<Student> getStudents() {
    return students;
  }
  public void setStudents(Collection<Students> students) {
    this.students = students;
  }
}
```

- pour les relations bidirectionnelles un-à-un, le propriétaire correspond au côté qui contient la clé étrangère correspondante
- le côté multiple de relations un à plusieurs / plusieurs à un doit être le côté propriétaire
  - par conséquent, l'élément MappedBy ne peut pas être spécifié dans l'annotation Manytoone

Agrégation



- Room – Chair
  - identity of chair *does not depend*on identity of room
  - chair is *sharable* – we can move chair from one room to another, and *chair will not change its identity*
  - chair may exist without room – it may temporarily be placed outside of the house
- All these properties characterize *aggregation* relationship – relationship with *sharable semantics*
  - Also known as *non-identifying*relationships in ER modeling



- House – Room
  - identity of room *depends*on identity of house – room is always part of some house
    - if house would change its identity (imagine reconstruction), rooms most probably would be renumbered
  - room is *non-sharable* – we cannot move room from one house to another
  - room *cannot exist* without house – if we destroy house, all its rooms are destroyed implicitly
- All these properties characterize *composition*relationship – relationship with *non-sharable semantics*
  - Also known as *identifying*relationships in ER modeling

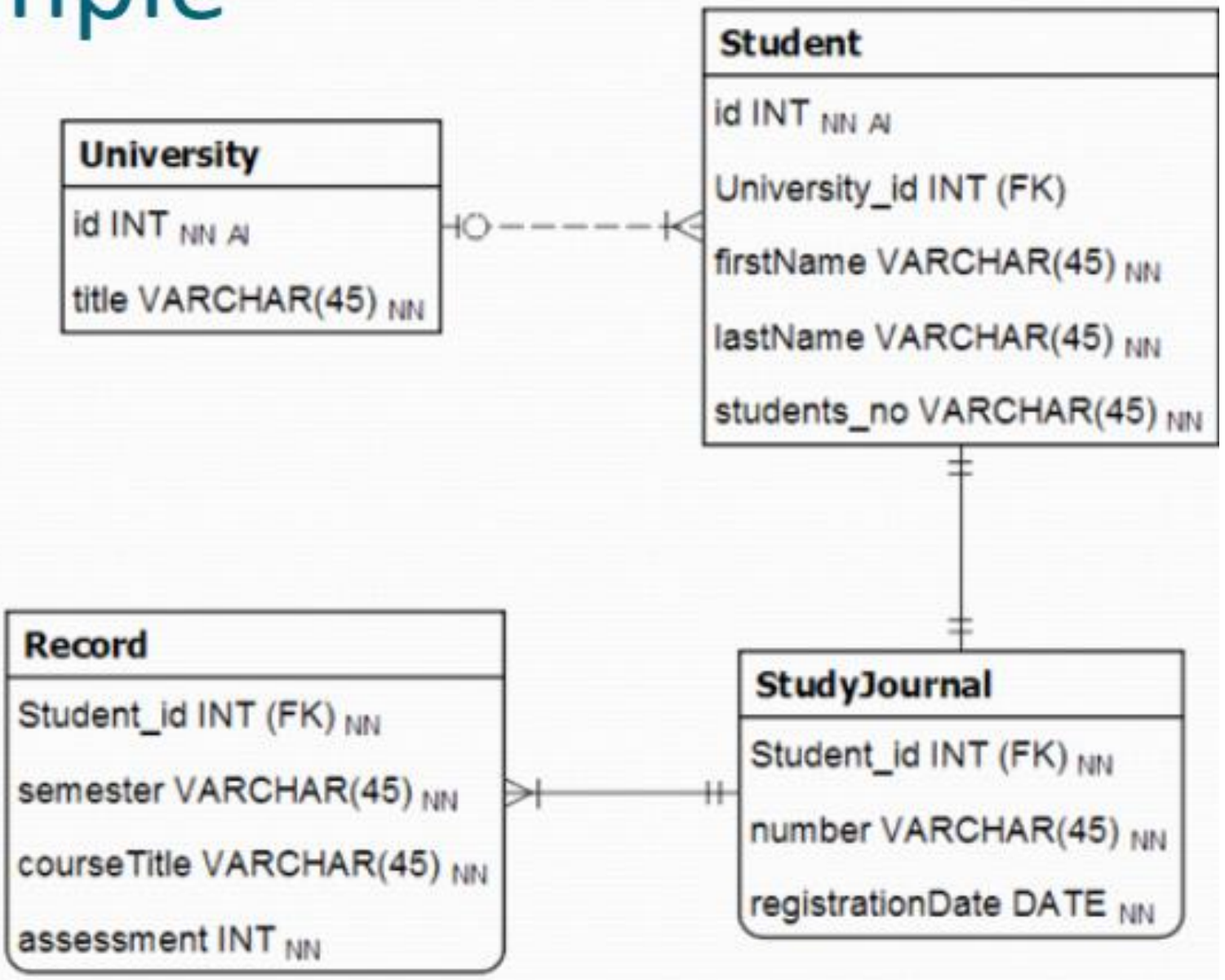
Implementing composition with JPA

- There are two ways to model composition (ER identifying) relationship:
  - Usual one-to-many relationship with additionally applied non-shared semantics (automatic orphan deletion) – attribute `orphanRemoval`
    - as noted above, programmer must take care of both ends of relationship
  - Specialized relationship – embedded element collection (`@ElementCollection`)
    - it is enough to take care of one end of relationship

one-to-many with orphan removal

- In the example above:
  - If room gets removed from house's collection (e.g.: `house.getRooms.remove(0)`), then the room will be deleted from DB automatically
  - If house is deleted from database then all its rooms are deleted from database automatically
- Note: programmer must obey the rules of non-sharable semantics
  - Removing one room from some house's collection and adding it to other house's collection would violate rules of non-sharable semantics

Example


University
id INT NN AI
title VARCHAR(45) NN
Student
id INT NN AI
University_id INT (FK)
firstName VARCHAR(45) NN
lastName VARCHAR(45) NN
students_no VARCHAR(45) NN
Record
Student_id INT (FK) NN
semester VARCHAR(45) NN
courseTitle VARCHAR(45) NN
assessment INT NN
StudyJournal
Student_id INT (FK) NN
number VARCHAR(45) NN
registrationDate DATE NN

