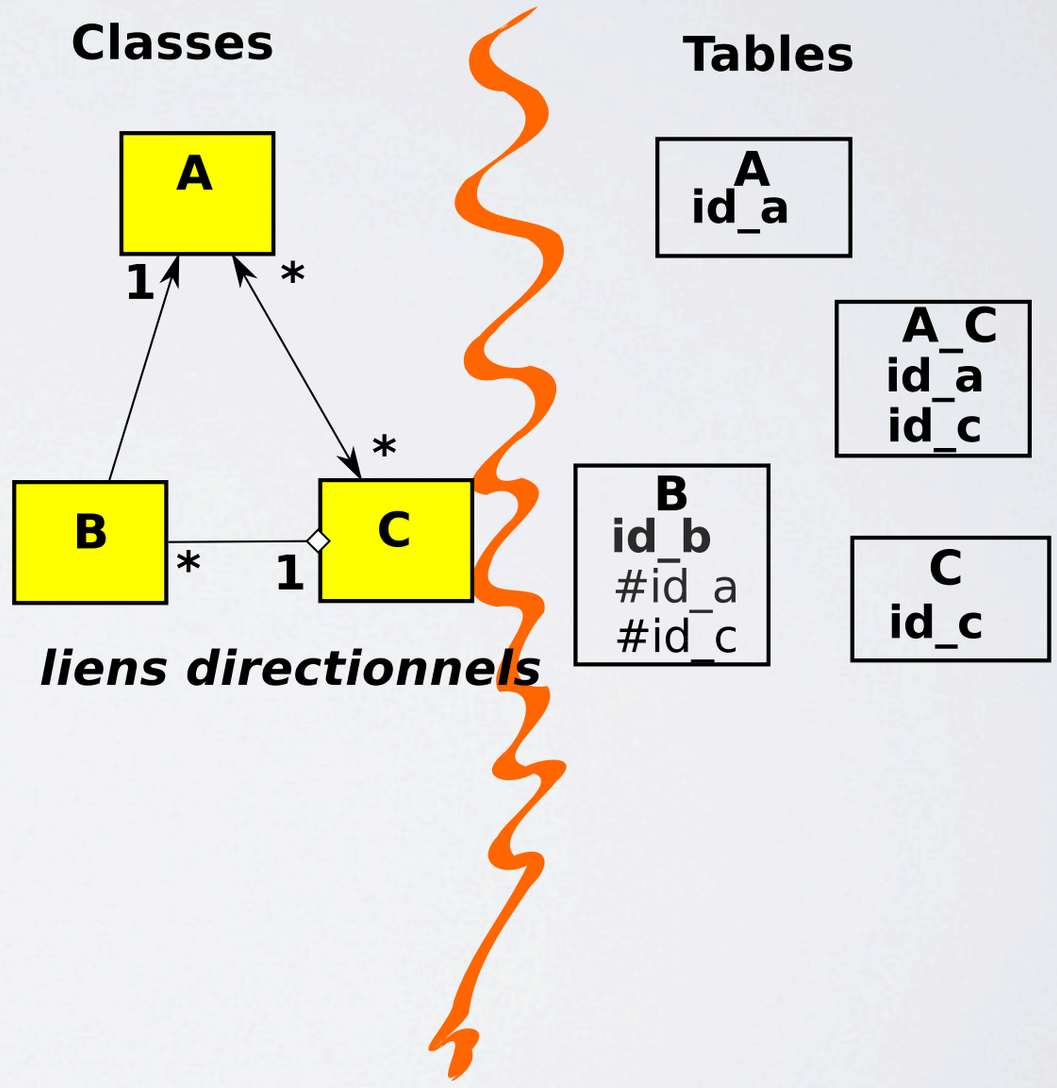
JPA

Saving objects in java

OBJECT/RELATIONAL MAPPING

- Solve the so-called «object-relational mismatch»
- Java Persistence API : general API
- Various implementations : Hibernate, TopLink...



JPA PRIMER

- Everything can be done with software binding
- Normally, configuration kept in META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
        xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/
ns/persistence/persistence_1_0.xsd">
  <!-- Definition and name of a "persistence unit" -->
  <persistence-unit name="EssaiNetbeansPU
    <!-- List of entity classes -->
    <class>essainetbeans.model.Prof</class>
    <class>essainetbeans.model.Cours</class>
    <!-- Connection information -->
    <properties>
      <property name="hibernate.connection.url" value="jdbc:derby://localhost:1527/cnam"/>
      <property name="hibernate.connection.driver_class"
value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="hibernate.connection.password" value="test"/>
      <property name="hibernate.connection.username" value="test"/>
      <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.NoCacheProvider"/>
    </properties>
  </persistence-unit>
</persistence>
```

ENTITIES/PERSISTENT OBJECTS

- have a mandatory ID

- changes to an entities in memory will cause changes in the database

- Need for a mapping between java code and SQL. Use implicit data when possible.

PERSISTANT OBJECTS DECLARATION

- XML declarations (for Hibernate)

- annotations (simpler)

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-
mapping-3.0.dtd">
<hibernate-mapping package="demoHibernate.model">
   <class name="Article">
      <id name="id" column="idArticle">
         <generator class="native"/> <!-- ou increment -->
      </id>
      <property name="designation" column="description"/>
      <property name="prix"></property>
   </class>
</hibernate-mapping>
```

USING ANNOTATIONS

```
@Entity
@Table(name = "PROF")
public class Prof{
    // Mandatory definition of ID :
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "PROF_ID")
    private Integer profId;

    @Column(name = "PROF_NAME")
    private String profName;

    @Column(name = "PROF_FIRST_NAME")
    private String profFirstname;

    // At least one default constructor (may be private)
    public Prof() {
    }

... rest of the class: POJO style.
```

NOTE :

- the existence of an ID is mandatory (may be compound)
- Ids are quite flexible, though
- The ID setter might be «private» (hence, only hibernate/jpa can manipulate it)

```
// We create an EntityManager factory
// using the name of the one defined in
// META-INF/persistence.xml
   EntityManagerFactory entityManagerFactory =
      Persistence.createEntityManagerFactory("EssaiNetbeansPU");
// Then we create the entity manager
   EntityManager entityManager=
entityManagerFactory.createEntityManager();
// we start a transaction
   EntityTransaction entityTransaction=
                  entityManager.getTransaction();
   entityTransaction.begin();
// We create an entity
   Prof p= new Prof();
   p.setProfFirstname("Olivier");
   p.setProfName("Pons");
// We save it
   entityManager.persist(p);
// We commit the transaction
   entityTransaction.commit();
// Closing everything
   entityManager.close();
   entityManagerFactory.close();
```

