Exercises JPA Relations

As you know from previous semesters, Java allows us to create collections of simple types as in these examples:

```
List<String> names= new ArrayList()
List<Integer> numbers = new ArrayList()
```

Java also allows us to make collections of complex user-defined types as in these examples, assuming we somewhere have declared the classes `Customer` and `Actor`

```
List<Customer> customers = new ArrayList();
List<Actor> actor = new ArrayList()
```

Finally, Java also allows us to relate classes using Inheritance. You should know that JPA handles this as well. This however is not a part of the 3. semester syllabus, but we have included an extra (separate) exercise for (very) red students if you like to try this as well

These exercises will demonstrate how to map collections like these to tables in a database using JPA

Important: *Do Not use any of our start code examples for the following exercises. Just create a plain maven project and remember to include the MySQL connector and grab your `EntityManagerFactory` as you did in flow1, week-1.*

Collections of basic types

1) In your NetBeans project, create an Entity class Customer, with a *firstName* and *lastName* property, sufficient constructors and getters/setters. Use GenerationType.Identity strategy for the ID.

2) Provide the Customer class with a list of hobbies as sketched below:

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

Add the following methods to the class: `addHobby(String s)` and `String getHobbies()` (returns a comma-separated list with all hobbies)

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.

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

Do you like what you see?

If not, add the following annotation to the hobbies List (do it anyway ;-)

@ElementCollection()

Regenerate (run the project) tables and observe the result. we assume you agree that this is so much better ;-)

🟥 4) What if you don’t like the names of the generated table and its column names?

Remember, by default, JPA uses *convenience over configuration* and supplies default values for everything you don’t supply. But YOU NOT JPA should be in control, to make a perfect database. Use info on this link and add the necessary annotations so the name of the new class is hobbies, the column with the foreign key is named Customer_ID and the column with the actual hobby is named HOBBY.

🟨 🟥 Maps of Basic Types

Add a map to your Customer class as sketched below:

```
private Map<String,String> phones = new HashMap();
```

Add the following methods to the class:

```
addPhone(String phoneNo, String description){..}

getPhoneDescription(String phoneNo){..}
```

Add a few phone numbers to your customer, in the Tester class, and execute (which should regenerate the tables).

Bloooob, do you like what you see?

If not, add the following annotations to the map:

```
@ElementCollection(fetch = FetchType.LAZY)
@MapKeyColumn(name = "PHONE")
@Column(name="Description")
```

Execute and observe the generated columns and values. Make sure you understand the purpose of each of the annotations

JPA Entity Mappings

These exercises are meant to assist you in learning important ORM-topics. The task is not to complete the exercises as fast as possible, ***but to LEARN important concepts****. Don't skip parts like: "Make sure you understand ..", "explain why .." etc. These parts match typical questions we will ask during the final examination.*

Relationship Mapping

Object-oriented programming is to a large degree about creating classes that relate to each other. If a class cannot solve a problem itself, it can delegate the job to another class.

Relations between classes have **cardinality** and can be either **bi- or *uni-directional***. Make sure you understand these terms.

There are 8 different combinations of Cardinality and Direction, and they can all be implemented in JPA, and if you like, with some NetBeans-assistance.

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

In the following, we will test four of these combinations. Make sure you have "read" the literature related to relation-mapping before you start with these exercises.

Getting Started)

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.`

Create a new plain java maven project (or continue with the project from part-1), and use the NetBeans-wizard to add the two entity classes + relevant getters/setters/constructors (don't forget the zero-arg constructor).

Add a Tester-class, similar to day-1, add and execute this line, to verify that we can create the matching classes:

```
Persistence.generateSchema("NAME_OF YOUR_PU", null);
```

Important: The following exercises are just as much a recap on your second-semester knowledge related to relational mappings. DON't skip the steps, where you are requested to compare Entity Classes to the generated tables. This is to a large degree what we will discuss during the examination when we are focusing on JPA.

