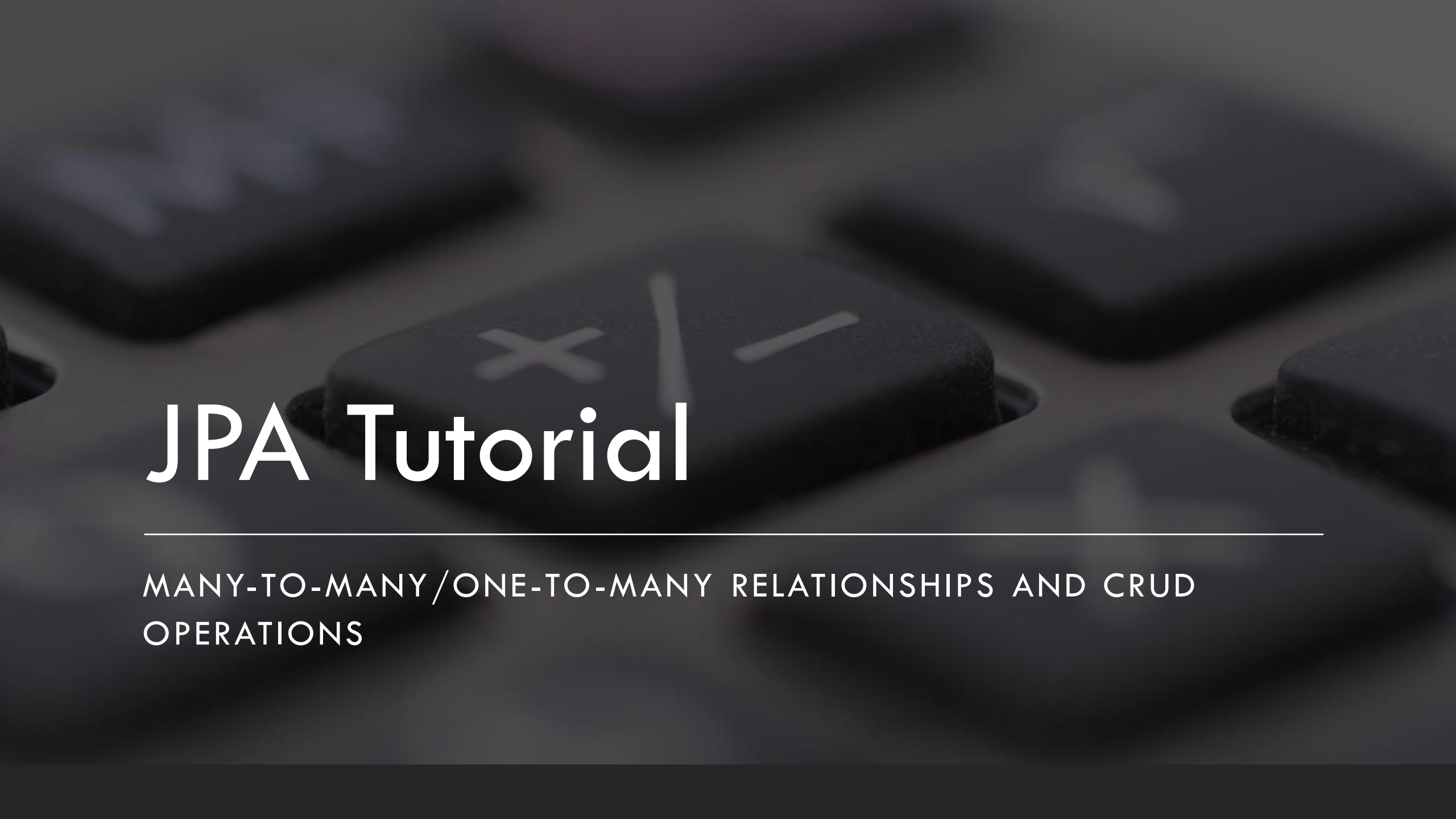
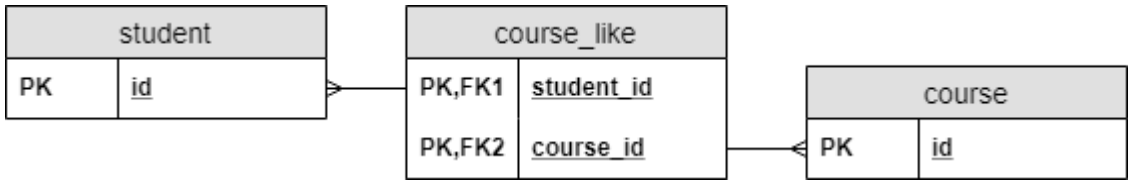


JPA Tutorial

MANY-TO-MANY/ONE-TO-MANY RELATIONSHIPS AND CRUD OPERATIONS

Many-to-many relationship

In this example, when the students mark the courses they like, we have that a student can like <u>many</u> courses, and <u>many</u> students can like the same course.



