Iben Andersen
cph-ia62@cphbusiness.dk

Exercises JPA Relations

Collections of basic types

1) Create an Entity class Customer

```
@Entity
public class Customer2 implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstname;
    private String lastname;

    public Customer2() {
    }

    public Customer2(String firstname, String lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
```

2) Provide the Customer class with a list of hobbies: private List<String> hobbies = new ArrayList(); Add the methods to the class: addHobby(String s) and String getHobbies().

```
private List<String> hobbies = new ArrayList();

public void addHobby(String hobby) {
    hobbies.add(hobby);
}

public String getHobbies() {
    return String.join(",", hobbies);
}
```

Iben Andersen
cph-ia62@cphbusiness.dk

3) Add a class, Tester.java, to test drive (manually, not with JUnit) the Customer class and create and persist a few customers with some hobbies.

```
public class Tester {

    public static void main(String[] args) {
        Customer2 cust = new Customer2("John","Smith");
        cust.addHobby("Tennis");
        cust.addHobby("Beer");

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("pu");
        EntityManager em = emf.createEntityManager();
        try{
            em.getTransaction().begin();
            em.persist(cust);
            em.getTransaction().commit();
        }finally{
            em.close();
        }
        em = emf.createEntityManager();
        Customer2 found = em.find(Customer2.class, cust.getId());
        System.out.println("Hobbies --->" + found.getHobbies());
    }

}
```

Test and verify how the list is stored by the Customer table.

If not, add the following annotation to the hobbies List @ElementCollection

```
@ElementCollection
private List<String> hobbies = new ArrayList();
```

Regenerate (run the project) tables and observe the result.

#	ID	FIRSTNAME	LASTNAME
1	1	John	Smith

#	Customer2_ID	HOBBIES
1	1	Tennis
2	1	Beer

Iben Andersen
cph-ia62@cphbusiness.dk

JPA Entity Mappings

Relationship Mapping

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

For this exercise you need two Entity classes as sketched below:

Customer, with the fields: id (Integer), firstname(String), lastname(String)

Address, with the fields: id (Integer), street (String), city (String)

For both classes, use GenerationType.IDENTITY.

```
@Entity
public class Customer implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstname;
    private String lastname;
```

```
@Entity
public class Address implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String street;
    private String city;
```

Add a Tester-class, add and execute this, to verify that we can create the matching classes.

Iben Andersen
cph-ia62@cphbusiness.dk

1) One to One – Unidirectional

Provide the Customer with an Address field private Address address;

```
@OneToOne
private Address address;
```

Make sure you understand how an OO-language implements OneToOne relations and how a relational database does the same.

OO-language implements OneToOne relations when the source object by having an attribute that references another target object.

In a relational database relations are defined through foreign keys.

2) One to One – Bidirectional

Remove the @OneToOne annotation and create a bidirectional one to one relationship

Make sure you understand what is meant by bidirectional.

A bidirectional relationship is when a source object references a target object and the target object also has a relationship to the source object.

Go to the Address Class. Investigate and <u>understand</u> the generated code.

```
@OneToOne(mappedBy = "address")
private Customer customer;
```

In the Address Class a Customer field has been added with a mappedBy function.

Run the project and investigate the generated tables (the foreign key). Is there any difference compared to the previous exercise.

There is no difference because the relations are still defined through foreign keys as in the previous exercise. The tables relate to each other with foreign key ID's.

Iben Andersen
cph-ia62@cphbusiness.dk

3) OneToMany (unidirectional)

Generate a OneToMany relationship and change your Address field so change it into

private List<Address> addresses = new ArrayList();

Now, use the wizard to generate a OneToMany Unidirectional relationship.

Nothing was changed in Address because the relationship is unidirectional (one-way).

When running the project a join table has been generated.

Use @JoinColumn annotation to implement the relation using a foreign key

```
@OneToMany
@JoinColumn
private List<Address> addresses = new ArrayList();
```

Create a "test" method and insert a number of Customers with Addresses into the tables

4) OneToMany (bidirectional)

Use the wizard to generate a OneToMany Bidirectional relationship.

Observe the generated code, especially where we find the mappedBy value.

Customer Class

```
@OneToMany(mappedBy = "customer")
private List<Address> addresses = new ArrayList();
```

Address Class

```
@ManyToOne
private Customer customer;
```

The mappedBy value is now found in the Customer Class.

Run the project and investigate the generated tables (the foreign key).

The relations are still defined through foreign keys where one Customer can have many Addresses, shown by ID's in the Address table, where the ID is connected to an ID in the Customer table.

Create a "test" method and insert a number of Customers with Addresses into the tables, using JPA. Which extra step is required for this strategy compared to OneToMany unidirectional?

The extra step is that you have to set the Address to the Customer and set the Customer to the Address. You have to make relations both ways now.

Iben Andersen
cph-ia62@cphbusiness.dk

5) Many To Many (bidirectional)

How can we implement ManyToMany relationships in an OO-language like Java?

Adding a field with a List of Addresses in the Customer Class and adding a field with a List of Customers in the Address Class.

How can we implement ManyToMany relationships in a Relational Database?

Adding a join table with both Address ID and Customer ID as foreign keys to the Address table and to the Customer table.

a) Right-click the addresses list and select create bidirectional Many to Many Relationship

Customer Class

```
@ManyToMany
private List<Address> addresses = new ArrayList();
```

Address Class

```
@ManyToMany(mappedBy = "addresses")
private List<Customer> customers;
```

Run the project and investigate the generated tables. Explain ALL generated tables.

The generated tables are: CUSTOMER, ADDRESS and CUS_ADD. The CUS_ADD table is a join table with foreign keys to the Customer table and to the Address table.

b) Create a "facade" class CustomerFacade and add the methods:

Facade is implemented in the NetBeans project jpa-relations in the Source Package facade CustomerFacade.java.