4. Entity Relationships

Contents

[1. Introducing Entity Relationships 1](#_Toc13954)

[2. Demo: Define One-to-many and Many-to-one Relationships 2](#_Toc11870)

[3. Demo: Define Many-to-many Relationships 5](#_Toc6400)

[4. Annotations for Relationship Definition 9](#_Toc25001)

[5. Demo: Join Tables on One Column 10](#_Toc12405)

[6. Demo: Join Tables on Multiple Columns 12](#_Toc17005)

[7. Embeddable Classes 14](#_Toc26462)

[8. Demo: Embedding Classes in Entities 15](#_Toc9775)

[9. Demo: Embedding Collections of Classes in Entities 18](#_Toc26179)

[10. Demo: Embedding Maps of Classes in Entities 21](#_Toc24606)

[11. Queries and Relationship Direction 24](#_Toc30007)

# 1. Introducing Entity Relationships

=>slides: Pg. 1

Hello. Welcome to this module about entity relationships. It is time now to take a detailed look at how entities are connected to each other. We'll examine a set of alternatives that we can choose from in order to make our design and our work efficient. So, let's move forward.

=>slides: Pg. 2

This module will examine the types of relationships, the directions of the relationships, annotations used for relationships, embedded classes, and collections of embedded classes.

=>slides: Pg. 3

JPA supports the following types of relationships for the entities: one‑to‑many, many‑to‑one, many‑to‑many, and one‑to‑one.

=>slides: Pg. 4

Relationships can be unidirectional. Such a relationship has only one owning side. Bidirectional. Such a relationship has an owning side and an inverse side.

=>slides: Pg. 5

Any relationship, no matter if it is unidirectional or bidirectional, has an owning side. This owning side drives the updates to relationship in a database. This means that this is the side that initiates the change of the relationship. In a one‑to‑many and many‑to‑one relationships, the many part of the relationship is always the owning side. It contains the physical reference as a foreign key. If there is a need to change the relationship between two records, the request will come from this side. The inverse side of a bidirectional relationship must refer to it's owning side.

=>slides: Pg. 6

This is an example of a one‑to‑many relationship. As we were saying, the owning side is on the many part. The passenger, John Smith, with ID 1 has multiple tickets, and the ID 1 is kept on the many side as a foreign key.

=>slides: Pg. 7

If we need to change the relationship so that the tickets no longer belonged to John Smith with ID 1, but to Mike Johnson with ID 2, we change the foreign key on the owning side. Now the passenger Mike Johnson with ID 2 is the owner of the tickets, and the ID 2 is kept on the many side as a foreign key. It is the many side as owning side that changed the relationship.

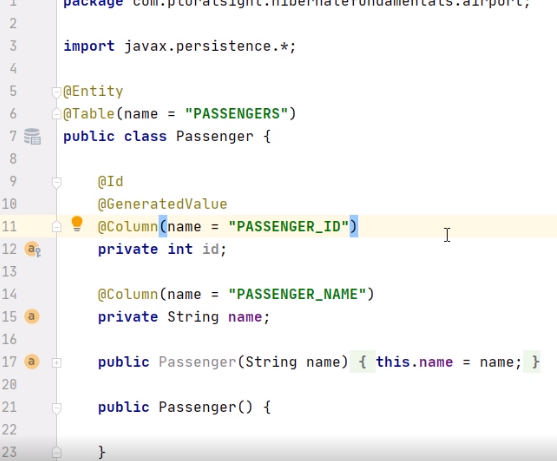
=>slides: Pg. 8

The annotations that are used to define entities relationships are OneToMany defines a one‑to‑many relationship, ManyToOne defines a many‑to‑one relationship, OneToOne defines a one‑to‑one relationship, and ManyToMany defines a many‑to‑many relationship.

