Java Programming 3

Week 8 - Relations - Generics

KdG

Agenda this week

Project review
Implementing relationships in the repository
One-To-One
Many-To-One
Many-To-Many
Generics: implementing a generic repository → this part will be covered later

KdG

Agenda this week

Project review
Implementing relationships in the repository
One-To-One
Many-To-One
Many-To-Many
Generics: implementing a generic repository → this part will be covered later

KdG

Project Review

KdG

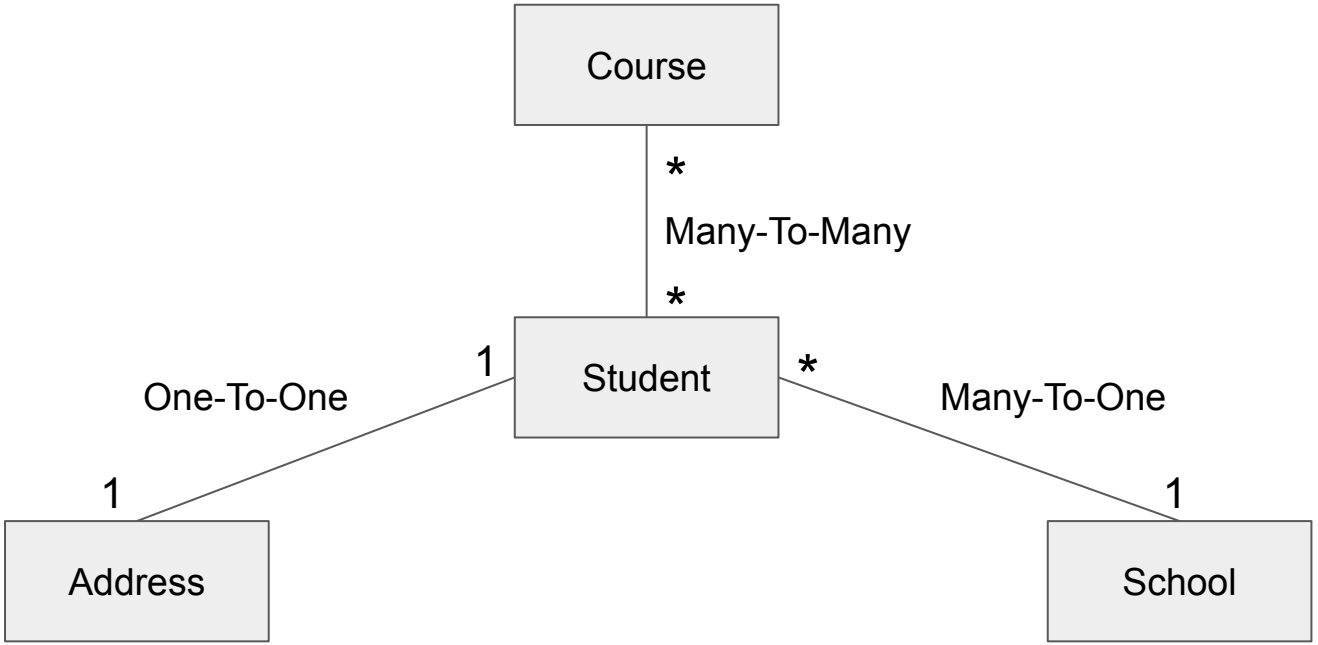
Agenda this week

Project review
Implementing relationships in the repository
One-To-One
Many-To-One
Many-To-Many
Generics: implementing a generic repository → this part will be covered later

KdG

Relationships in the repository

- Student Management System:



KdG

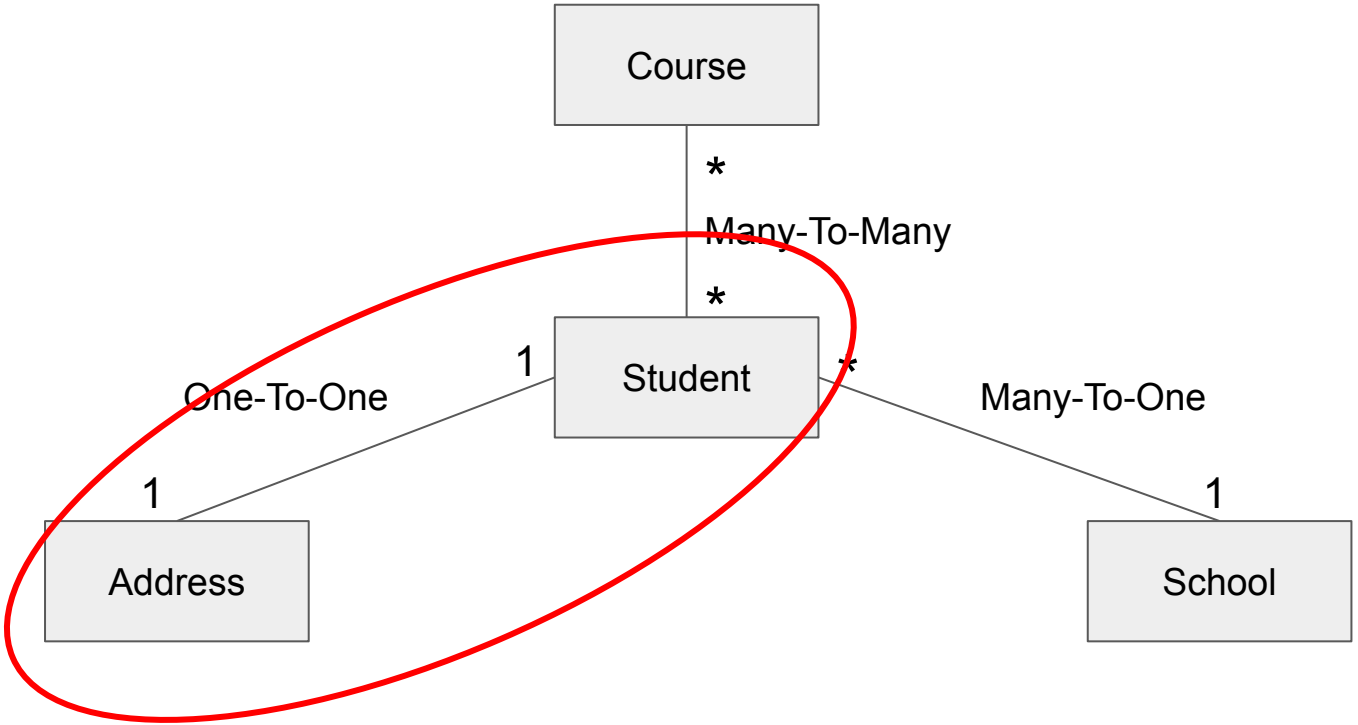
Agenda this week

Project review
Implementing relationships in the repository
One-To-One
Many-To-One
Many-To-Many
Generics: implementing a generic repository → this part will be covered later