Since both sides should be able to reference the other, we need to create a separate table to hold the foreign keys. Such a table, called join table, will have as primary key a combination of the foreign keys.

Entities

```java
@Entity
   class Student {

       @Id
       Long id;

       @ManyToMany
       Set<Course> likedCourses;

       // additional properties
       // standard constructors, getters, and setters
   }
```

```java
@Entity
   class Course {

       @Id
       Long id;

       @ManyToMany
       Set<Student> likes;

       // additional properties
       // standard constructors, getters, and setters
   }
```

In JPA,for modelling a many-to-many relationship we should include a Collection in both classes, which contains the elements of the others. After that, we need to mark the class with @Entity, and the primary key with @Id to make them proper JPA entities. Also, we should configure the relationship type. Hence we mark the collections with @ManyToMany annotations.

Annotations(I)

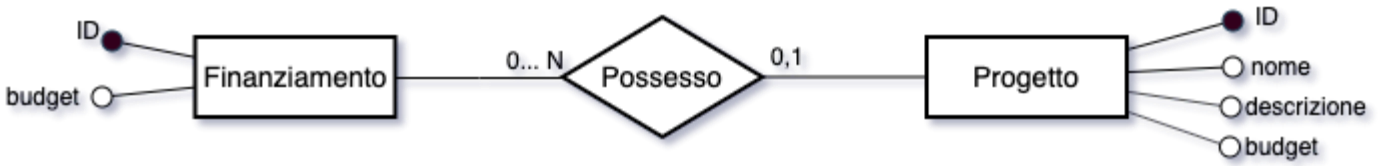
We have also to configure how to model the relationship in the RDBMS.

```
@ManyToMany
   @JoinTable(
     name = "course_like",
     joinColumns = @JoinColumn(name = "student_id"),
     inverseJoinColumns = @JoinColumn(name = "course_id"))
   Set<Course> likedCourses;
```

```
@ManyToMany(mappedBy = "likedCourses")
   Set<Student> likes;
```

The owner side is where we configure the relationship, which for this example we'll pick the Student class.
We can do this with the @JoinTable annotation in the Student class. We provide the name of the join table (course_like), and the foreign keys with the @JoinColumn annotations. The joinColumn attribute will connect to the owner side of the relationship, and the inverseJoinColumn to the other side. On the target side, we only have to provide the name of the field, which maps the relationship. Therefore, we set the mappedBy attribute of the @ManyToMany annotation in the Course class.
Since a many-to-many relationship doesn't have an owner side in the database, we could configure the join table in the Course class and reference it from the Student class.

One-to-many relationship



In this example, present in our application, we have that each Project can have <u>many</u> Finanziamento, while each Finanziamento is done on only <u>one</u> Project.

Entities

```java
@Entity
@Table(name = "finanziamento", schema = "esercizio1", catalog = "")
public class FinanziamentoEntity {
    private int id;
    private Integer budget;

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)

      //get and set

    @Basic
    @Column(name = "budget", nullable = true)

      //get and set

      //…more code…
```

```java
@Entity
@Table(name = "progetto", schema = "esercizio1", catalog = "")
public class ProgettoEntity {
    private int id;
    private String nome;
    private String descrizione;
    private Integer budget;
    private AziendaEntity azienda;

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
       //get and set
    @Basic
    @Column(name = "nome", nullable = true, length = 500)

       //get and set

      //…more code…
```

This is the representation of the traduction as JPA Entities of the table Progetto and Finanziamento without considering the Possesso relationship

Annotations(I)

Referring to JPA's Best Practices it would in any case be preferable to implement Possession as a two-way relationship, even if an actual discriminant depends on the cases of use of the application.
If in use cases there was no need to actually keep the two entities connected, or to keep them connected only in one direction, it might be convenient to exploit a one-way relationship.

```
public class FinanziamentoEntity {

//…more code…

private ProgettoEntity progetto;
@ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "progetto_id", nullable = false)
    public ProgettoEntity getProgetto(){ return this.progetto ;};
    public void setProgetto(ProgettoEntity _progetto){ this.progetto
= _progetto; };

//…more code…


}
```

```
public class ProgettoEntity {

//…more code…

private List<FinanziamentoEntity> myStakes;

@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch =
FetchType.LAZY, mappedBy = "progetto")
    public List<FinanziamentoEntity> getMyStakes(){ return
this.myStakes ;};
    public void setMyStakes(List<FinanziamentoEntity> _stakes){
this.myStakes = _stakes; };

//…more code…
}
```