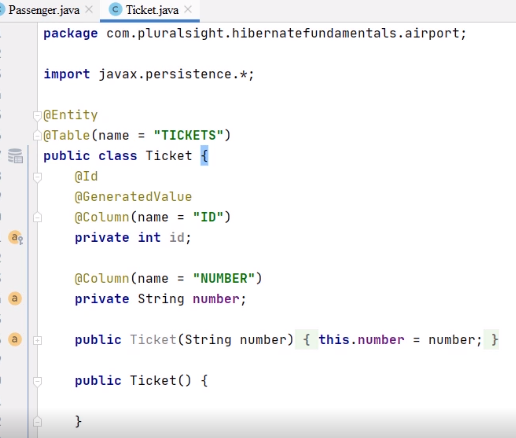
# Demo: Define One-to-many and Many-to-one Relationships

=>slides: Pg. 9

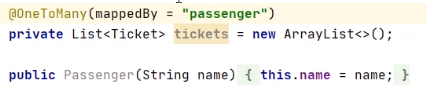




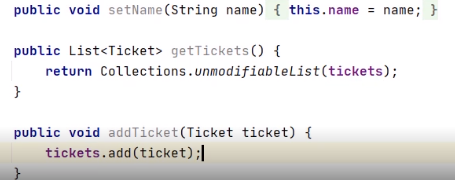
Let's move to the practical demonstration. We'll see how to put in practice defining one‑to‑many relationships and defining many‑to‑one relationships. We start with a Maven project for which the pom.xml and persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence unit and the database name. The project contains an entity Passenger class. It contains the ID field, which is primary key as generated value, and the name field. We define the constructor having the name argument and the constructor without arguments. We also defined the accessor methods for the ID and name fields.



We also have defined the Ticket entity class. It contains the ID field, which is primary key as generated value, and the number field. We defined the constructor having the number argument and the constructor without arguments. We also defined the accessor methods for the ID and number fields.

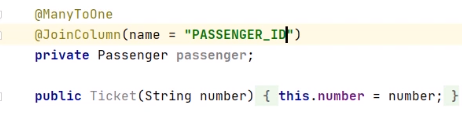


In the Passenger class, we add a new field, the list of tickets owned by a passenger, private List of Ticket tickets, and we initialize this list as a new ArrayList. We annotate the field with the OneToMany annotation, meaning that one passenger may own many tickets. The relationship is mapped by the many owning side more exactly by the passenger field.

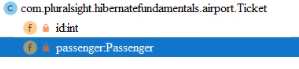


We'll add two new methods that will manage the tickets field. The public list of ticket getTickets method that will return Collections.unmodifiableList of tickets. From outside the Passenger class, we provide access only to an unmodifiable list of tickets so that the client of the class does not alter the content of the list. And we add the public void addTicket method that receives as argument a reference to a ticket. From outside the Passenger class, we allow to add the ticket to the list of tickets owned by a passenger. 

In the ticket class, we add as new field the passenger that owns that ticket.



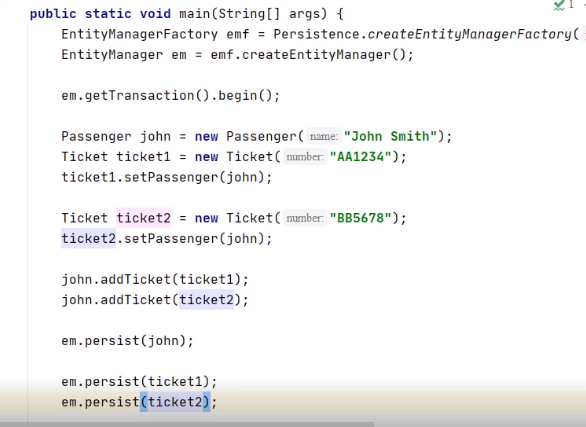
We annotate this new field with the ManyToOne annotation, indicating that many tickets may be owned by one passenger. We indicate the JoinColumn between the passengers and tickets as PASSENGER\_ID.



And we generate the Getter and the Setter for the passenger field.



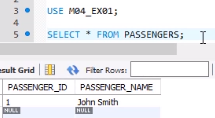
Moving to the main class, we have already configured the skeleton of the code, including the creation of the EntityManagerFactory of the EntityManager, the begin of the transaction, the commit of the transaction, and the closing of the EntityManagerFactory.



We create the new passenger, John Smith. We create the new ticket1 AA1234, and we set ticket1 as belonging to John. We create the new ticket tool with the number BB5678, and we set ticket2 as belonging to John. On the side of the passenger, we add ticket1 and ticket2 on the list of tickets owned by John. We persist the objects that we created one by one. We persist John, we persist ticket1, and we persist ticket2.