KdG

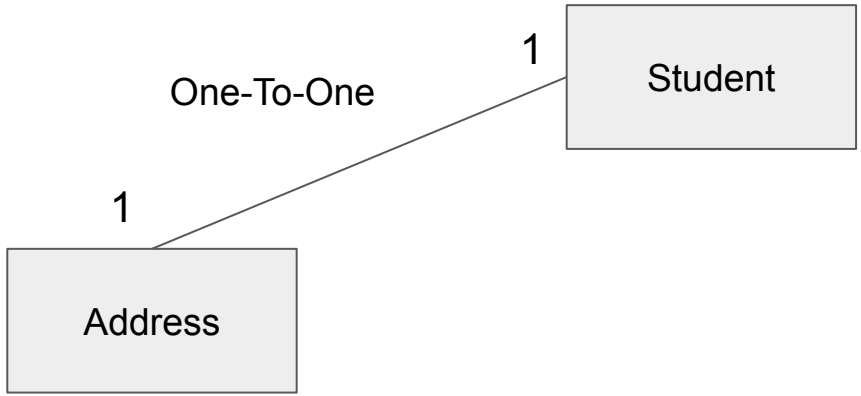
Relationships in the repository

- Student versus Address

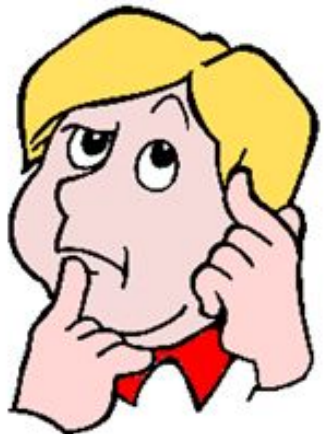


KdG

One-To-One

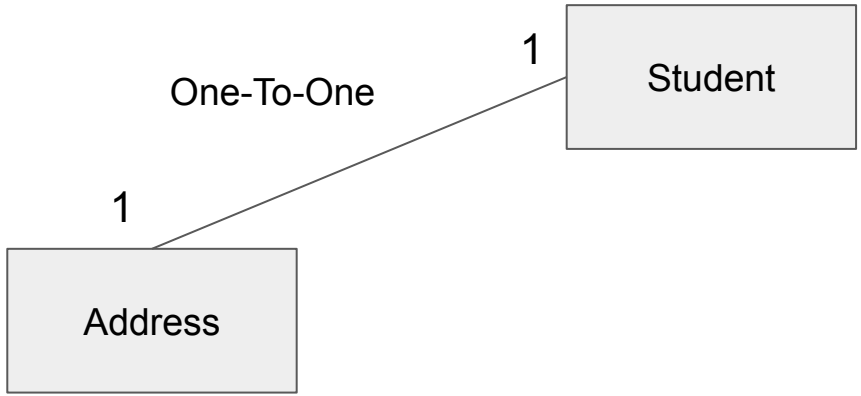


- Student has 1 Address
- Each Address belongs to 1 Student
- The Address is not necessary: a Student can exist without an Address

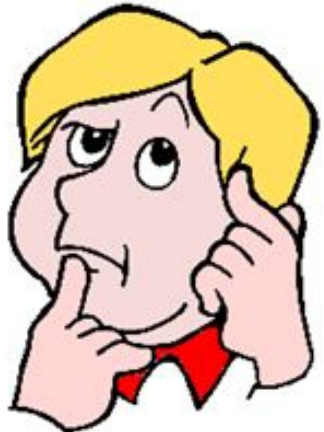


KdG

One-To-One: in Database

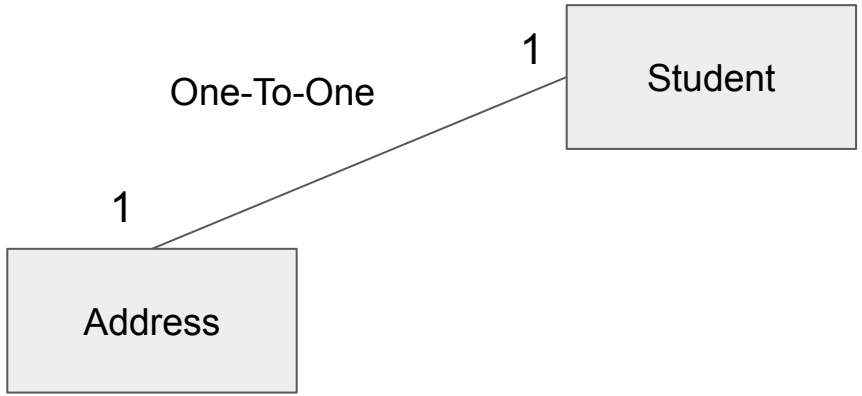


- Different options…
  - Table STUDENT and Table ADDRESS?
    - STUDENT has foreign key to ADDRESS?
    - ADDRESS had foreign key to STUDENT?
    - Separate STUDENT_ADDRESS table with STUDENT_ID and ADDRESS_ID?
    - …?
  - Table STUDENT and extra columns for Address information? (“embedded”)
  - …?

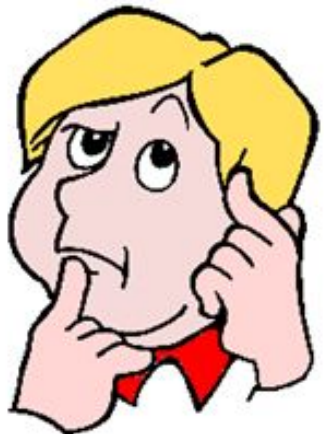


KdG

One-To-One: in Domain Model

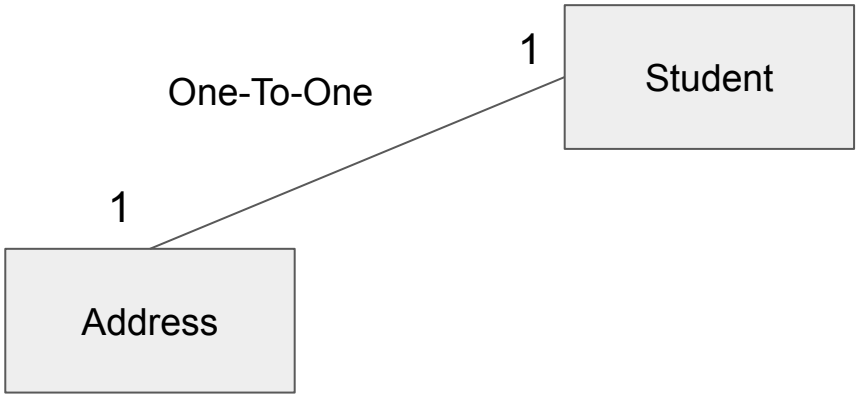


- Student and Address class
- Different options…
  - Unidirectional:
    - From Student to Address:
      - Student had Address attribute
      - Address does not have a Student attribute
    - From Address to Student
      - Address has Student attribute
      - Student does not have an Address attribute
  - Bidirectional
    - Student has Address attribute
    - Address has Student attribute



KdG

One-To-One: we choose…

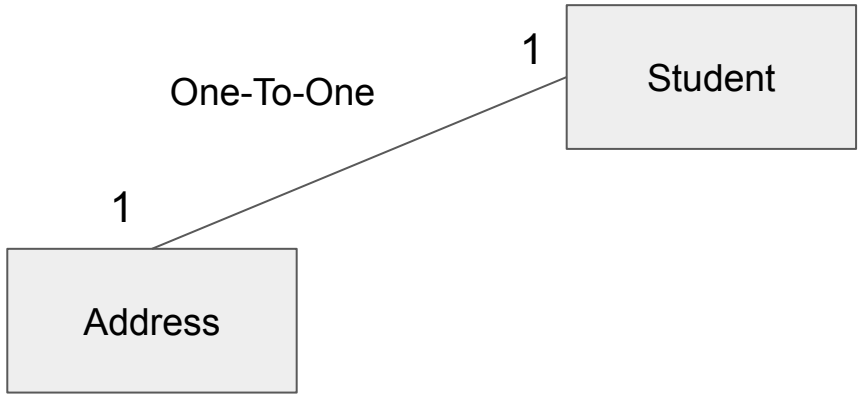


- Database:
  - Table ADDRESS (and of course STUDENTS)
  - ADDRESS has foreign key to STUDENTS
  - ADDRESS does not even have an ID field…

```
create table ADDRESS
(
   STREET      CHARACTER VARYING(100) not null,
   POSTAL_CODE INTEGER                not null,
   CITY        CHARACTER VARYING(100) not null,
   STUDENT_ID  INTEGER                not null,
   constraint FOREIGN_KEY_NAME
       foreign key (STUDENT_ID) references STUDENTS
);
```

KdG

One-To-One: we choose…



- Domain Model:
  - Class Address (and of course class Student)
    - No id field: *Address is not an entity → has no meaning on it's own…*
  - Bidirectional:
    - Student has Address attribute (with setter and getter)
    - Address has Student attribute (with setter and getter)
  - Setting the Address (in Student) will set the Student (in Address)