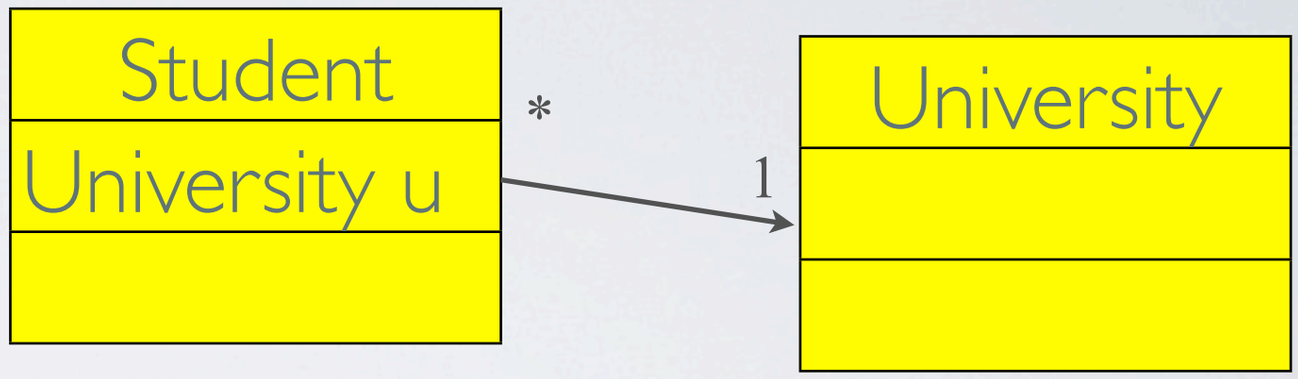
SIMPLE DEMO

SEAMLESS OBJECTS UPDATE

```
EntityManagerFactory entityManagerFactory =
        Persistence.createEntityManagerFactory("EssaiNetbeansPU");
EntityManager entityManager=
        entityManagerFactory.createEntityManager();
EntityTransaction entityTransaction= entityManager.getTransaction();
entityTransaction.begin();
// Find the object of class «Compteur» with ID 0
Compteur compteur= entityManager.find(Compteur.class, 0);
// Print it and change its value...
System.out.println("*** VALEUR : "+compteur.getValeur());
compteur.setValeur(compteur.getValeur() + 1 );
//Done! the data will be saved in the Database. Now close:
entityTransaction.commit();
entityManager.close();
entityManagerFactory.close();
```

MORE ANNOTATIONS

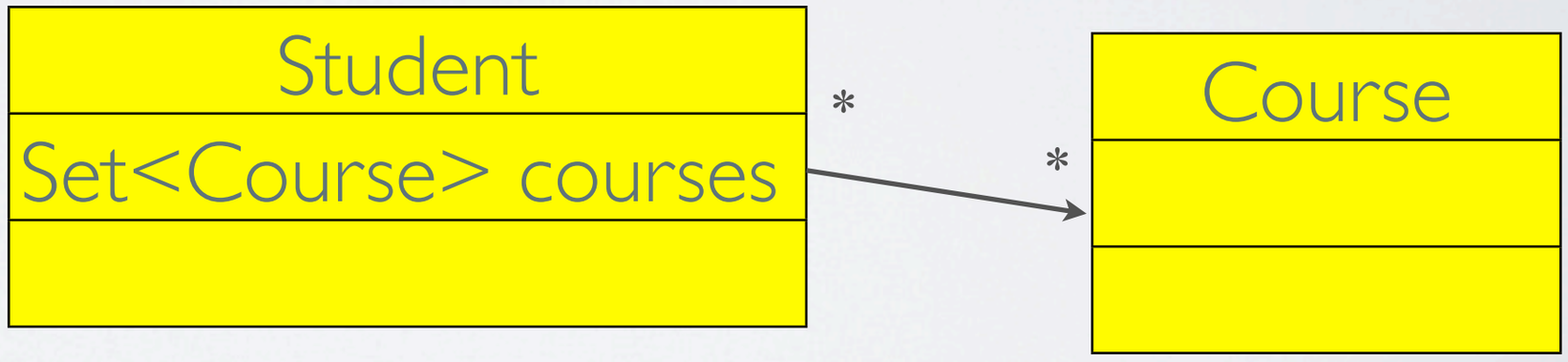
ONE-WAY LINKS



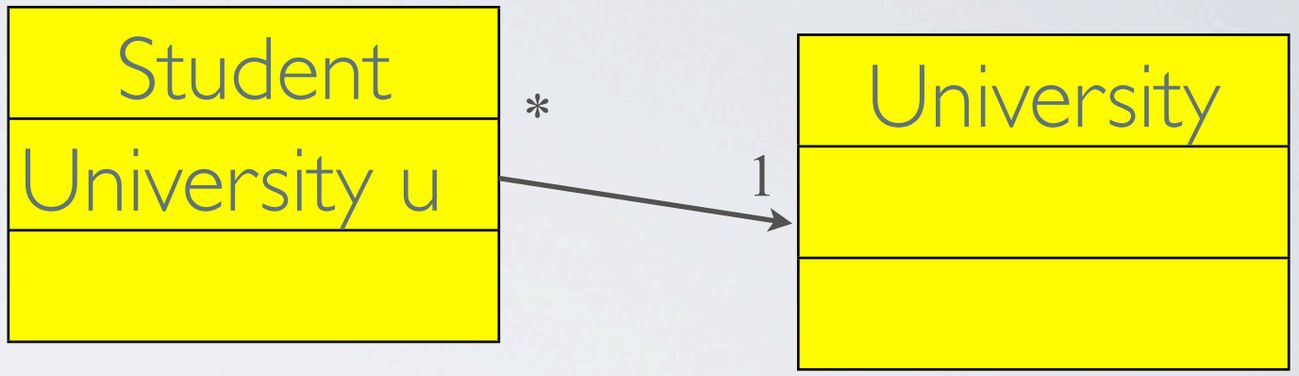
- Simpler to do in Java (only one side)