1) One to One – Unidirectional

Provide the Customer with an Address field:

```
private Address address;
```

NetBeans hint: Add the cursor on the field and press ALT + ENTER → Select Create *unidirectional one to one relationship*

Make sure you understand everything that changed in the `Customer` and (if any) in the `Address` class.

Regenerate the schema and investigate the generated tables. Observe the location of the *foreign key*

Important: Before you continue, make sure you understand (100% exam relevant) how an OO-language implements OneToOne relations and how a relational database does the same.

2) One to One – Bidirectional

Remove the @OneToOne annotation and create a bidirectional *one to one* relationship *(*with NetBeans use the same hint as above, and select the relevant option).

Make sure you understand what is meant by bidirectional before you continue (how would you show bidirectional using UML, and how is it implemented in your two entity classes)

A bidirectional relationship will obviously require a reference in the Address class, pointing back to Customer. Provide a name when requested by the wizard (`customer`) and select the default for the owning side[1].

- Go to the Address class. Investigate and <u>understand</u> the generated code.
- Run the project and investigate the generated tables (the foreign key). Is there any difference compared to the previous exercise. <u>If not explain why</u>.

[1] *The owning side of the relation is the side of the relation that* owns *the foreign key in the database*

3) OneToMany (unidirectional)

Remove the generated code in both classes and use the wizard, one more time. This time to generate a *OneToMany relationship*. You obviously can't do that with your current Address field so change it into:

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

Now, a Customer can have several addresses. If you feel the opposite makes more sense; an address can have more Customers (i.e two customers are married, and live together), just do that instead, this is a "business decision".

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

- Observe the generated code.
- Run the project and investigate the generated tables. Make sure to press Refresh so see all tables.
- How many tables were generated? Explain the purpose of each of the tables.
- If you (as us) don't like the number of generated tables generated by this strategy, you can use the `@JoinColumn` annotation to implement the relation using a foreign key. Do this, but before you test, delete ALL generated tables in the database
- Create a "test" method and insert a number of Customers with Addresses into the tables, using JPA.

4) OneToMany (bidirectional)

Remove the generated code in both classes and comment out your test code.

Use the wizard to generate a OneToMany *Bidirectional relationship*. Make sure you understand all the suggestions given by the wizard before you accept.

- Observe the generated code, especially where we find the *mappedBy* value. **Explain**.
- Run the project and investigate the generated tables (the foreign key).
- 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?

Again, before you continue, make sure you can explain/answer the questions above, and generally explain the generated tables and how they map to the code.

5) Many To Many (bidirectional)

Finally, let's implement a ManyToMany relationship between Customer and Address, That is: a customer can have many addresses, and an address can "have" many Customers.

IMPORTANT: Before you do this, refresh your knowledge from 1-2 semester and answer the following questions.

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

a) Remove the generated code in both classes

Right-click the addresses list and select create *bidirectional Many to Many Relationship* (observe; both sides can be the owning side)

- Observe the generated code and make sure you understand every line generated in BOTH classes.
- Run the project and investigate the generated tables. Explain ALL generated tables.
- Create a "test" method and insert a number of Customers and Addresses. Make sure to test both the scenario where a customer can have more than one address and an Address can belong to more than one customer.

b) Create a "façade" class *CustomerFacade* and add the following methods:

```
Customer getCustomer(int id);
List<Customer> getCustomers(); //(Check out the hints below)
Customer addCustomer(Customer cust);
Customer deleteCustomer(int id);
Customer editCustomer(Customer cust);
```

If not already done, provide the Customer Class with the following methods:

- `List<Address> getAddresses();`
- `Void addAddress(Address address);`

Provide the Address class with the similar methods (for Customers)

Hints

Open the JPQL readings for tomorrow. Don't read it, just find the SELECT queries near the top. You should be able to modify one for this use case.

In order to persist both a Customer and his Addresses you can:

- Use the cascade property on the @ManyToMany annotation, or:
- Persist the customer, and persist the addresses in the Customers address list.