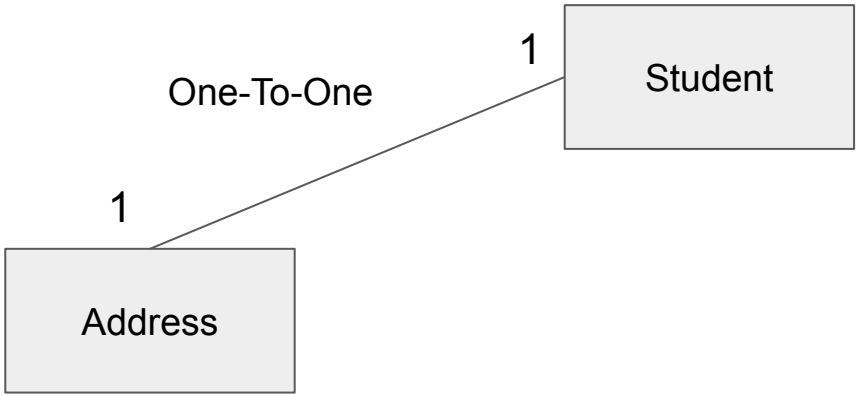
```
public class Student {
   private int id;
   private String name;
   private double lenght;
   private LocalDate birthday;

   //relation: One to One
   private Address address;
```

```
public class Address {
   private String street;
   private int postalCode;
   private String city;

   //relation: One to One
   private Student student;
   …
```

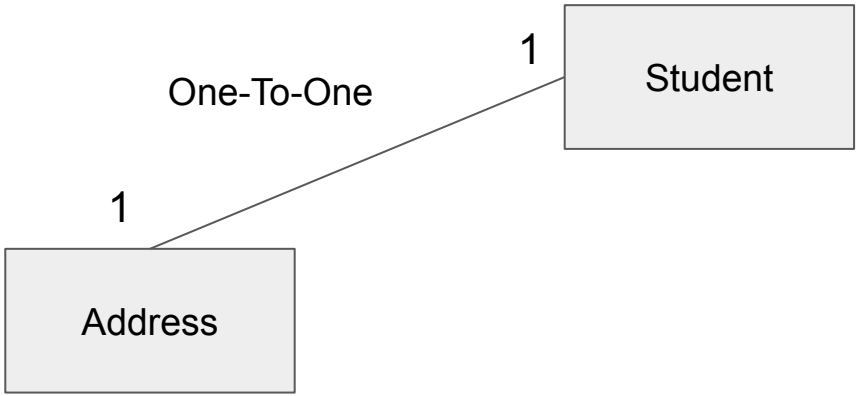
One-To-One: let's do it!



- Start from this code: https://gitlab.com/kdg-ti/programming-3/exercises/relationsdemo
- Database:
  - Create an H2 file database based on the schema.sql and data.sql
  - Add the ADDRESS table with FK to STUDENTS
    - Generate the DDL and add to the schema.sql
    - Add some addresses to ADDRESS, generate SQL INSERTS and add to data.sql
  - Run the application on the h2 memory database
  - Check http://localhost:8080/h2-console

KdG

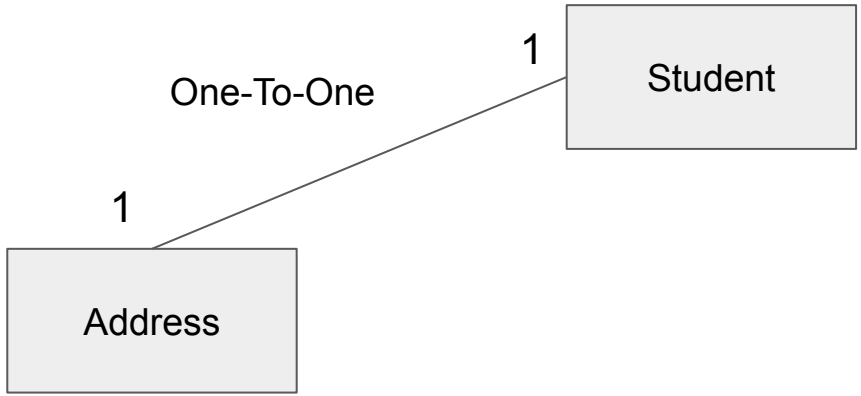
One-To-One: let's do it!



- Domain Model:
  - Create Address class in domain package
    - No id attribute
    - street, postalCode, city attribute
    - Student attribute
    - Getters, setters, toString (do not print the student!)
  - Student class:
    - Add Address attribute to Student class
    - Getter and setter: setter also sets student in Address
    - Add address to the toString

KdG

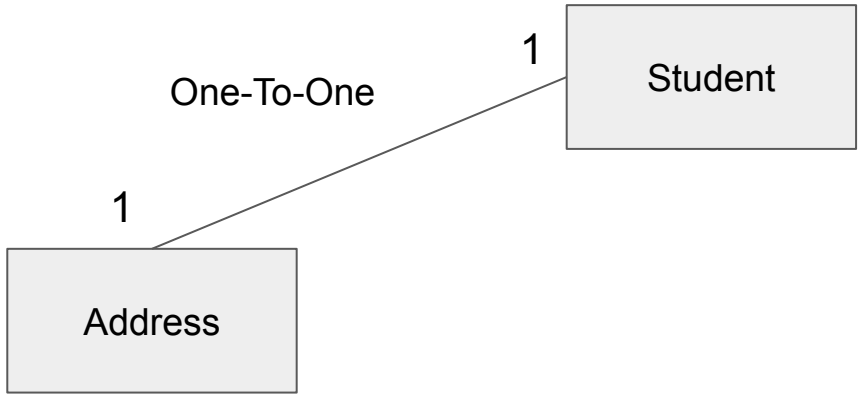
One-To-One: the repository?



- findAll - findById methods
  - When we load the Student, should we load the Address?
    - Eager loading: yes we do
    - Lazy loading: no we don't
  - For One-To-One: eager loading often good idea… → let's do it like this!
- createStudent method
  - If the Student has an Address, we insert it into the ADDRESS table
- updateStudent method
  - If the Student has an Address, we insert or update it into the ADDRESS table
    - You can use a MERGE INTO for that…
- deleteStudent method
  - If we delete a STUDENT we should also delete the ADDRESS

KdG

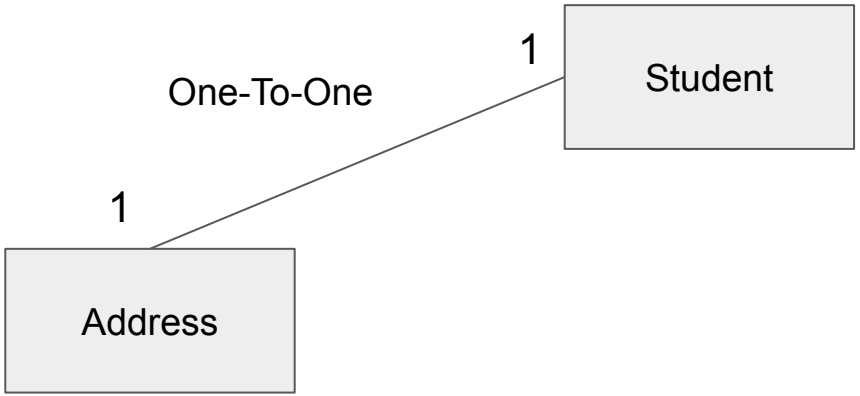
One-To-One: let's do it



- Implement the repository methods:
  - findAll - findById methods → eager loading of Address
  - createStudent method
    - If the Student has an Address, we insert it into the ADDRESS table
  - updateStudent method
    - If the Student has an Address, we insert or update it into the ADDRESS table
      - You can use a MERGE INTO for that…
  - deleteStudent method
    - If we delete a STUDENT we should also delete the ADDRESS

KdG

One-To-One: StudentMenu



- StudentMenu:
  - Test the “list all students”
    - Should show the Address if it exists
  - Test the “delete student”
    - Should also delete the Address from the database
    - Check this in the h2-console
  - Implement the “change address of student”
    - You ask for student id
    - You ask for Street, Postal code and City
    - Create and set the Address of the Student
    - Call updateStudent method on repository

