

# Enterprise Programming 1

## Lesson 02: JPA

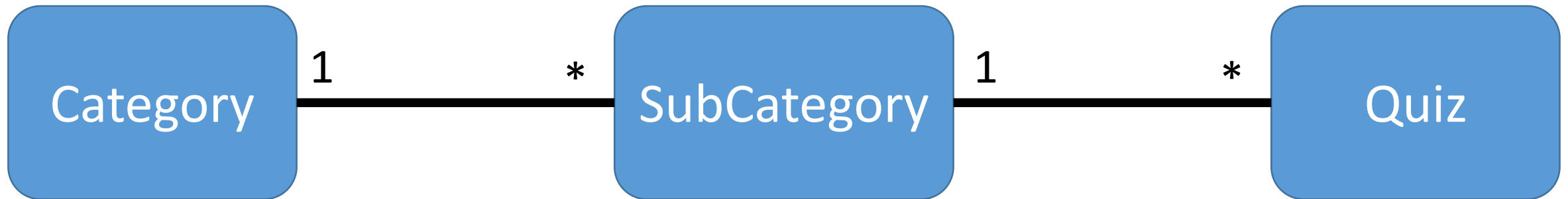
Prof. Andrea Arcuri

# About these slides

- These slides are just high level overviews of the topics covered in class
- The details are directly in the code comments on the Git repository

# Relationships

Database (DB) tables can have relationship among them



- A category can have many subcategories
- A subcategory has one parent category
- Same kind of relations between SubCategory and Quiz
- “Links” are *foreignkey* constraints

# Relationship Annotations

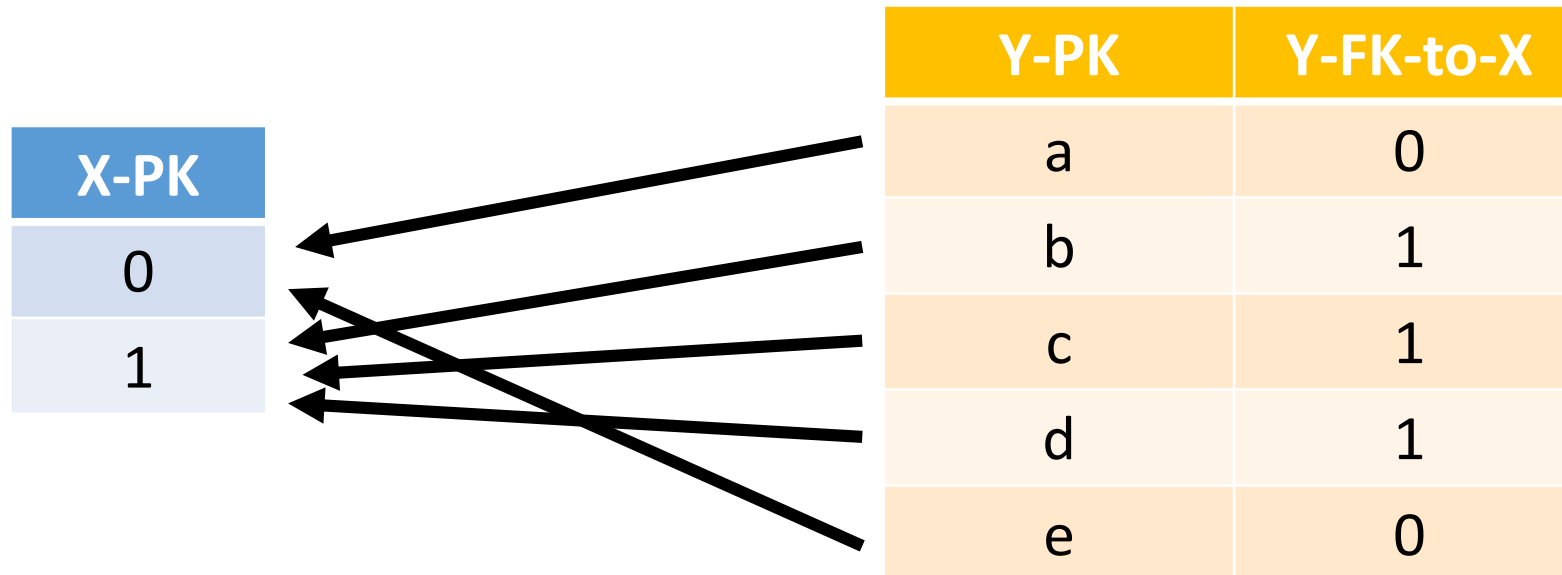
- 5 kinds of annotations
- *@OneToOne*
- *@OneToMany*
- *@ManyToOne*
- *@ManyToMany*
- *@ElementCollection*