- Less coupling

- 1..* or n..n



MANY-TO-ONE ONE-WAY LINK



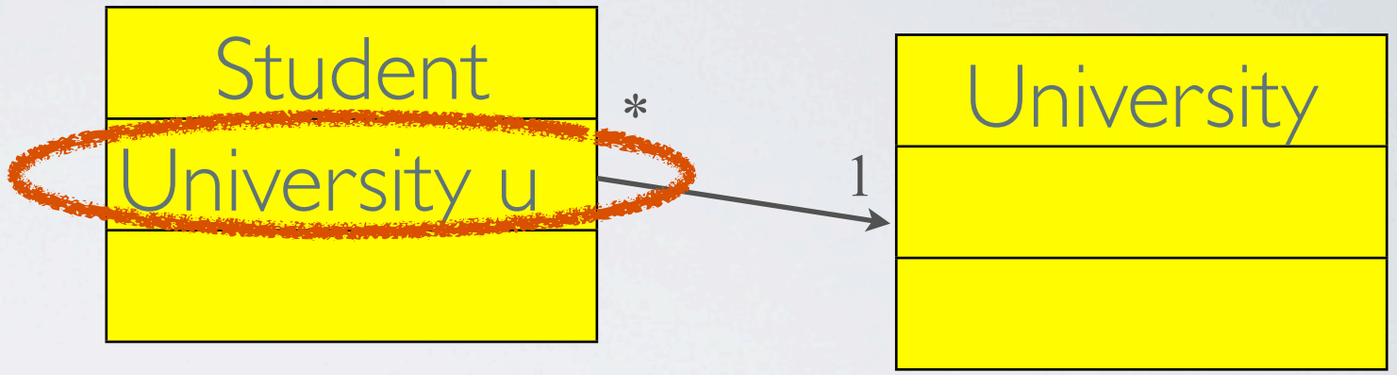

```
@Entity
public class Student implements Serializable {
    @Id
    @Column(name = "STUDENT_ID")
    private Long studentId;


    @ManyToOne
    @JoinColumn(name="UNIVERSITY_ID")
    private University university;
    ...
}
```

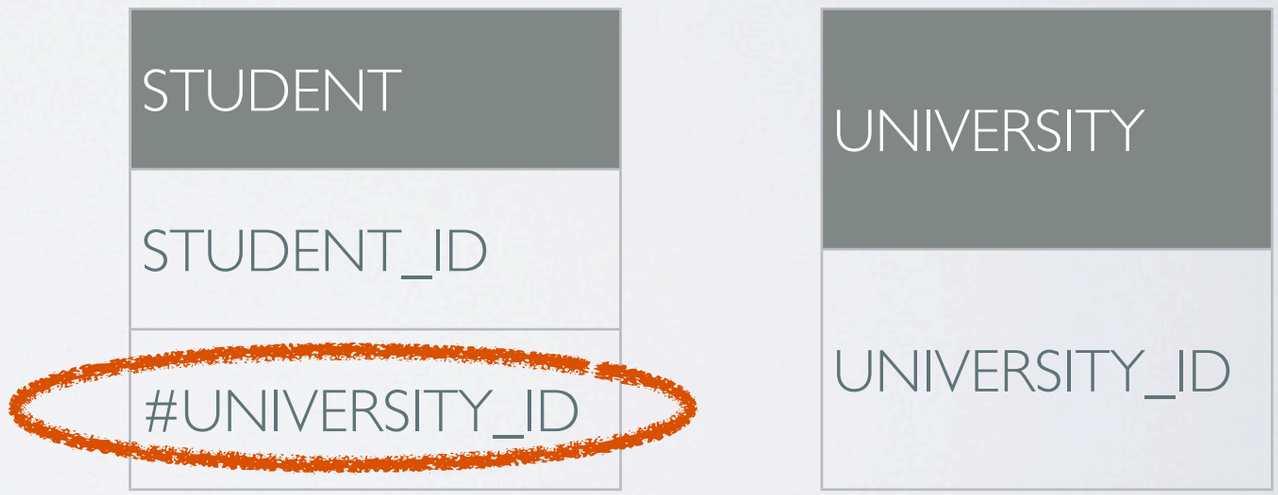
STUDENT
STUDENT_ID
#UNIVERSITY_ID

UNIVERSITY
UNIVERSITY_ID

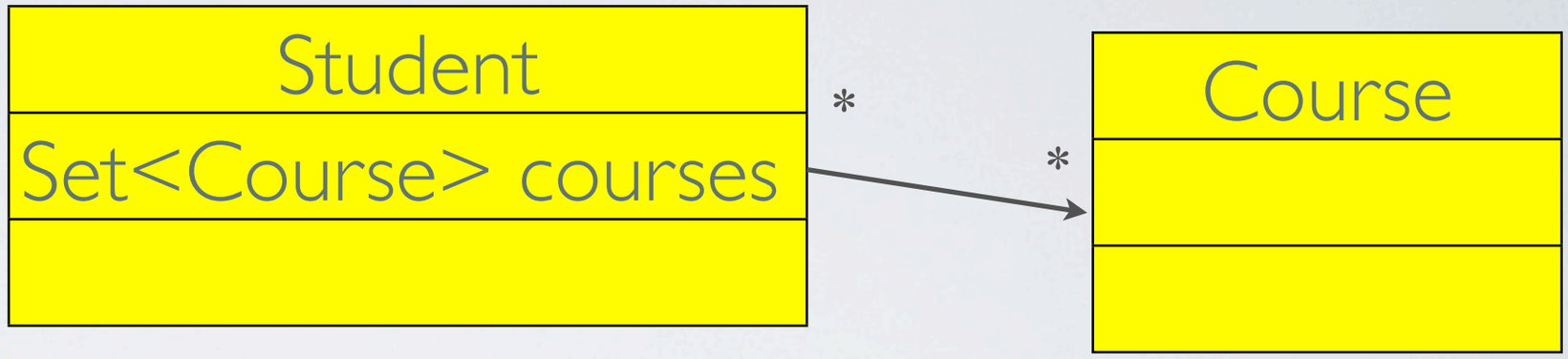
MANY-TO-ONE ONE-WAY LINK

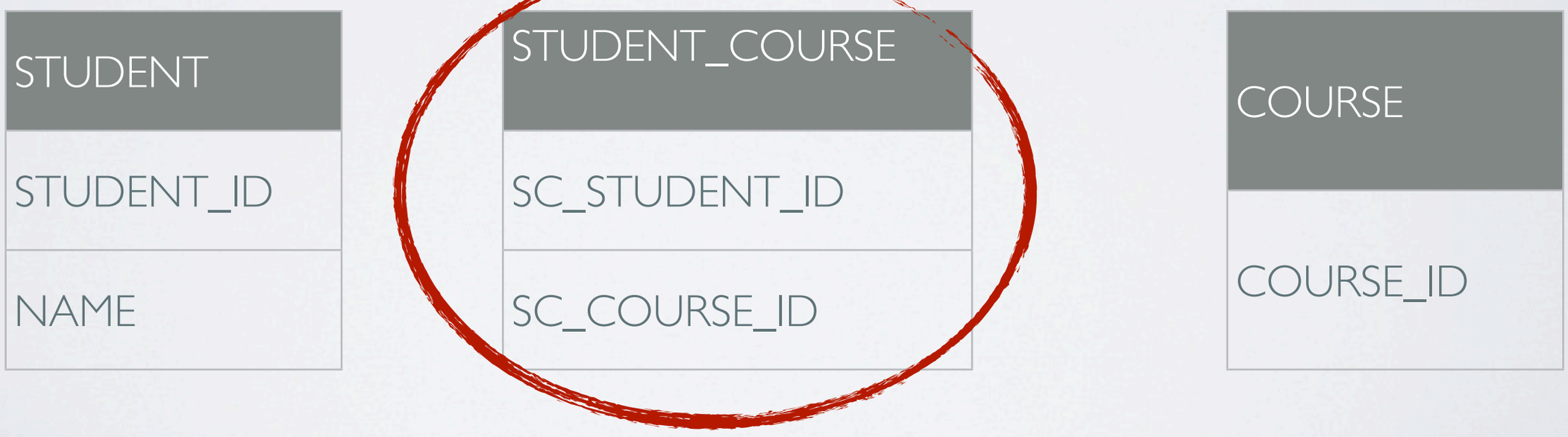


- NOTE : the java class which maintain the relation corresponds to the TABLE which contains the foreign key: here STUDENT