KdG

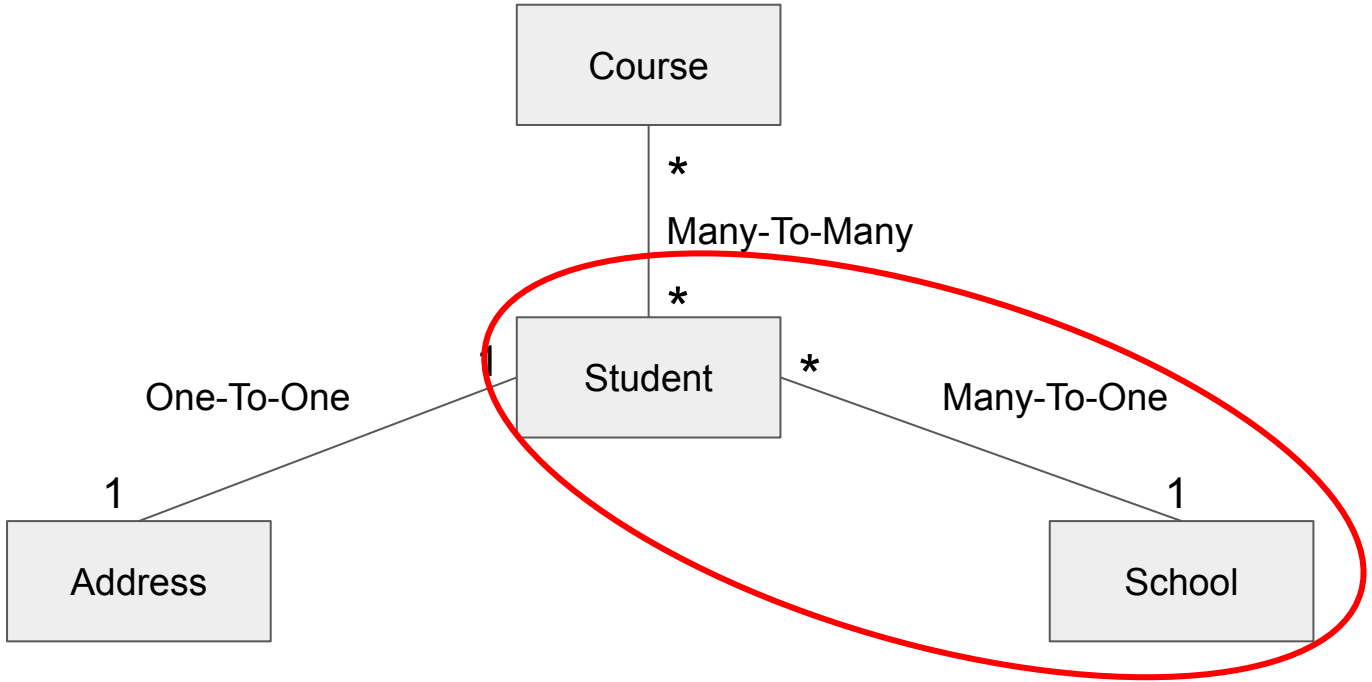
Agenda this week

Project review
Implementing relationships in the repository
One-To-One
Many-To-One
Many-To-Many
Generics: implementing a generic repository → this part will be covered later

KdG

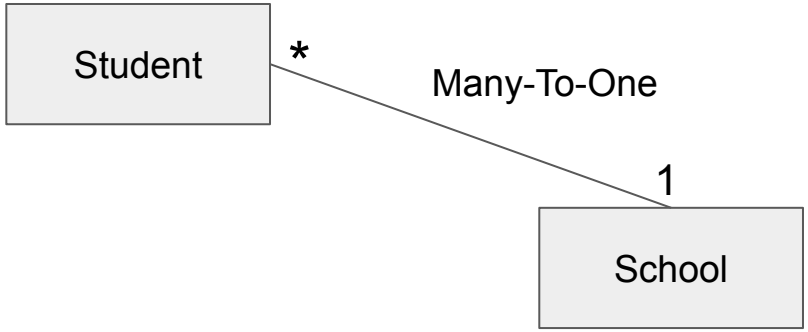
Relationships in the repository

- Student versus School:

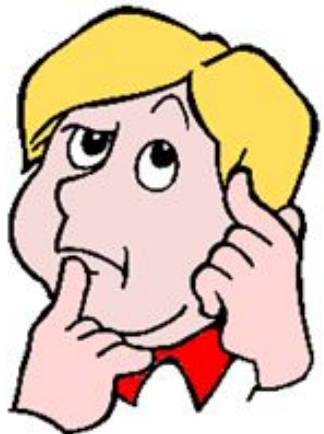


KdG

Many-To-One

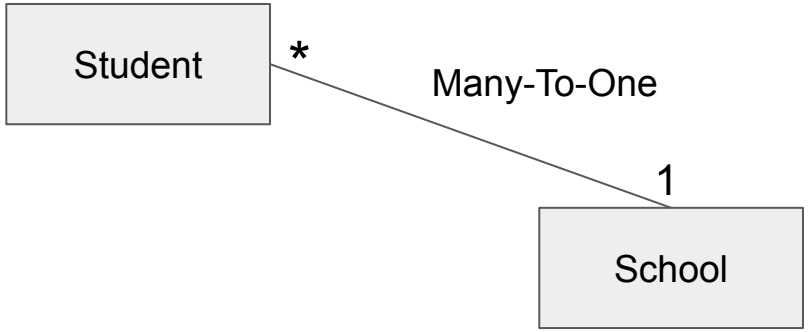


- Student has 1 School
- School has many Students
- A Student always has a School, a Student without a School cannot exist



KdG

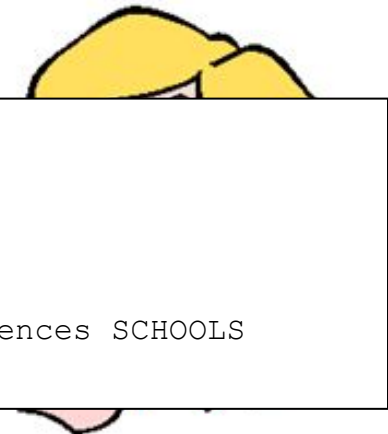
Many-To-One: Database



- Table SCHOOLS
  - School has an ID and a NAME → School is an entity
- Table STUDENTS
  - Has a SCHOOL_ID: foreign key to SCHOOLS

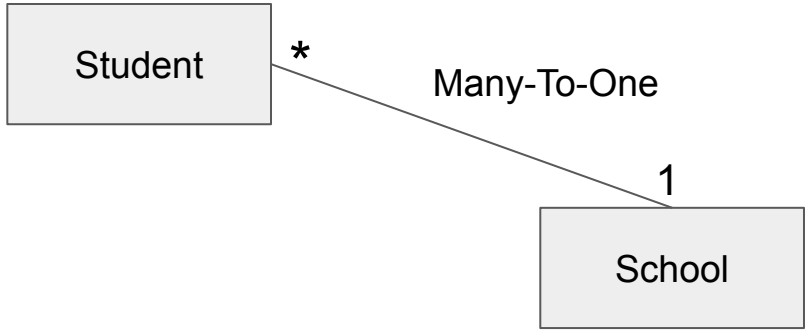
```
create table SCHOOLS
(
   ID   INTEGER auto_increment
       primary key
       unique,
   NAME CHARACTER VARYING(100) not null
);
```

```
create table STUDENTS
(
   …
   SCHOOL ID INTEGER not null,
   constraint FK SCHOOL ID
       foreign key (SCHOOL_ID) references SCHOOLS
);
```

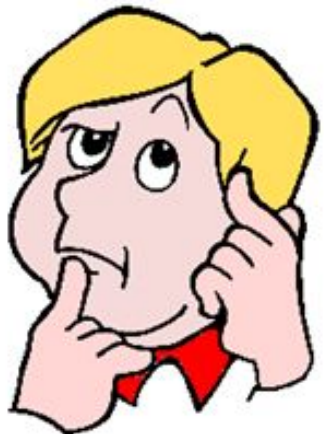


KdG

Many-To-One: in Domain Model

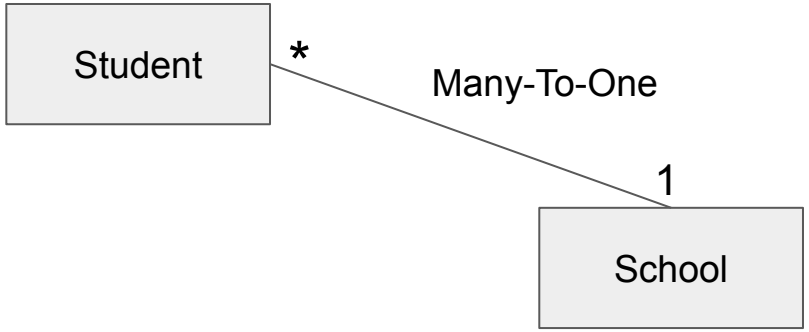


- School class
  - Has a List<Student> attribute and an addStudent method
  - toString method does not show the students
- Student class
  - Has a School attribute with getter and setter
    - Setter adds student to the School using addStudent method
  - toString method shows the School