We move on the side of the database, and we have already created the M04\_EX01 database and commuted to it. And now we may run the Java program. We'll create the needed tables, we'll create the relationships between them, and we'll insert the information from our objects. Let's check the content of the two expected tables.

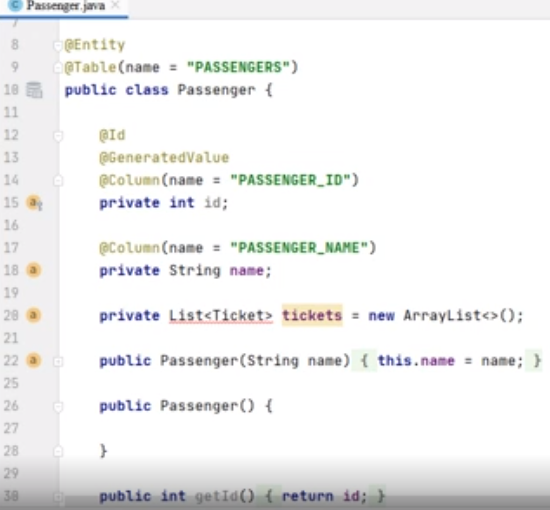


We execute select all from passengers. We expect John Smith to be there. And yes, we have the passenger inside the table. We finally verify the tickets table. We execute select all from tickets. We expect John Smith's tickets to be there, and there is a many‑to‑one relationship between tickets and passengers. And yes, we have the two tickets belonging to John. So, this concludes our demonstration. We proved how we can create one‑to‑many and many‑to‑one relationships, and we did this in particular between the passengers and the tickets.

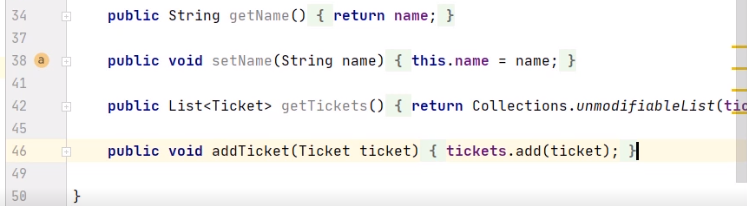
# Demo: Define Many-to-many Relationships

=>slides: Pg. 10

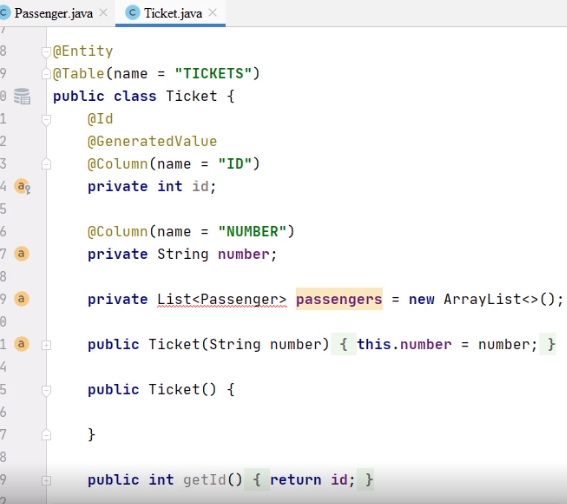
Let's move now to the next practical demonstration. We'll see how to put in practice defining many‑to‑many relationships. We start with a Maven project for which the pom.xml and persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence unit and the database name.



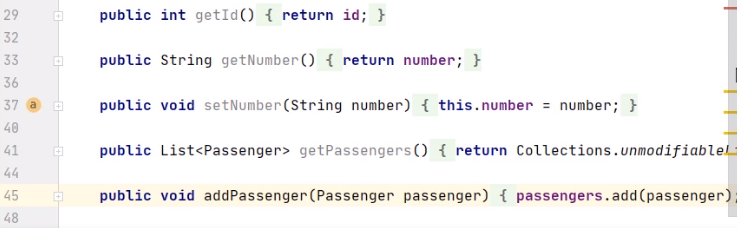
The project contains an Entity Passenger class corresponding to the PASSENGERS table. It contains the Id field, which is primary key as GeneratedValue, the name field, and the list of tickets belonging to that passenger. We defined the constructor having the name argument and the constructor without arguments.