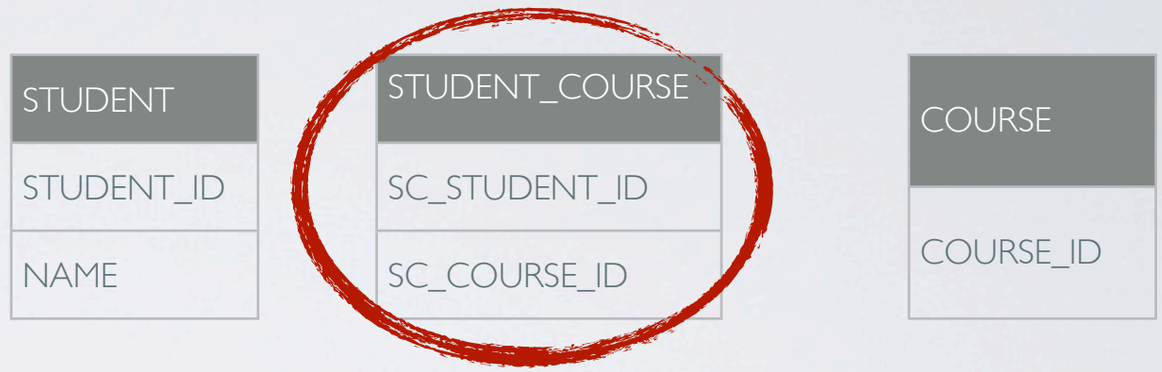


MANY-TO-MANY ONE-WAY LINK



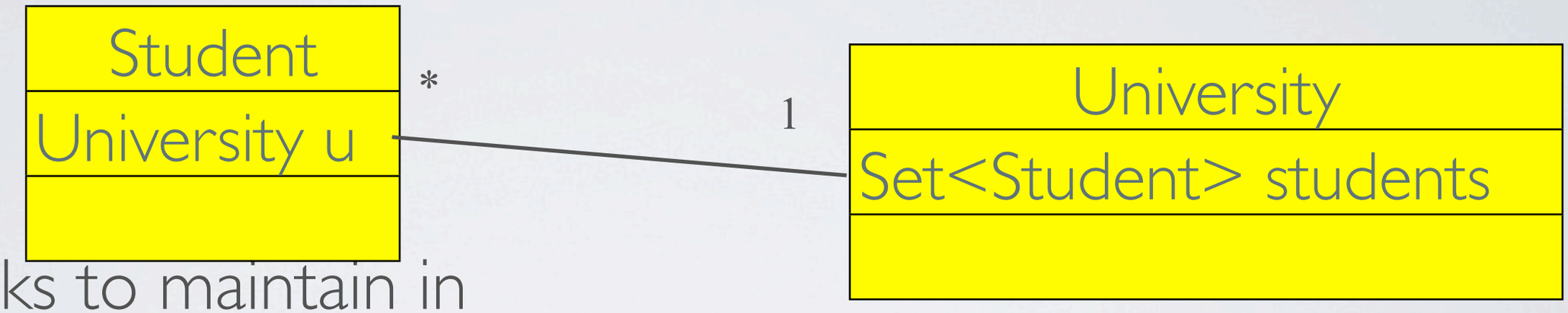


MANY-TO-MANY ONE-WAY LINK



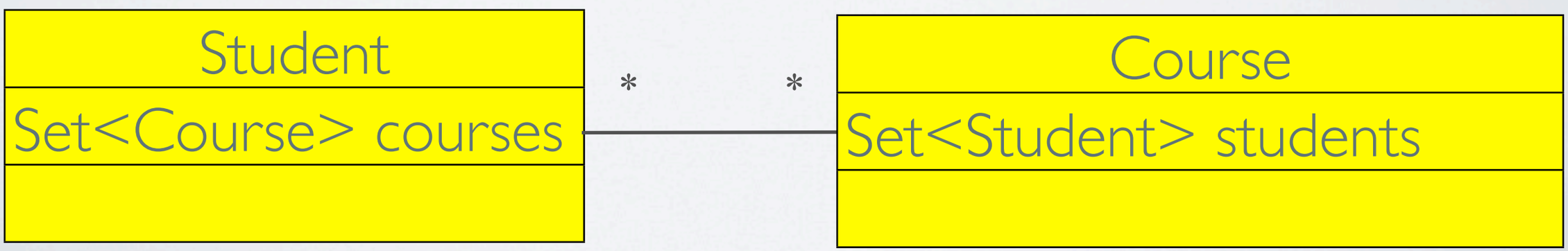

```
@Entity
public class Student implements Serializable {
    ...
    @ManyToMany
    @JoinTable(name = "STUDENT_COURSE",
        joinColumns = @JoinColumn(name = "SC_STUDENT_ID"},
        inverseJoinColumns = @JoinColumn(name = "SC_COURSE_ID"))
    private Collection<Course> courses;
    ...
}
```

BIDIRECTIONAL LINKS

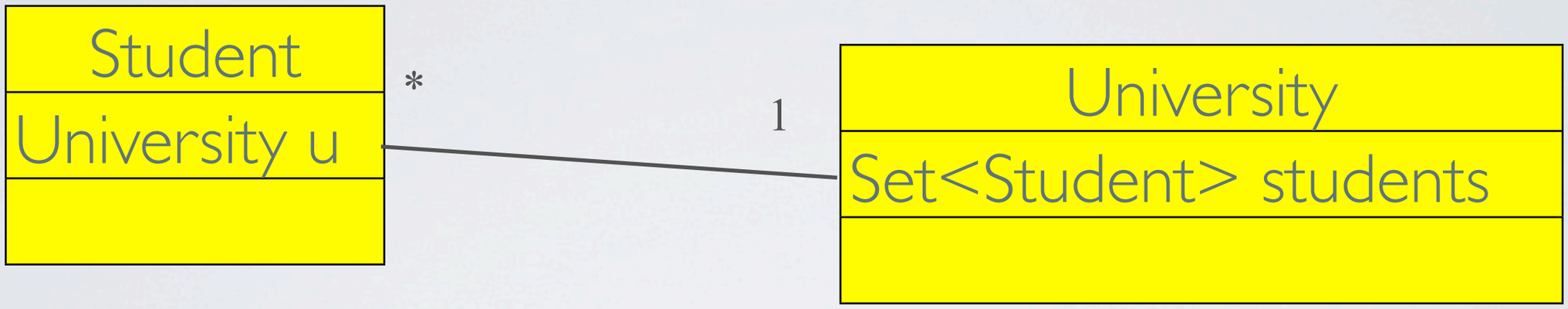


- Two links to maintain in Java

- Solution: break the symetry : one master side



N..1 BIDIRECTIONAL




```
@Entity
public class Student implements Serializable {
    ...
    @ManyToOne
    @JoinColumn(name=
                "UNIVERSITY_ID")
    private University university;
```

secondary attribute.
Note: «university» : name of the variable in the «Student» class.

```
@Entity
public class University implements Serializable {
    ...
    @OneToMany(mappedBy="university")
    private Set<Student> students;
```