```
public class StudyJournal implements Serializable {

  @ElementCollection
  @CollectionTable(
    name="Record",
    joinColumns=@JoinColumn(name="Student_id"))
  private List<Record> records;

  ...
}
```

```java
@Embeddable // not @Entity !
public class Record implements Serializabl
  @Column(name = "semester")
  private String semester;
  @Column(name = "courseTitle")
  private String courseTitle;
  @Column(name = "assessment")
  private Integer assessment;

  ... setters/getters ...

  public boolean equals(Object obj) {...}
  public int hashCode() {...}
}
```

- One-to-one relationship cannot be implemented as embedded element collection, we use `orphanRemoval` attribute instead:
- In entity "Student":

```
@OneToOne(mappedBy = "student", orphanRemoval=true)
private StudyJournal studyJournal;
```

- In dependent entity "StudyJournal":

```
@Id
@OneToOne
@JoinColumn(name = "Student_id",
            referencedColumnName = "id")
private Student student;
```

Héritage

- L'héritage en JPA permet de modéliser les relations entre classes dans une hiérarchie tout en définissant comment ces relations sont mappées dans une base de données relationnelle. En JPA, plusieurs stratégies permettent de représenter l'héritage, chacune ayant ses avantages et inconvénients en fonction des besoins du projet.

Types d'Héritage

- En Java, l'héritage est un mécanisme permettant à une classe (sous-classe) d'hériter des attributs et méthodes d'une autre classe (super-classe). JPA fournit des annotations pour gérer l'héritage au niveau de la base de données :
  - **Single Table (Table unique) :** Toutes les classes de la hiérarchie sont mappées dans une seule table.
  - **Table per Class (Une table par classe) :** Chaque classe possède sa propre table avec ses attributs spécifiques.
  - **Joined (Jointure) :** Les tables sont créées pour chaque classe et reliées par des jointures.

Annotations Utilisées

- @Inheritance : Spécifie la stratégie d'héritage.
- @DiscriminatorColumn : Définit une colonne pour distinguer les types dans une table unique.
- @MappedSuperclass : Utilisé pour définir une classe qui sert de super-classe mais qui ne correspond pas à une table.

Single Table

```
@Entity @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "entity_type",
 discriminatorType = DiscriminatorType.STRING)
 public class Person {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        private String name;
 }
```

Single Table

```
@Entity
@DiscriminatorValue("STUDENT")
public class Student extends Person{
        private String course;
}

@Entity
@DiscriminatorValue("TEACHER")
public class Teacher extends Person {
        private String subject;
}
```

Single Table

Exemple de contenu de table Person genere :

id	name	entity_type	course	subject
1	Alice	STUDENT	Mathématiques	NULL
2	Bob	TEACHER	NULL	Physique
3	Charlie	STUDENT	Histoire	NULL

Table per Class

Dans cette stratégie, chaque classe concrète possède sa propre table. Les tables ne contiennent que leurs propres attributs.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Person {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
}
```

```
@Entity
public class Student extends Person {
   private String course;
}
```

```
@Entity
public class Teacher extends Person {
   private String subject;
}
```

Exemple de contenu

1.Table Person (**Vide**, car seules les instances des sous-classes sont stockées dans leurs tables respectives.)

- **Table Student**

id	name	course
1	Alice	Mathématiques
2	Charlie	Histoire

- **Table Teacher**

id	name	subject
3	Bob	Physique

Jointure (Joined)

Cette stratégie crée des tables pour chaque classe et utilise des jointures pour relier les données.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Person {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
}
```

```
@Entity
public class Student extends Person {
   private String course;
}
```

```
@Entity
public class Teacher extends Person {
   private String subject;
}
```

Jointure (Joined)

Les tables des sous-classes incluent une clé étrangère référant à la clé primaire de la table de la classe de base.