KdG

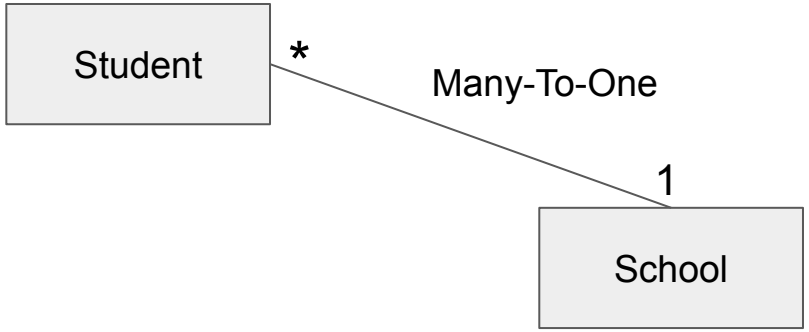
Many-To-One: let's do it!



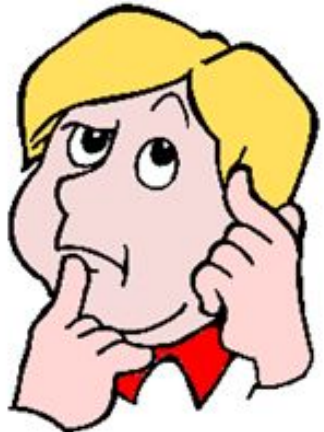
- Database:
  - Add SCHOOLS table to filedatabase
  - Alter the STUDENTS table: add SCHOOL_ID with FK to SCHOOLS
  - Add some data
  - Generate DDL and SQL INSERTS, add to schema.sql and data.sql
- Domain Model
  - Create School class, has id and name. Create constructors, getters, setters, toString
  - Add School to Student, List<Student> to School
  - Update toString of Student

KdG

Many-To-One: repository?

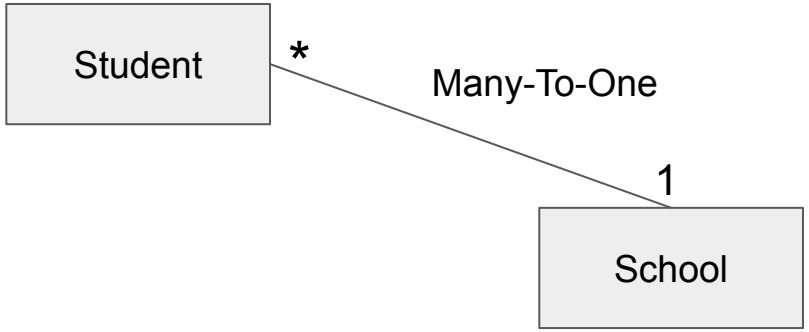


- We add a SchoolRepository
  - Interface - implementation
  - findAll - findById
    - Do we load the students → let's not do that!
  - createSchool - updateSchool
  - deleteSchool:
    - If we delete a school, should we delete the students?
      - Yes: students without a school cannot exist
- In StudentRepository
  - findAll en findById: should we load the School?
    - Many-to-One: ok, let's do it
  - Let's add a findBySchool method
  - createStudent → add the SCHOOL_ID
    - School should already exist, we don't create it here
  - updateStudent → add the SCHOOL_ID



KdG

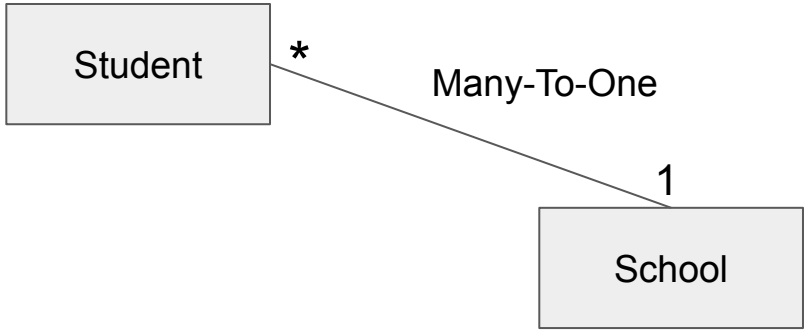
Many-To-One: let's do it



- We add a SchoolRepository
  - Interface - implementation
  - findAll - findById → do not load the students
  - createSchool - updateSchool
  - deleteSchool → delete the students → delete the students addresses
- In StudentRepository
  - findAll en findById: → load the School
  - Let's add a findBySchool method
  - createStudent, updateStudent → add the SCHOOL_ID

KdG

Many-To-One: StudentMenu



- StudentMenu:
  - Test the “list all students”: should show school information
  - addStudent
    - Ask for to (E)xisting or (N)ew school
      - New → ask school name and create school using schoolRepository
      - Existing → ask school id and findById from schoolRepository and setSchool on student object
  - updateStudent: idem!
  - Implement “Change school of student” → to only change school
  - Implement “list all students of school” → use findBySchool method of studentrepo
  - Implement “Delete school” → use schoolRepository

KdG

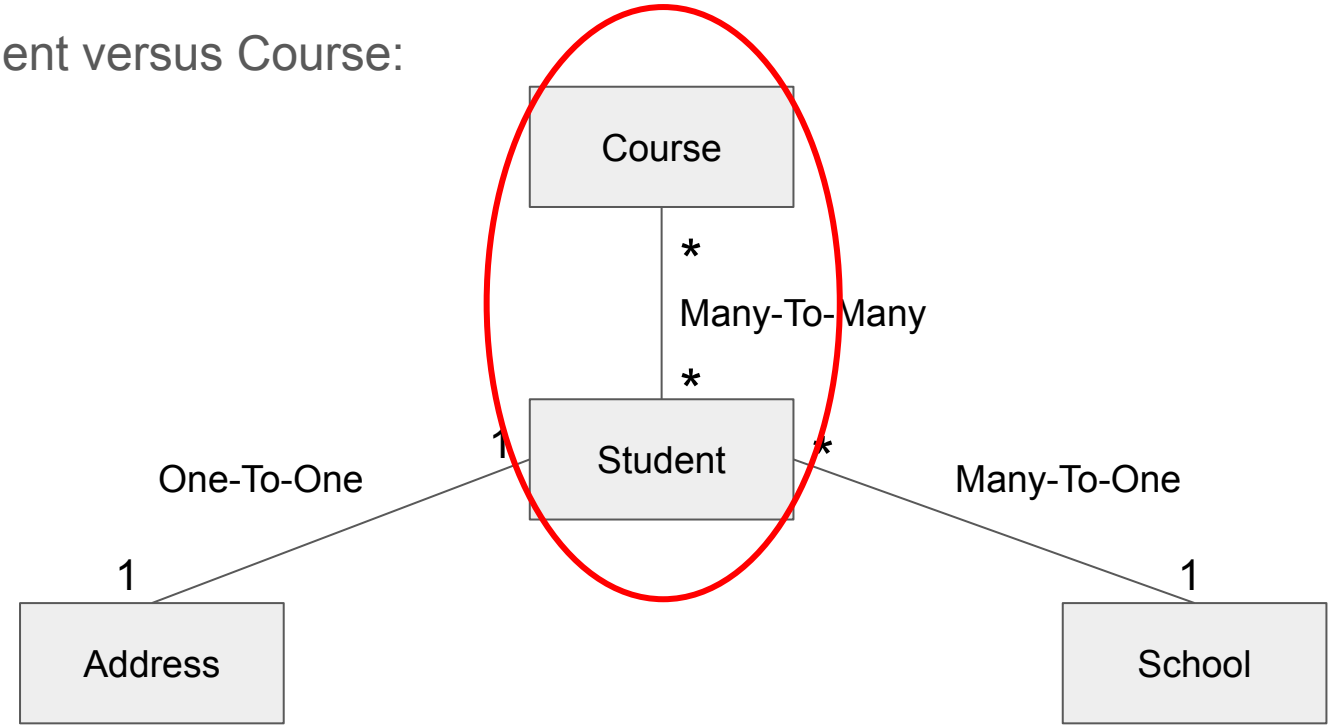
Agenda this week

Project review
Implementing relationships in the repository
One-To-One
Many-To-One
Many-To-Many
Generics: implementing a generic repository → this part will be covered later