INCORRECT JAVA...

MORE JAVA...

```
@Entity
public class Student implements Serializable {
    ...
    @ManyToOne
    @JoinColumn(name="UNIVERSITY_ID")
    private University university;
    ...
    public void setUniversity(University u) {
        if (this.university != null) {
            this.university.removeStudent(this);
        }
        this.university= u;
        this.university.addStudent(this);
    }
```

```
@Entity
public class University implements Serializable {
    ...
    @OneToMany(mappedBy="university")
    private Set<Student> students;

    void addStudent(Student s) {
        students.add(s);
        // s.setUniversity(this); !!!!! INFINITE RECURSION?
    }

    void removeStudent(Student s) {
        students.remove(s);
    }
```

- addStudent : NOT public !!!
- called **only** by setUniversity (else : infinite recursive calls !!)

N..N BIDI LINKS

Master side

```
@Entity
public class Student implements Serializable {
    @ManyToMany
    @JoinTable(name = "STUDENT_COURSE",
        joinColumns = @JoinColumn(name = "SC_STUDENT_ID"),
        inverseJoinColumns = @JoinColumn(name = "SC_COURSE_ID"))
    private Collection<Course> courses;
    ...
}
```

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

```
@Entity
public class Student implements Serializable {
    @ManyToMany
    @JoinTable(name = "STUDENT_COURSE",
          joinColumns = @JoinColumn(name = "SC_STUDENT_ID"},
          inverseJoinColumns = @JoinColumn(name = "SC_COURSE_ID"))
    private Collection<Course> courses;

    public void addCourse(Course c) {
        courses.add(c);
        c.addStudent(this); // addStudent NOT public !
    }
    ...
}
```

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

    void addStudent(Student s) {
        students.add(s);
    }
}
```

OTHER SOLUTION

```
@Entity
public class Student implements Serializable {
    @ManyToMany
    ....
    private Collection<Course> courses;
    public void addCourse(Course c) {
        if (! courses.contains(c)) {
            courses.add(c);
            c.addStudent(this);
        }
}
```

tests stop mutual recursive call !

```
@Entity
public class Course implements Serializable {
    @ManyToMany(mappedBy= "courses")
    private Collection<Student> students;
    ...
    void addStudent(Student s) {
        if (! students.contains(s)) {
          students.add(s);
          s.addCourse(this);
        }
    }
}
```

SUMMARY

- Not too difficult : a few usual patterns
- lots of possibilities : maps, etc... (read documentation)
- easier if you don't fight the JPA logic
- For Bidi links, YOU are in charge of the Java side.

SEARCHES : JPQL

Object oriented SQL-like language

```
// Find all teachers with a name starting in «T».

List<Prof> listeProfs =
                    em.createQuery(
   "select p from Prof p where p.name like :name"
                        ).setParameter("name", "T%")
   .getResultList();

for (Prof p: listeProfs) {
    System.out.println(p);
}
```

SEARCHES : JPQL

```
// Find all teachers for Math

List<Prof> listeProfs =
                    em.createQuery(
    "select p from Prof p JOIN p.subjects s where s.label='math'")
    .getResultList();

for (Prof p: listeProfs) {
    System.out.println(p);
}
```

PART II

Advanced JPA

MORE ON JPQL

- usual form of a request : **select** *data* **from** *source* **where** *conditions*
- select part: a list of object/object references identifiers.

- from part : declares identifiers, and explains from which sources the data is taken
- where par : conditions
- Simple example :
  - `select p from Person p where p.name = 'Turing';`

QUERY RESULT

- if the select contains only one value of type A: a list of A.
- if the select contains multiple values : a list of Object arrays.

```
select s.name, s.age from Student s;
```

- returns a list of Object[ ], where t[0] is the student name, a String, and t[1] is the student age, an Integer.

OBJECT PROPERTIES

- You can access object properties in select :

```
select c from Command c
   where c.client.address.city = 'Paris'
```

JOINS

- needed when a property is a collection.
- allows you to name the collection element

```
select p from
  University u join u.professors p
where u.name= 'cnam';

select u from
  University u join u.professors p
where p.name= 'Trèves';
```

JOINS

- You can combine joins:

```
select p from
  University u
    join u.professors p1 join u.professors p2
where
    p1.name= 'Pollet' and p2.name= 'Trèves'
```

JOINS

- you can cascade joins:

