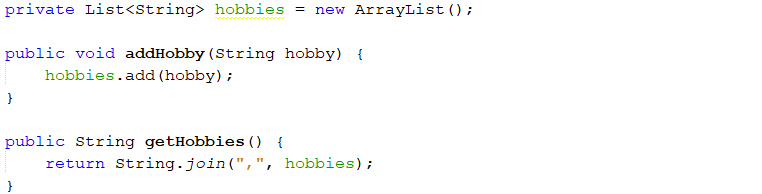
Exercises JPA Relations

# Collections of basic types

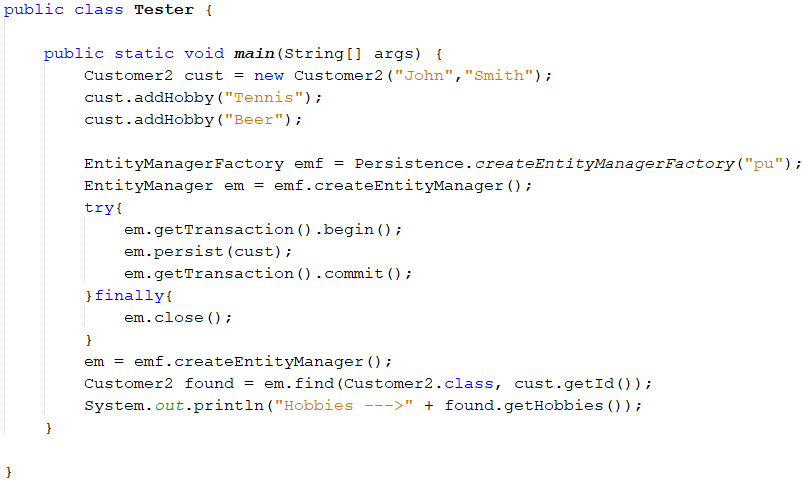
***1) Create an Entity class Customer***



***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().***



***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.***



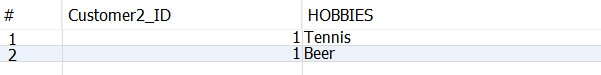
***Test and verify how the list is stored by the Customer table.***

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



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





## 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

## Relationship Mapping

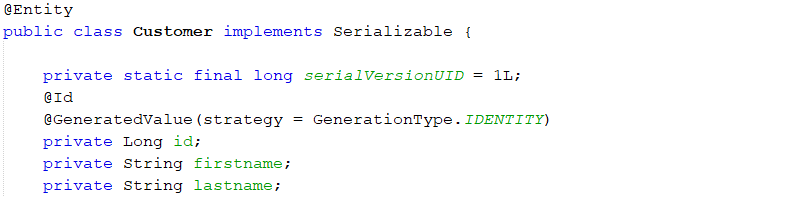
|  |  |
| --- | --- |
| **Cardinality** | **Direction** |
| One-to-one | Unidirectional |
| One-to-one | Bidirectional |
| One-to-many | Unidirectional |
| Many-to-one/one-to-many | Bidirectional |
| Many-to-one | Unidirectional |
| Many-to-many | Unidirectional |
| Many-to-many | Bidirectional |

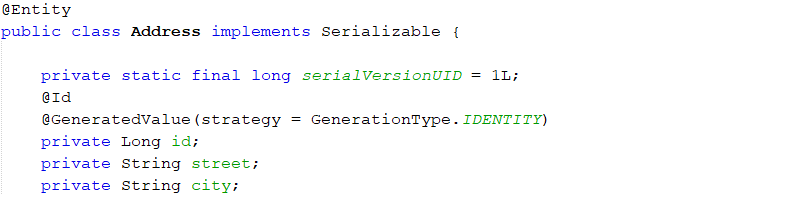
***For this exercise you need two Entity classes as sketched below:***

***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.***





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

## 1) One to One – Unidirectional

***Provide the Customer with an Address field 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 understand the generated code.***



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.

## 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***



***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

Address Class

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.