KdG

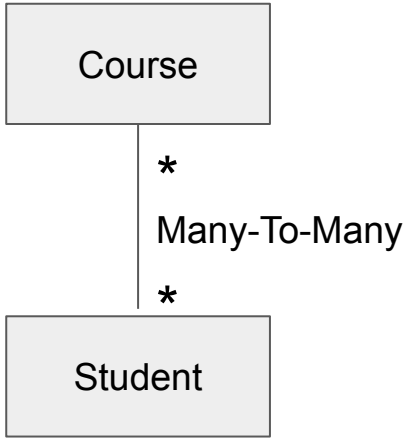
Relationships in the repository

- Student versus Course:

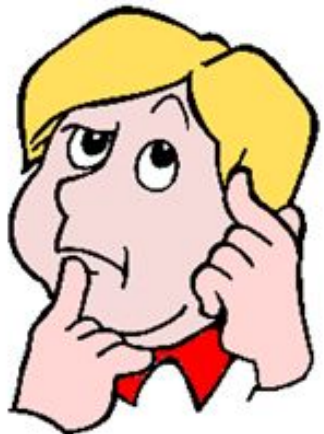


KdG

Many-To-Many

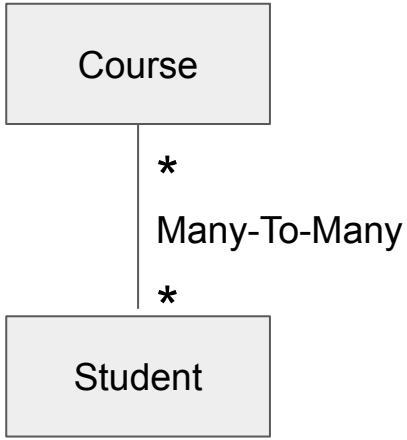


- Student follows many Courses
- Course has many Students



KdG

Many-To-Many: Database

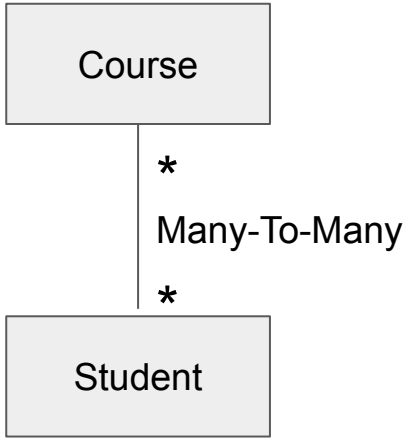


- Table COURSES
  - Has ID and NAME and ACADEMIC_YEAR
- We use a “crosstable” to link courses to students:
  - STUDENTS_COURSES
    - Has STUDENT_ID: foreign key to STUDENTS
    - Has COURSE_ID: foreign key to COURSES

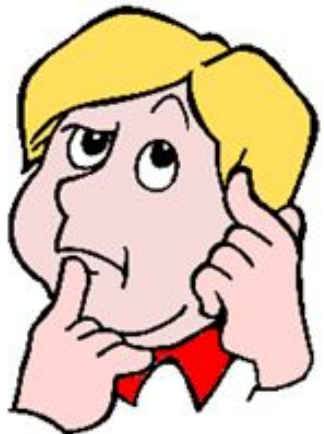
```
create table COURSES
(
   ID            INTEGER auto_increment
       primary key
       unique,
   NAME          CHARACTER VARYING(100) not
null,
   ACADEMIC_YEAR INTEGER                not null
);
```

```
create table STUDENTS_COURSES
(
   STUDENT_ID INTEGER not null,
   COURSE_ID  INTEGER not null,
   constraint FK_COURSE_ID
       foreign key (COURSE_ID) references COURSES,
   constraint FK_STUDENT_ID
       foreign key (STUDENT_ID) references
STUDENTS
);
```

Many-To-Many: in Domain Model

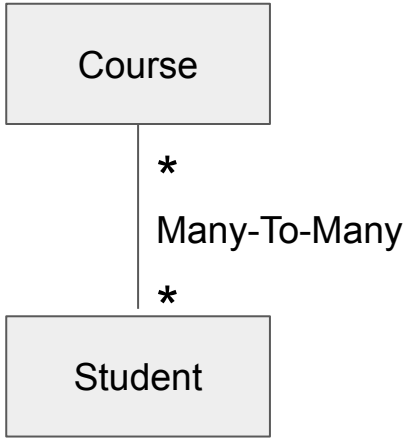


- Course class → is an *entity*
  - Has id, name, academicYear → getters, setters, constructors
  - Has List<Student> students → getter and setter, addStudent
  - toString does not show the students
- Student class:
  - Has List<Course> courses
    - addCourse → also adds student to course
    - Getter - setter
  - We do not add the courses to the toString



KdG

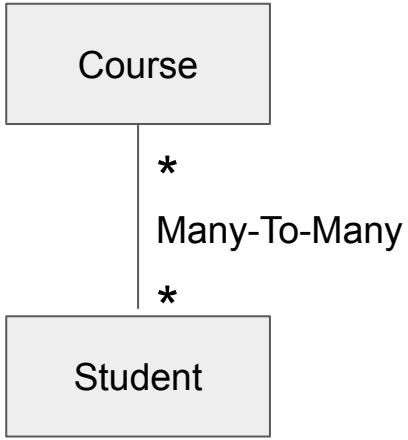
Many-To-Many: let's do it!



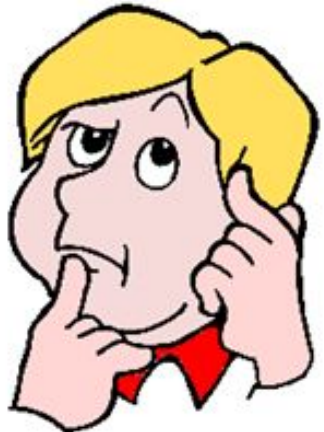
- Database:
  - Add COURSES to filedatabase - add some data
  - Add the crosstable - add some data
  - Generate DDL and SQL INSERTS and update schema.sql and data.sql
- Domain Model
  - Create Course class, see previous slide
  - Update Student class

KdG

Many-To-Many: repository?

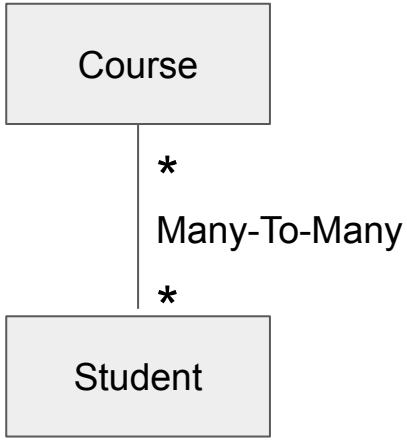


- We need a CourseRepository
  - findAll, findById
    - we do NOT load the Students! (=lazy loading)
    - → that would also load Address and School, for each Student…!
  - createCourse, updateCourse
  - deleteCourse
    - We also need to delete the records in the STUDENTS_COURSES!
- StudentRepository
  - findAll, findById → let's load the courses… (eager loading, good idea?)
  - Add a findByCourse method
  - updateStudent: delete and re-insert the STUDENTS_COURSES?
  - deleteStudent
    - We also need to delete the records in the STUDENTS_COURSES!