# Links

- “Links” are represented with SQL *foreign-key* constraints
  - ie, a field in table X is mapped with the value of primary-key in Y
- If X has link to Y, then need to decide if Y has link back to X
  - eg, *unidirectional* or *bidirectional* links
- In annotations @, this is specified with *mappedBy*
  - if forget it, might end up with independent links, eg X x with link to a Y y, where the link back from y could point to a different X  $k \neq x$

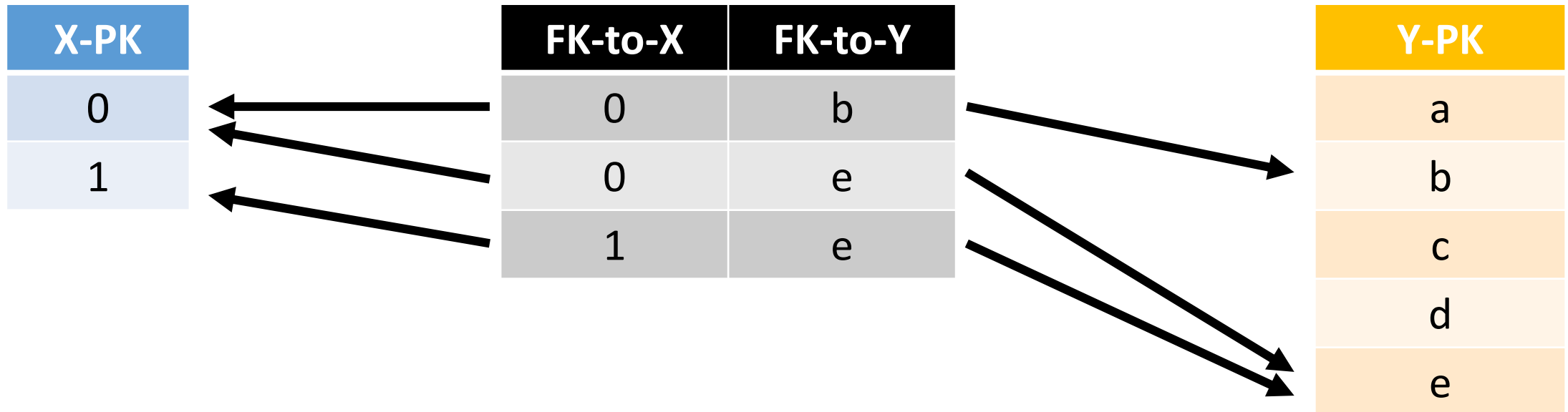
# 1-To-Many

- If a  $X$  has links to many  $Y$ s, the FK will be from  $Y$  to  $X$
- The table  $X$  has no info on  $Y$ , but can use SQL to find all rows in  $Y$  that has FK pointing to a specific  $X$
- Here uni/bi-directionals will use the same SQL tables



# Many-to-Many

- If an *X* element has links to many *Y*s, and a *Y* can have links to many *X*s, we need a third table with 2 FKs
- Eg, *0* has link to *b* and *e*, where *e* also has link to *1*



# EntityManager

- Object used to sync the entities with the data in the DB
- Different operations
  - *persist()*
  - *clear()*
  - *find()*
  - *contains()*
  - *merge()*
  - *remove()*
  - etc.