```
select t from
   University u
      join u.department d
      join d.teams t
where u.name= 'cnam'and t.domain='math';
```

CARTESIAN PRODUCT

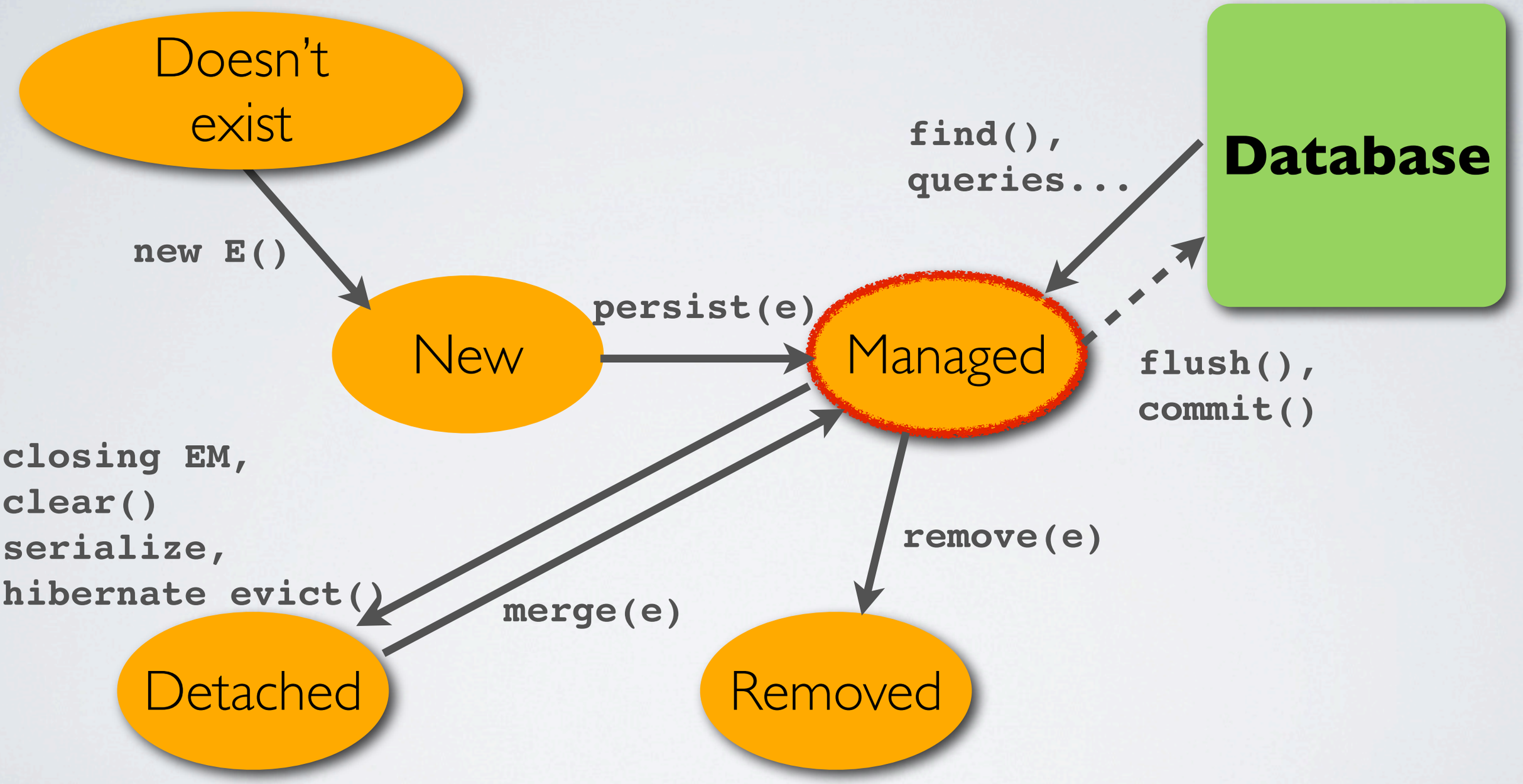
- A query can involve independant classes, and filter through common attributes :

```
select p, c from
  University u, Company c
where u.address.town= c.address.town;
```

THERE'S MORE...

- Lots of other possibilities : aggregate functions,

OBJECTS LIFECYCLE



(From oracle documentation)

LIFE CYCLE

- For a given logical entity, JPA will keep only **one** managed object.
- It means that *if all objects are managed*, «==» gives the expected result for equality.
- Problem: applications might use detached objects

LAZY OBJECT RETRIEVAL

Problem with object models: if the University object contains data about all its students, professors, etc... will

```
select u from University u;
```

load the entire database in memory?

NO

LAZY LOADING: the collections stored in the object are «intelligent»: the connected data will only be fetched when it is accessed.

LAZY LOADING

- Example