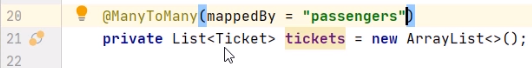
We defined the Accessor methods for the Id and name fields, the getTickets method that returns an unmodifiable list of tickets, and the addTicket method that adds the ticket to the list of tickets owned by the passenger.



We also have the Ticket Entity class corresponding to the TICKETS table. It contains the Id field, which is primary key as GeneratedValue, the number field, and the list of passengers from that ticket. We define a constructor having the number argument and the constructor without arguments.

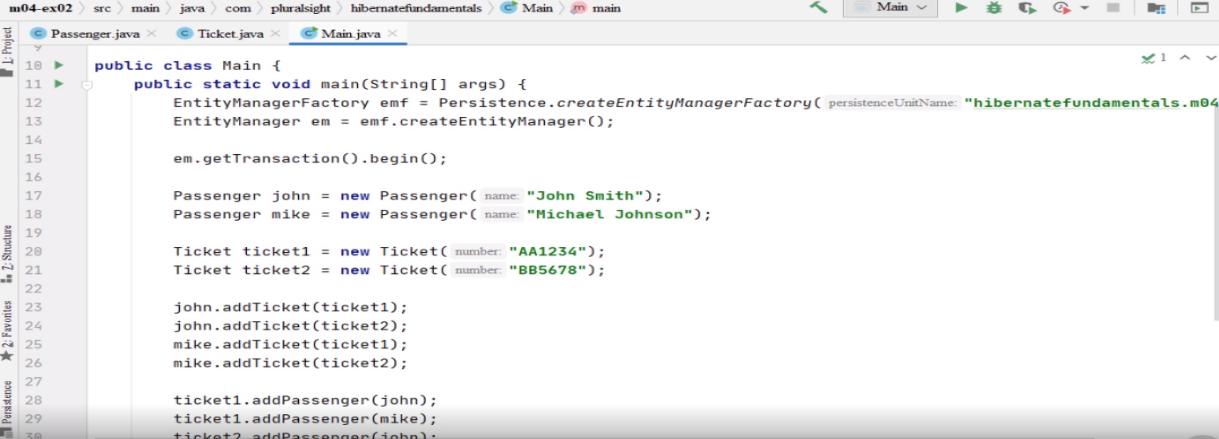


We define the Accessor methods for the Id and number fields, the getPassengers method that returns an unmodifiable list of passengers, and the addPassenger method that adds a passenger to the list of passengers from the ticket.

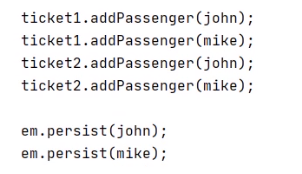


In the Passenger class, we annotate the tickets field with the ManyToMany annotation, meaning that many passengers may own many tickets. The relationship is mapped by the ticket side, more exactly by the passengers field.



In the Ticket class, we annotate the list of Passenger passengers field with the ManyToMany annotation, indicating that many tickets may be owned by many passengers. 

Moving to the Main class, we have already configured the skeleton of the code, including the creation of the EntityManagerFactory of the Entity Manager, the begin of the transaction, the commit of the transaction, and the closing of the EntityManagerFactory. We created two new passengers, John Smith and Michael Johnson, and two new tickets with the numbers AA1234 and BB5678. On the side of the passengers, we add ticket1 and ticket2 to the list of the tickets owned by John and to the list of the tickets owned by Mike.



On the side of the tickets, we added john and mike on the list of passengers from ticket1 and on the list of passengers from ticket2. We persist the objects that we created one by one. We persist (john), we persist (mike), we persist (ticket1), and we persist (ticket2).

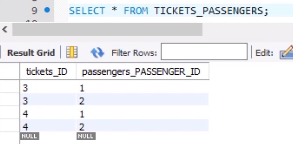


We move on the side of the database, and here we have already created the M04\_EX02 database and commuted to it with the USE command. And now we may run the Java program. We'll create the needed tables, we'll create the relationships between them, and the information from our objects. We notice that Hibernate created the intermediary table TICKETS\_PASSENGERS that will model the many‑to‑many relationship between the PASSENGERS and the TICKETS table. Such a table is needed for modeling many‑to‑many relationships. Let's check the content of expected tables.