# Java Persistence query language (JPQL)

- You can use *EntityManager#find(id)* to query an *@Entity* with a given *id*
- But what if you need to find all quizzes in a given category?
- You can of course use SQL
- JPQL: similar to SQL in syntax, but works by referring directly to *@Entity*, and not tables in DB
- JPA will translate JPQL into SQL at runtime

# JPQL Example

```
select u from User u where u.address.country = 'Norway'
```

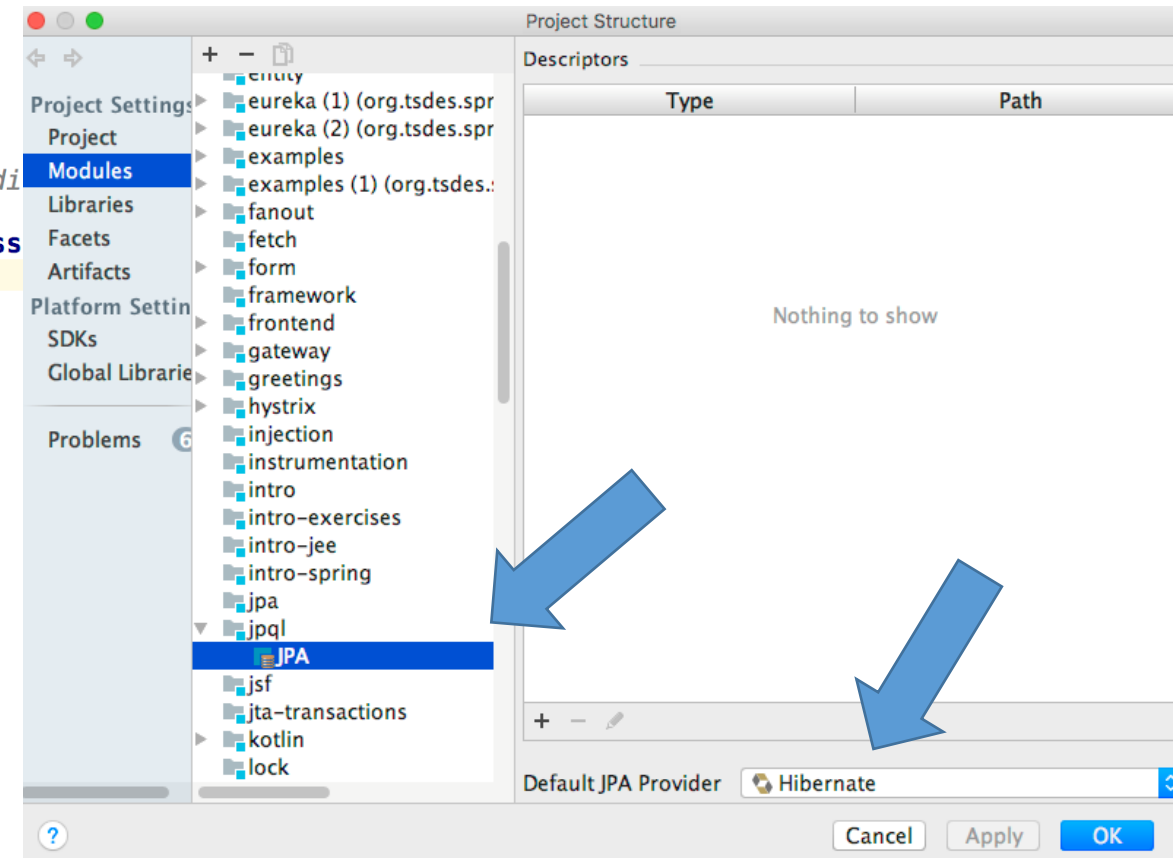

- Very similar to SQL, eg, using SELECT/FROM/WHERE
- However, the FROM is an *@Entity*
- We give a name to the entity, eg “*u*” (but could be anything)
  - that works as instance of the *@Entity*, on which we can access fields like an object

# JPQL and IntelliJ

IntelliJ can automatically analyze syntax and do code completion for JPQL strings, but need configuration

```
@Test
public void testGetAllWithOnTheFlyQuery() {
    //you can create queries on the fly. but if a query is used in a lot of di
    //places, it might be best to use a named one
    TypedQuery<User> query = em.createQuery("select u from User u", User.class);
    List<User> users = query.getResultList();

    assertEquals(4, users.size());
}
```



# Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **intro/jee/jpa/relationship**
- **intro/jee/jpa/relationship-sql**
- **intro/jee/jpa/manager**
- **intro/jee/jpa/jpql**
- **intro/jee/jpa/fetch**
- Exercises for Lesson 02 (see documentation)