```
List<Course> l= (List<Course>) em.createQuery(
  "select c from Course c").getResultList();
for (Course course: l) {
  System.out.println(l.getProfessor.getName());
}
```

- A request will be sent for the professor's data
- no request will be made on the StudentGroup data, although Course has a StudentGroup instance variable too.

PROBLEM WITH LAZY LOADING

- Only works correctly when connected (EclipseLink is able to open the connexion when needed, but it's not transactionnal then)

- N+1 select problem

A NOTE ON EQUALITY

- Example : points
- what if values change?
- problems with sets...
- equals() and hashCode() specifications

ENTITIES VS VALUES

BUSINESS (MODEL) KEY

- examples : user login, ad-hoc key (given at object creation!)

EQUALITY AND JPA

NAÏVE SOLUTION

- generated id-based equality
- problems with sets (and related stuff)

OBJECT IDENTITY-BASED SOLUTION

- descriptions
- limits and detached objects

BUSINESS-KEY BASED SOLUTION

MORE ON MERGE

- Detached objects are not managed by the entity manager
- merge allows you to reattach them

- correct use :
  - Student managedStudent= em.merge(detachedStudent);
  - returns the correct managed instance of the previously detached object.
  - use this system if you want equality to work correctly.

OPTIMISTIC LOCKING

- Avoid concurrent modification *and* incoherent reads at all cost
- Most secure : pessimistic locking, very expensive
- Usually, many reads for one write; very unusual concurrent modification of the *same entry* (two customers may change their own addresses at the same time; the address of customer A won't probably be changed concurrently).
- goal : secure operations. explicitly ***fail*** when concurrent modification is asked.
  - the failed operation is rolled back. Data should be updated.

OPTIMISTIC LOCKING

- Idea : elements have a Version number (or a timestamp)
- When the object is saved, version is checked. If version has changed, the object has been modified by another transaction → **OptimisticLockException**
- In this case, the user should probably be told he must start again.

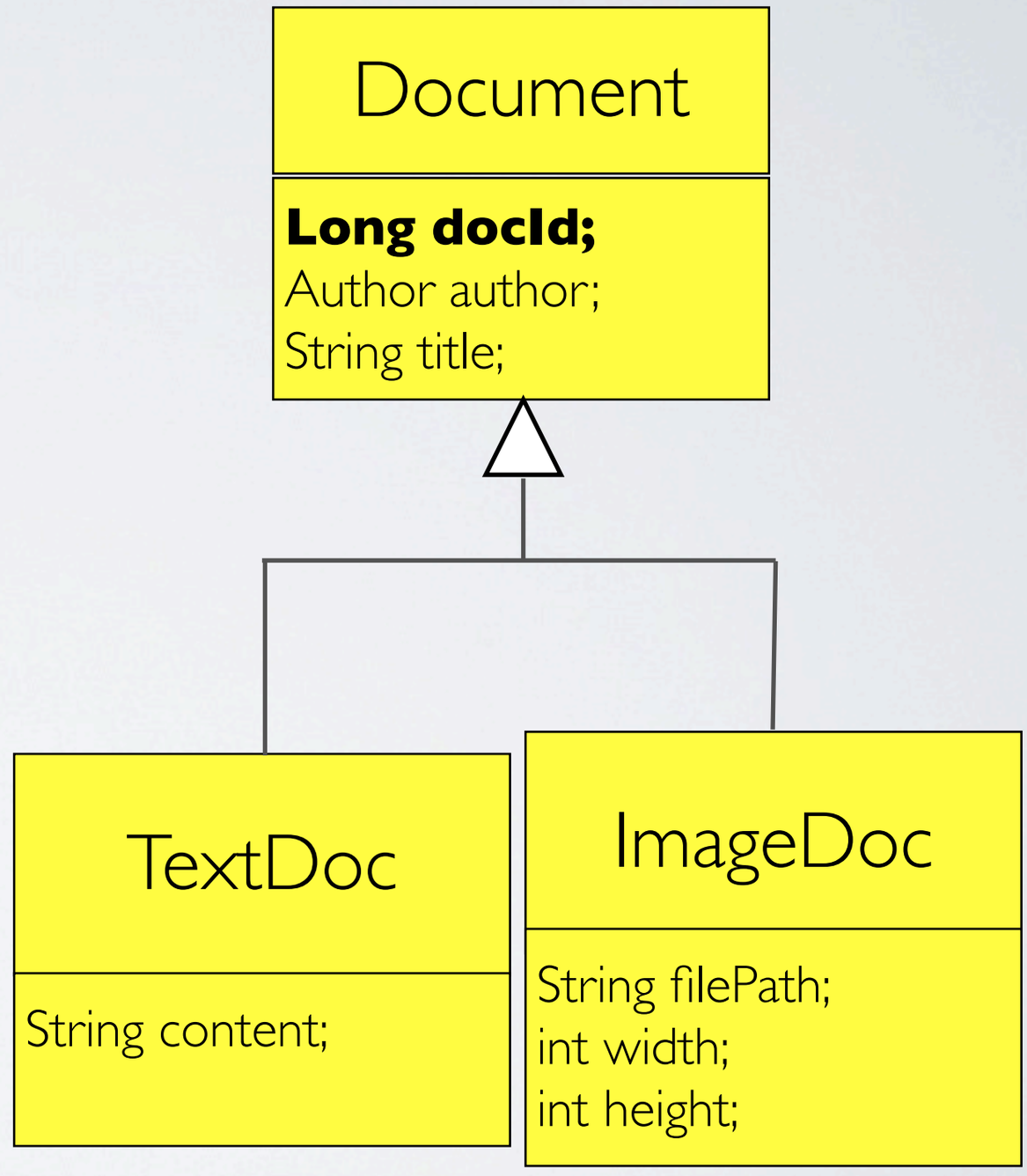
```
@Entity
public class Student {
  @Id
  private Long id;
  @Version
  private long version;
  ...
}
```