We execute SELECT \* FROM PASSENGERS. We expect John Smith and Michael Johnson to be there. And yes, we have the passengers inside the table. We verify the TICKETS table. We execute 

SELECT \* FROM TICKETS. We expect the AA1234 and BB5678 tickets to be there. And yes, we have the two tickets here. We finally verify the TICKETS\_PASSENGERS table.



We expect it to reflect the many‑to‑many relationship between the two passengers and the two tickets. And yes, we see that each of the tickets belongs to each of the passengers and that each passenger owns each of the tickets. So, this concludes our demonstration. We proved how we can create many‑to‑many relationships, and we did this in particular between the passengers and the tickets.

# Annotations for Relationship Definition

=>slides: Pg. 11

Let's examine the annotations for relationship definition.

=>slides: Pg. 12

The annotations that are useful for relationship definition are JoinTable, JoinColumn, and JoinColumns.

=>slides: Pg. 13

JoinTable specifies the cross‑reference table for the mapping of a relationship. It must be specified on the owning side of a relationship. It can receive parameters as the name of the cross‑reference table, JoinColumns, the foreign key columns which reference the table of the entity that owns the relationship, or inverse JoinColumns, the foreign key columns which reference the table of the entity that does not own the relationship.

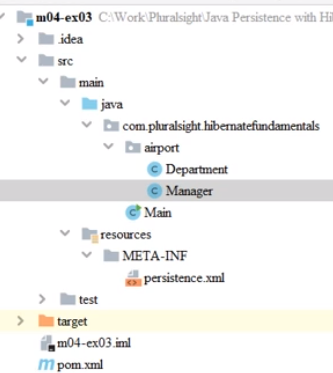
=>slides: Pg. 14

JoinColumn specifies a column for joining an entity association. It may receive various parameters as the name of the foreign key or the name of the column referenced by this foreign key column. =>slides: Pg. 15

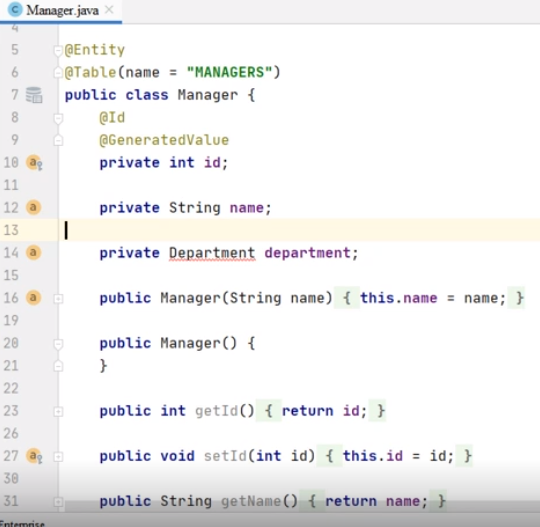
JoinColumns defines the mapping for composide foreign keys. It groups JoinColumn annotations. The value parameter specifies an array of JoinColumn defining a composide foreign key.

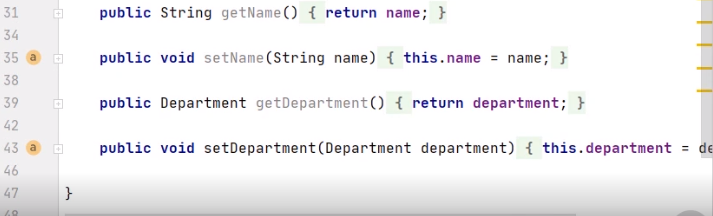
# Demo: Join Tables on One Column

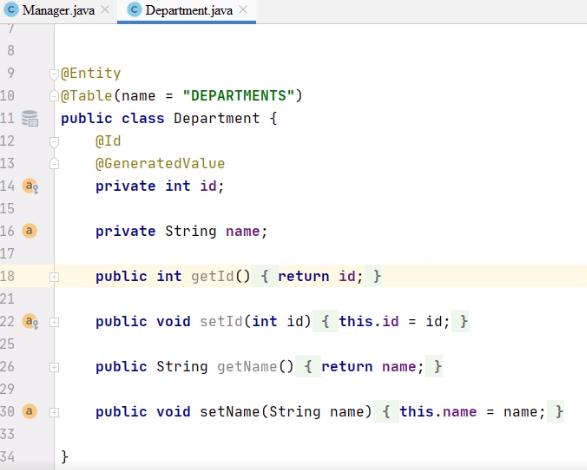
=>slides: Pg. 16

Let's move to the practical demonstration. We'll see how to put in practice joining tables on one column. 