## 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 

Address Class 

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

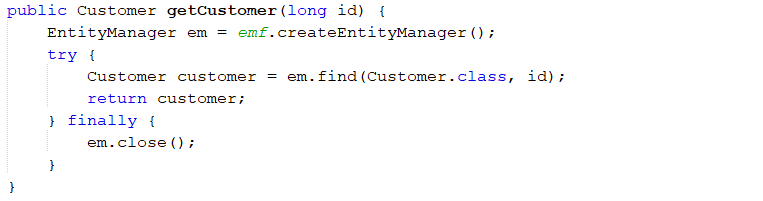
The generated tables are: CUSTOMER, ADDRESS and CUS\_ADD. The CUS\_ADD table is a join table with foreign keys to the Customer table and to the Address table.

***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.***

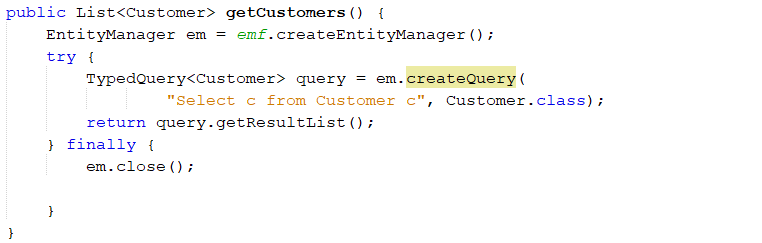
…

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

***public Customer getCustomer(int id);***



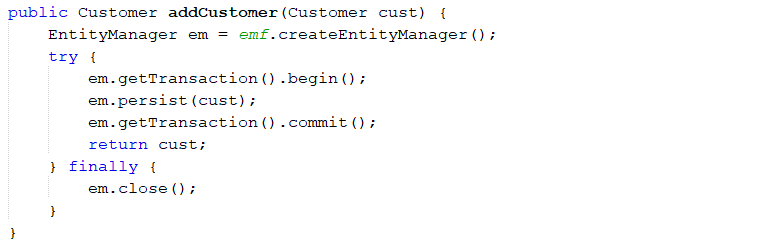
***List<Customer> getCustomers();***

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.

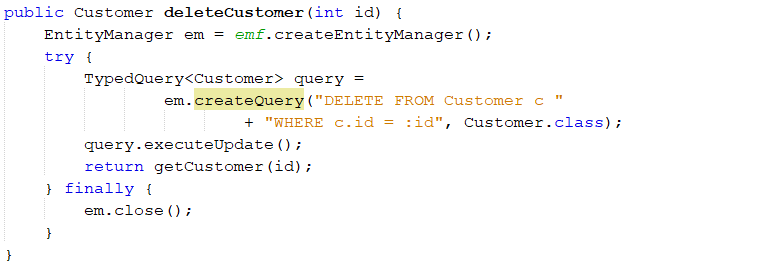
In order to persist both a Customer and his Addresses you can:

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

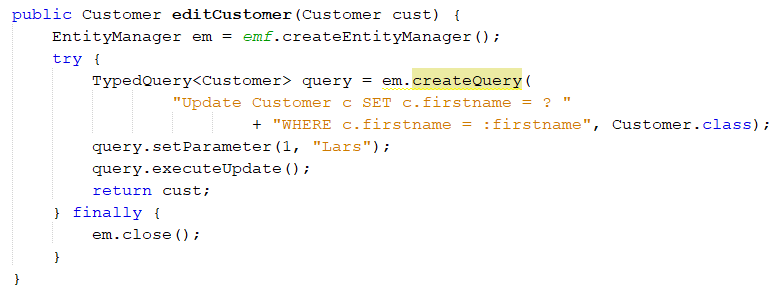
***Customer addCustomer(Customer cust);***



***Customer deleteCustomer(int id);***



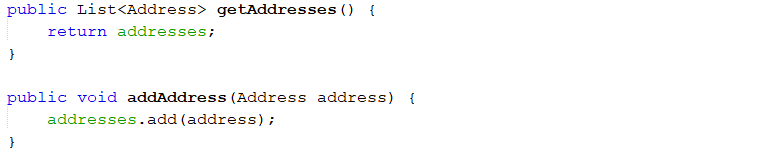
***Customer editCustomer(Customer cust);***



***Provide the Customer Class with the following methods***

***List<Address> getAddresses();***

***Void addAddress(Address address);***

Customer Class 

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

Address Class 