Modified with each save

MAPPING INHERITANCE

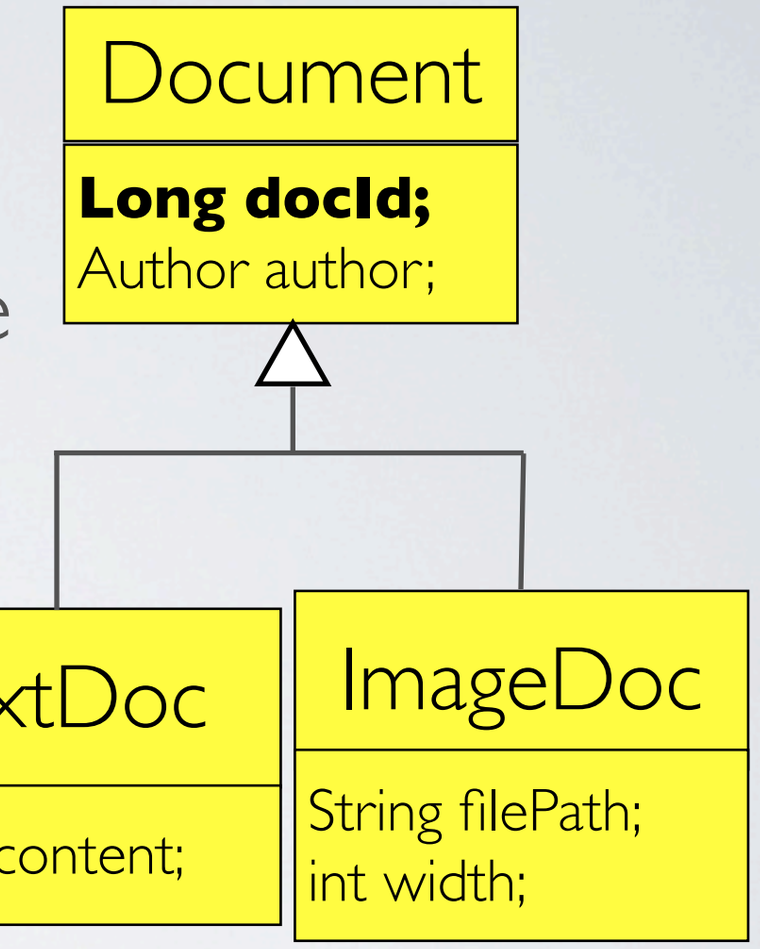
- More than one way to represent inheritance in relational databases

- Important feature: the ID space is the same for all children classes.



MAPPING TO A SINGLE TABLE

- Simple system: store everything in one table
- pro: simple, no duplication
- cons: large table, unused fields

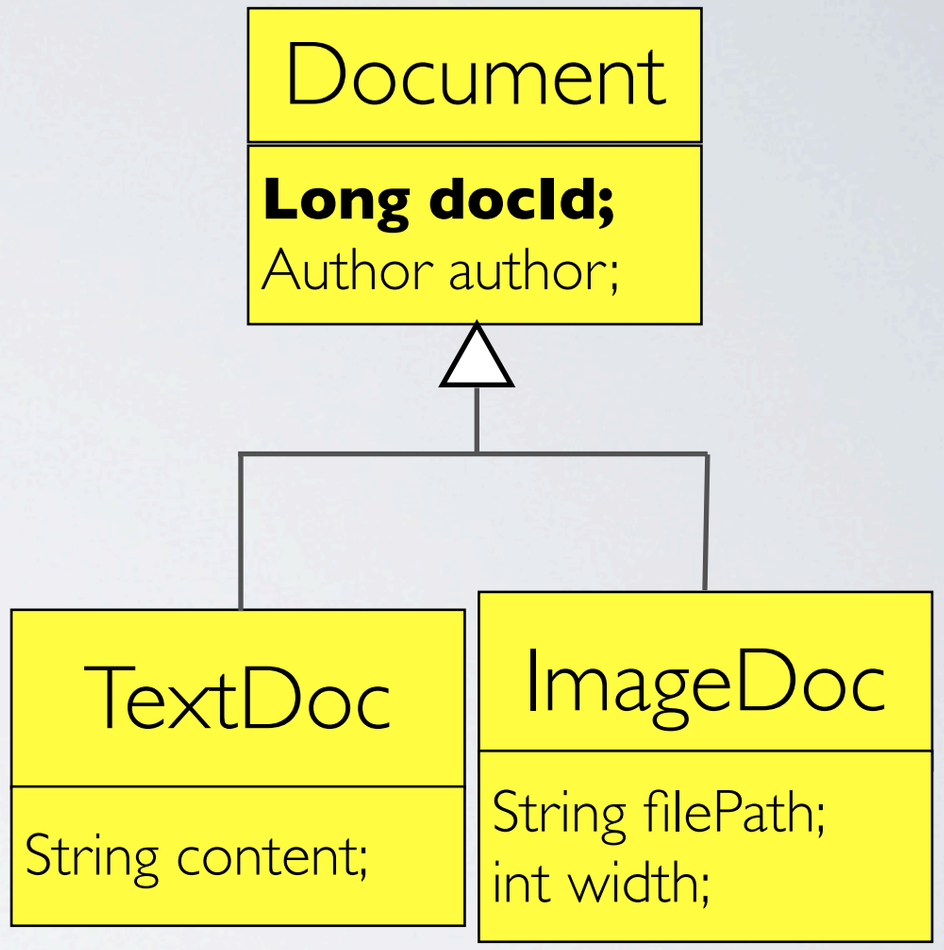


DOCUMENT
docId BIGINT
doctype CHAR
#authorID BIGINT
textContent TEXT
filePath VARCHAR(255)
width INTEGER
height INTEGER

We need to code the actual type of the entry

MAPPING TO ONE TABLE BY CLASS

- Pro: simple
- Cons: duplicate information
- Use if parent class is very simple, with little common information

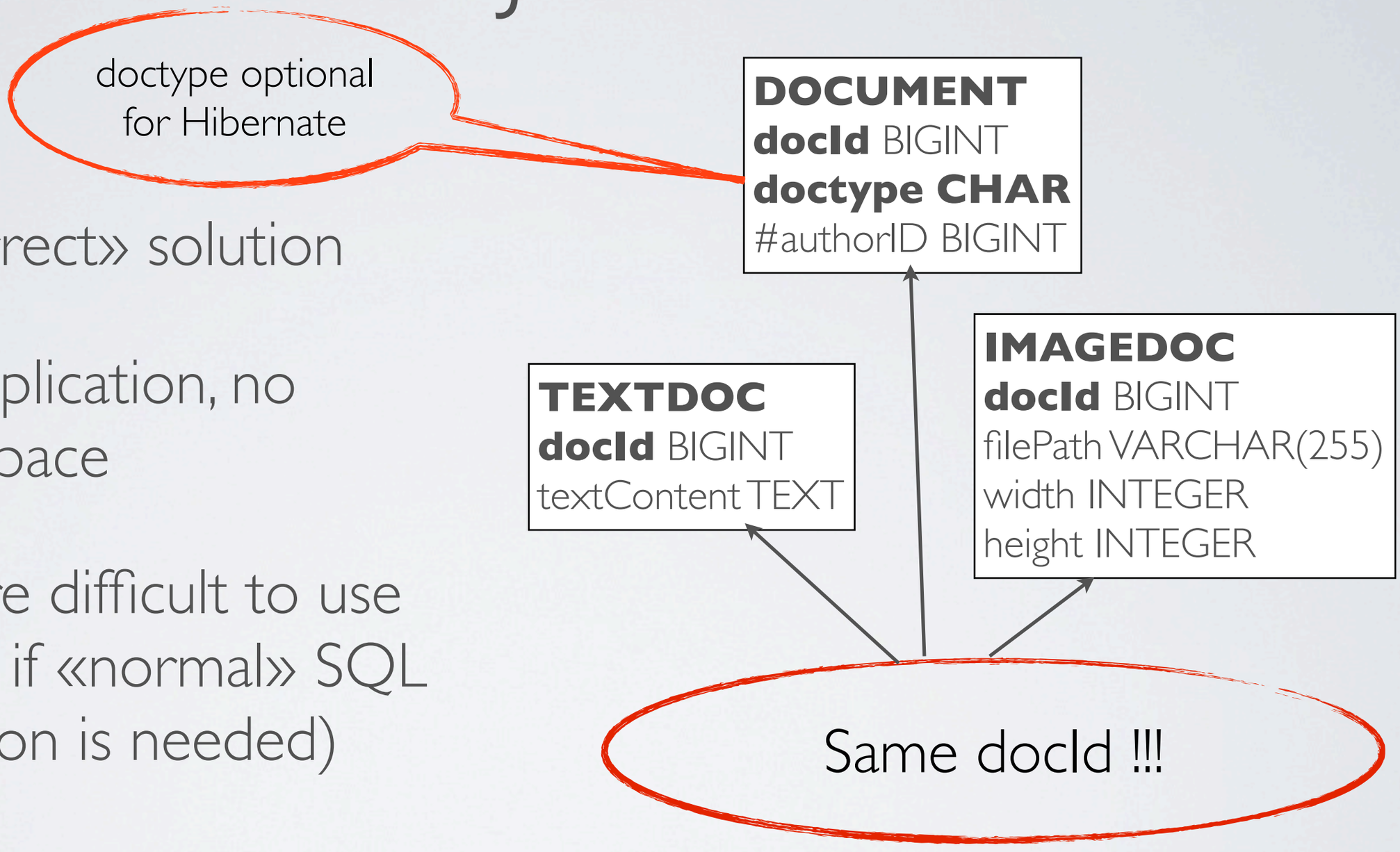


TEXTDOC
docId BIGINT
#authorID BIGINT
textContent TEXT

IMAGEDOC
docId BIGINT
#authorID BIGINT
filePath VARCHAR(255)
width INTEGER
height INTEGER

MAPPING TO JOINED TABLES

- Most «correct» solution
- Pro: no duplication, no waste of space
- Cons: more difficult to use (especially if «normal» SQL manipulation is needed)
- Good when parent class is complex and children classes too.



SINGLE TABLE

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DTYPE")
abstract class Document {
    @Id Long docId;
}
```

only in db. Not an instance variable

```
@Entity
@DiscriminatorValue("txt")
class TextDoc extends Document {

}
```

TABLE_PER_CLASS

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
abstract class Document {
    @Id Long docId;
}
```

```
@Entity
@DiscriminatorValue("txt")
class TextDoc extends Document {

}
```

JOINED TABLES

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="DTYPE")
abstract class Document {
    @Id Long docId;
}
```

```
@Entity
@DiscriminatorValue("txt")
class TextDoc extends Document {

}
```

needed by some implementations (e.g. eclipseLink)

STUFF TO ADD

- about Id, generation and flush (see photo)

EXERCISE : CREATE MODEL CLASSES, DATABASE ENTRIES,

- Create a JPA model and database for the forum model
  - create a database
  - create a persistence unit (toplink)
  - create your Entity objects
- Test it (write a few programs to add users, messages..., list them and change them)
- Change the web application to use your JPA model.