KdG

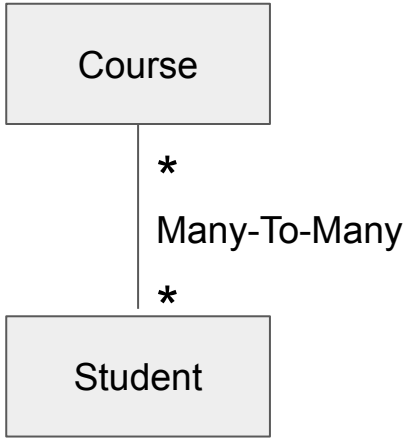
Many-To-Many: let's do it!



- Create a CourseRepository
  - findAll, findById → do NOT load the Students! (=lazy loading)
  - createCourse, updateCourse
  - deleteCourse → also need to delete the records in the STUDENTS_COURSES
- StudentRepository
  - findAll, findById → load the courses…
  - Add a findByCourse method
  - updateStudent: delete and re-insert the STUDENTS_COURSES
  - deleteStudent → delete the records in the STUDENTS_COURSES!

KdG

Many-To-Many: StudentMenu



- StudentMenu:
  - Implement “Add student to course”
  - Implement “List students of course”
  - Implement “List courses of students”
  - Implement “Delete course”
- Test!!!!

KdG

Agenda this week

Project review
Implementing relationships in the repository
One-To-One
Many-To-One
Many-To-Many
Generics: implementing a generic repository → this part will be covered later

KdG

Generics: later!

- In what follows we try to use java generics to implement a “generic” version of the repositories
- **To give you more time to study the relationship exercise we will not cover this topic this week…**

KdG

Generics: examples of use

What we already know:

- Generics with java collection framework (list, set, …)

```
List<String> myList = new ArrayList<>();

myList.add("test");
```

- Generics with Iterator:

```
Iterator<String> it = myList.iterator();

while(it.hasNext()) {

 System.out.println(it.next().toUpperCase());

}
```

No cast needed: compiler knows it's a String

KdG

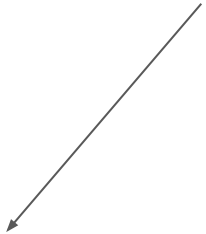
Generics: examples of use

- Generics with Map (key & value):

```
Map<Klant, List<Bestelling>> myMap = new
TreeMap<>();
```

- Generics with Comparable:

```
class Customer implements Comparable<Customer>{
     private int id;
     private String name;
     @Override
     public int compareTo(Customer other) { return
    this.id - other.id;
     }
}
```



KdG

Generic method

```
public class OverloadedMethods {
   public static void printArray(Integer[] array) {
       for(Integer element : array) {
           System.out.println(element);
       }
   }
   public static void printArray(Double[] array) {
       //...
   }
   public static void printArray(String[] array) {
       //...
   }
}
```

Can you replace that with one method?

KdG

Generic method

- We replace by one generic method

```
public static <E> void printArray(E[] inputArray) {
    for (E element : inputArray) {
        System.out.println(element);
    }
}
```

```
public static void main(String args[]) {
    // Create arrays of Integer, Double and String:
    Integer[] intArray = {1, 2, 3, 4, 5};
    Double[] doubleArray = {1.1, 2.2, 3.3, 4.4};
    String[] strArray = {"Just", "Another", "Day"};
    printArray(intArray);
    printArray(doubleArray);
    printArray(strArray);
}
```

Generic method: syntax

```
public static <E> void printArray(E[] inputArray) {
    for (E element : inputArray) {
        System.out.println(element);
    }
}
```

Type parameter section, just before return type. Indicates that we use E as generic type in this method...

- Most common parameter types (naming convention):
  - E - Element (used bij Java Collections)
  - K - Key (used in Map)
  - N - Number
  - T - Type
  - V - Value

KdG

Generic class

This class is generic for a certain type T

```
public class Box<T> {
   private List<T> myList = new ArrayList<>();

   public void add(T t) { myList.add(t); }

   public T get(int i) { return myList.get(i); }

   @Override
   public String toString() {
       StringBuilder sb = new StringBuilder();
       for (T t : myList) {
           sb.append(t + " ");
       }
       return sb.toString();
   }
}
```

The same type T is used for creation of the List, as parameter, as returntype, ...

KdG

Use this generic class

```
public static void main(String[] args) {
   Box<String> stringBox = new Box<>();
   stringBox.add("Hello");
   stringBox.add("World");
   System.out.println(stringBox);

   Box<Integer> integerBox = new Box<>();
   integerBox.add(10);
   integerBox.add(20);
   System.out.println(integerBox);

   Box generalBox = new Box();
   generalBox.add(5.5);
   generalBox.add("O my God!");
   System.out.println(generalBox);
}
```

You can still use it without generics: it uses *raw* types

KdG

Generic interfaces

Example: the Comparable interface:

```
/**
* …
* @param <T> the type of objects that this object may
* be compared to
*
* @author  Josh Bloch
* @see java.util.Comparator
* @since 1.2
*/
public interface Comparable <T> {
   public int compareTo(T o);
}
```

KdG

Bound generics

- ? is the wildcard in generics
  - It means “unknown type”
  - Can be used as type for a parameter, attribute, local variable or return-value
- ? is used in 3 different ways:
  - Upper bound wildcard: `<? extends Number>`
  - Lower bound wildcard: `<? super Integer>`
  - Unbounded wildcard: `<?>`

In the upper/lower bound form you can also use a type parameter so you can reference it in the code that follows: `<N extends Number>`

KdG

Upper bound example

```
public static double sum(List<? extends Number> list) {
    double sum = 0;
    for (Number number : list) {
        sum += number.doubleValue();
    }
    return sum;
}
```

Number is the upper bound class. Watch out: this can not be replaced by List<Number>!

Method of Number class

```
List<Integer> ints = new ArrayList<>();
ints.add(3); ints.add(5); ints.add(10);
double sum = sum(ints);
System.out.println("Sum of ints = " + sum);
```

```
List<Double> doubles = new ArrayList<>();
doubles.add(1.5); doubles.add(3.5); doubles.add(10.0);
sum = sum(doubles);
System.out.println("Sum of doubles = " + sum);
```

KdG

Upper bound example

```
public static double sum(List<? extends Number> list) {
    double sum = 0;
    for (Number number : list) {
        sum += number.doubleValue();
    }
    return sum;
}
```

Number is the upper bound class. Watch out: this can not be replaced by List<Number>!

Method of Number class

- **List<Number>:** can contain mix of Integer, Double, …
- **List<? Extends Number>:** is a List<Integer> or a List<Double> or ...

→ When you are using the *homogeneous* List, you are sure it is possible to *read* a Number.

KdG

Upper bound: the calling code

```
List<Integer> ints = new ArrayList<>();
ints.add(3);ints.add(5);ints.add(10);
double sum = sum(ints);
System.out.println("Sum of ints = " + sum);

List<Double> doubles = new ArrayList<>();
doubles.add(1.5);doubles.add(3.5);doubles.add(10.0);
sum = sum(doubles);
System.out.println("Sum of doubles = " + sum);
```

List of integer, you cannot add a Double

List of Double, you cannot add a Integer

KdG

Lower bound example

```
public static void addIntegers(List <? super Integer> list) {
   list.add(new Integer(50));
   list.add(new Integer(100));
}
```

All supperclasses of Integer, Integer is the lower bound...

List<? super Integer> is not the same as List<Integer>:

- List<Integer>: elements are of type Integer
- List<? super Integer>: it can be a List<Integer> or List<Number> or List<Object>

→ When you are using the *homogeneous* List, you are sure it is possible to *add* an Integer.