If you need to know which project is linked to each funding you need to use the annotation @ManyToOne. In this way it will be possible to create a column, in the financing table, "progetto_id" that contains the key of the Project entity, through the annotation @JoinColumn. To achieve this you need to insert an object of type ProgettoEntity in the class FinanziamentoEntity that is instantiated each time a project is created.

Annotations(II)

In case we need to keep in memory all the funds owned by a single project, to be able to access them through the entity ProjectEntity instead you need to use the annotation @OneToMany.
In this way, the entity ProgettoEntity contains within it a list of financing (also instantiated as a "Set") that represents all the financing owned by it. The FinanziamentoEntity objects belonging to the list are also mapped through the property "mappedBy" by the project attribute (ProgettoEntity) present in FinanziamentoEntity, thus creating a bidirectional relationship.
In that case the attribute cascade must be setted to CascadeType.All because if we delete a project, we also have to delete all the funds related to it.

Annotations: Solution adopted

For the optimization of the client side memory it would be convenient to use only a one-way relationship because there are situations, when you access to a project, in which you are not interest in the fundings made on it. In this way we prefer to do a further call to the database on the Finanziamento table, using the project_id field relative to the project of which we are interested in.

Create

A FinancingEntity instance is built as a normal Java object. To insert this object into the database, an i.e. transaction is opened first. e.getTransaction().begin()
A call to this function associates the 'Object' object to an EntityManager and changes its status to Managed.
The new object will then be stored in the database when the transaction commit is called. The entity manager will then be closed.

```
public <T> T create( T entity ){
        if( entity == null || em == null)
            return null;

        try {
            em.getTransaction().begin();
            em.persist(entity);
            em.getTransaction().commit();
            return entity;
        } catch(Exception ex){
            ex.printStackTrace();
            return null;
        }
    }
```

The persist() function throws a TransactionRequiredException exception if there is no active transaction and a type IllegalArgumentException if the argument of the function is not an instance of a class, in fact only instances of entity classes can be stored within the database.

Read

Each object can be identified and retrieved within the database using its class and primary key.
To read an object from the database it is possible, once a transaction is opened, to use the find() function that will return an instance of the same class passed as first argument. The entity manager will then be closed by the caller of that function.

```
public <T> T read(Class<T> tableClass, String id) {
        if (em == null)
            return null;

        T result = null;

        try {
            em.getTransaction().begin();
            result  =  em.find(tableClass, id);
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            em.getTransaction().commit();
            return result;
        }
}
```

In this case the function throws an exception of type IllegalArgumentException if the class passed as argument is not an Entity Class.

Update

Before calling this function that will update the entity in the database, the object is first taken and passed as an argument.
A transaction is opened and then merged the entity -i.e. em.merge(entity)- with what is already in the database, updating it accordingly.
Finally, the transaction is committed and the entityManager is closed by the caller of that function.

```
public <T> T update(T entity){
        if( entity == null || em == null )
            return null;

        try{
            em.getTransaction().begin();
            T result  = em.merge(entity);
            em.getTransaction().commit();
            return result;
        }catch(Exception ex){
            ex.printStackTrace();
            return null;
        }
    }
```

The function throws an exception of type IllegalArgumentException if the class passed as argument is not an Entity Class.

Delete

The argument for this function is the type and primary key of the object to be deleted from the database. This two pieces of information are used to retrieve the object once a transaction has been opened. This object is then passed to the construct remove -i.e. em.remove(old)- through which the object will actually be removed deleted once the transaction is committed. The entity manager will finally be closed by the caller of the transaction.

```
public <T> T delete(Class<T> type, int id) {
        if (em == null)
            return null;
        try {
            em.getTransaction().begin();
            T old = em.getReference(type, id);
            em.remove( old );
            em.getTransaction().commit();
            return old;
        }catch(EntityNotFoundException ex){
            return null;
        }
        catch (Exception ex) {
            ex.printStackTrace();
            return null;
        }
    }
```

If there is no object with such a class and primary key, an exception of the type EntityNotFoundException