KdG

Unbound example

```
public static void printData(List <?> list){
   for(Object obj : list){
       System.out.print(obj + "::");
   }
   System.out.println();
}
```

Same as <? extends Object>

```
List<Integer> ints = new ArrayList<>();
ints.add(3); ints.add(5); ints.add(10);
printData(ints);

List<String> strings = new ArrayList<>();
strings.add("Just"); strings.add("Another"); strings.add("Day");
printData(strings);

List<Object> objects = new ArrayList<>();
objects.add(3.14); objects.add("Hello"); objects.add(new Random());
printData(objects);
```

KdG

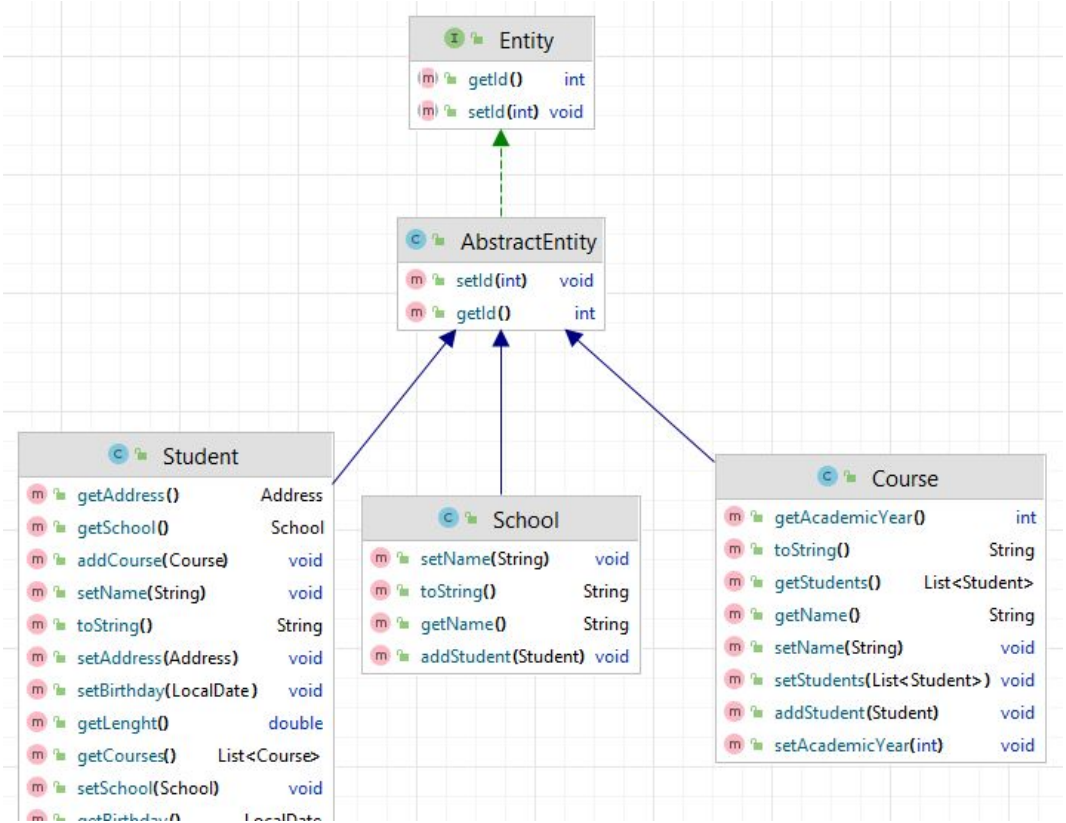
Exercise: can we create a *generic* repository?

- One repository that can be used for any entity!
- Let's call it an EntityRepository
- Preparation:
  - Create an interface Entity that defines an entity: it's something with an id...

```
public interface Entity {
    int getId();
    Void setId(int id);
}
```

KdG

All entities implement this interface



KdG

Generic repository interface

```
public interface EntityRepository<T extends Entity> {
    List<T> findAll();
    T findById(int id);
    T create(T entity);
    void update(T entity);
    void delete(int id);
}
```

KdG

Creating the different interfaces is easy!

```
public interface SchoolRepository extends EntityRepository<School> {
}
```

```
public interface CourseRepository extends EntityRepository<School> {
}
```

```
public interface StudentRepository extends EntityRepository<School> {
    List<Student> findBySchool(int schoolid);
    List<Student> findByCourse(int courseId);
}
```

StudentRepository has some specific extra methods…

KdG

Generic JDBC Implementation of the generic interface

```
public class JDBCRepository<T extends Entity> implements EntityRepository<T> {
    …
    @Override
    public List<T> findAll() {
        …
    }
    @Override
    public T findById(int id) {
        …
    }
    @Override
    public T create(T entity) {
        …
    }
    @Override
    public void update(T entity) {
        …
    }
    @Override
    public void delete(int id) {
        …
    }
```

Check the JDBC repositories you have created before: what is the common code?

Generic JDBC Implementation of the generic interface

- Attributes:

```
protected JdbcTemplate jdbcTemplate;
protected SimpleJdbcInsert inserter;
```

- Constructor:

```
public JDBCRepository(JdbcTemplate jdbcTemplate) {
   this.jdbcTemplate = jdbcTemplate;
   this.inserter = new SimpleJdbcInsert(jdbcTemplate)
           .withTableName(getTableName())
           .usingGeneratedKeyColumns("ID");
}
```

We suppose the primary key is always in a column called ID...

KdG

Generic JDBC Implementation of the generic interface

- findAll implementation:

```
@Override
public List<T> findAll() {
   return jdbcTemplate.query("SELECT * FROM " + getTableName(), this::mapEntityRow);
}
```

The name of the Table is specific for each repository: let's add an abstract method to return this String

The mapping of the columns of the database to the fields of the entity is also repository-specific. We add an abstract method for that also!

KdG

Make the implementation abstract:

```
public abstract class JDBCRepository<T extends Entity> implements EntityRepository<T> {
```

- And we add the 2 abstract methods:

```
abstract String getTableName();
abstract T mapEntityRow(ResultSet rs, int rowid) throws SQLException;
```

KdG

The concrete implementation: eg JDBCSchoolRepository

```
@Repository
public class JDBCSchoolRepository extends JDBCRepository<School> implements
SchoolRepository {
```

- Implementation of the abstract methods:

```
@Override
String getTableName() {
   return "SCHOOLS";
}

@Override
School mapEntityRow(ResultSet rs, int rowid) throws SQLException {
   return new School(rs.getInt("ID"),
            rs.getString("NAME"));
}
```

KdG

Now try to implement the generic JDBCRepository!

- findById
  - Suppose the primary key is always in a column called ID
- delete
- create
  - This is harder: we need the parameters HashMap
    - Add another abstract method that returns this..
- update
  - This is a challenge: can you use the parameters HashMap to create the UPDATE string?
    - The keys are the names of the columns
    - The values are the values to pass on to the jdbcTemplate.update

KdG

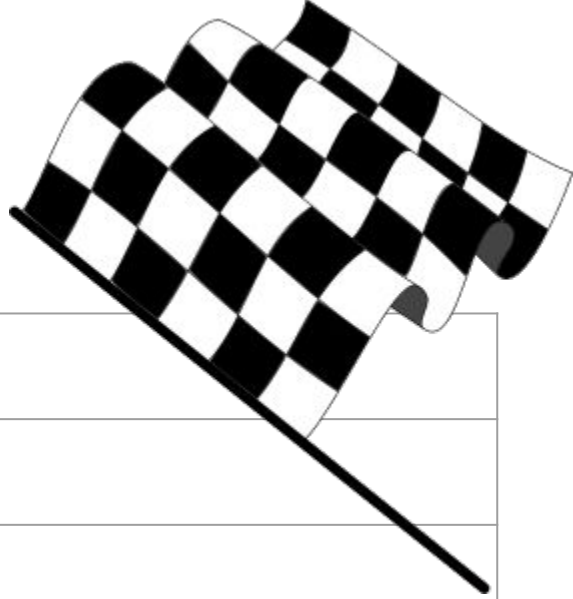
Now create the different JDBC repositories

- JDBCSchoolRepository extends JDBCRepository<School> implements SchoolRepository
- JDBCCourseRepository extends JDBCRepository<Course> implements CourseRepository
- JDBCStudentRepository extends JDBCRepository<Student> implements StudentRepository

→ If those implementations need specific code, you override the methods of the superclass

KdG

Agenda this week



Project review
Implementing relationships in the repository
One-To-One
Many-To-One
Many-To-Many
Generics: implementing a generic repository → this part will be covered later

KdG

Project

- Add the relationships to your project
  - Add them in the database, use cross table for many-to-many
  - Adjust/add repository classes where necessary
    - You use the jdbctemplate implementation
  - If you click on an entity in the table to see its details, you also see its related entities (eg: click on Book shows bookdetails and list of Authors, click on Author shows list of Books)
- Add the possibility to delete the main entities, for example by adding a delete icon at the end of each row of the tables

KdG