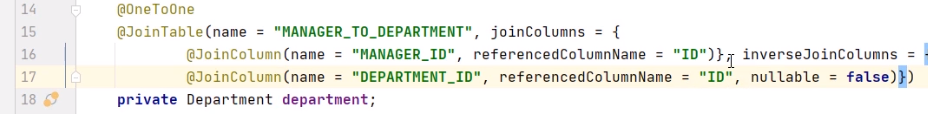
We start with a Maven project for which the pom.xml and persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence unit and the database name.

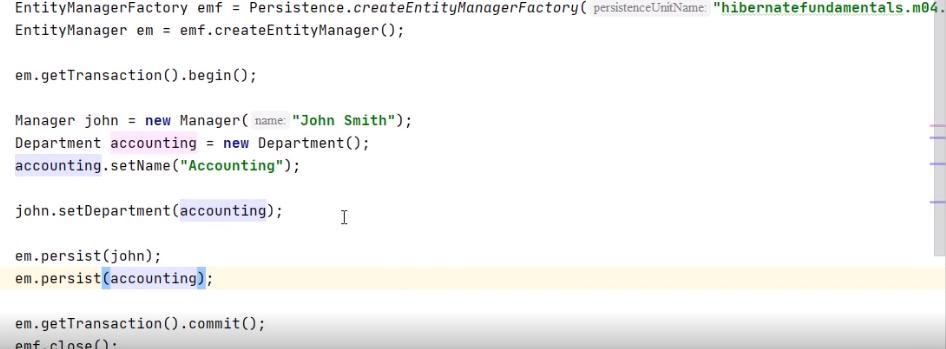


The project contains an entity Manager class, corresponding to the MANAGERS table. It contains the ID field, which is primary key as generated value, the name field, and the department managed by the manager. We defined the constructor having the name argument and the constructor without arguments. We defined the accessor methods for the ID, name, and department fields.



We also have the Department entity class corresponding to the DEPARTMENTS table. It contains the ID field, which is primary key as generated value, and the name field. We defined the accessor methods for the ID and name fields.



In the Manager class, we annotate the department field with a OneToOne annotation, as one manager leads one department. We'll indicate the JoinTable between manager and department as being the MANAGER\_TO\_DEPARTMENT table. The JoinColumn will be named MANAGER\_ID, referencing the ID column from the MANAGERS table. We defined the inverseJoinColumn to be named DEPARTMENT\_ID, referencing the ID column from the DEPARTMENTS table and being not nullable. 

In the main class, we have already configured the persistent skeleton and the creation of the objects and the link between them. We persisted two objects, the manager John and the accounting department.



We move on the side of the database, and here we have already created the M04\_EX03 database and commuted to it with the use command. And now we may run the Java program. We'll create the \_\_\_\_\_ tables, we'll create the relationships between them, and we'll insert the information from our objects. We noticed that besides the MANAGERS and DEPARTMENTS tables, Hibernate created the intermediary table MANAGER\_TO\_DEPARTMENT, as we specified using the join table annotation inside our code. This table will model the one‑to‑one relationship between the MANAGERS and the DEPARTMENTS table. Let's check the content of expected tables.

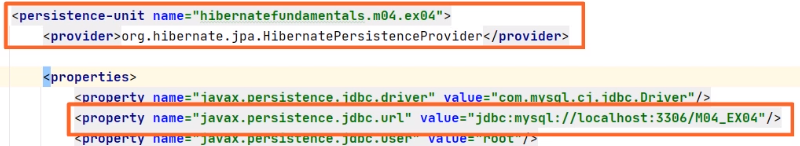


We execute select all from managers. We expect John Smith to be there, and yes, we have this manager inside the table. We verify the DEPARTMENTS table. We execute select all from departments. We expect the accounting department to be there. And yes, we have it here. We finally verified the MANAGER\_TO\_DEPARTMENT table. We expect it to reflect the one‑to‑one relationship between the MANAGERS and the DEPARTMENTS. And yes, we see that there is one manager leading one department. So, this concludes our demonstration. We proved how we can join tables on one column, and we did this in particular between the managers and the departments.

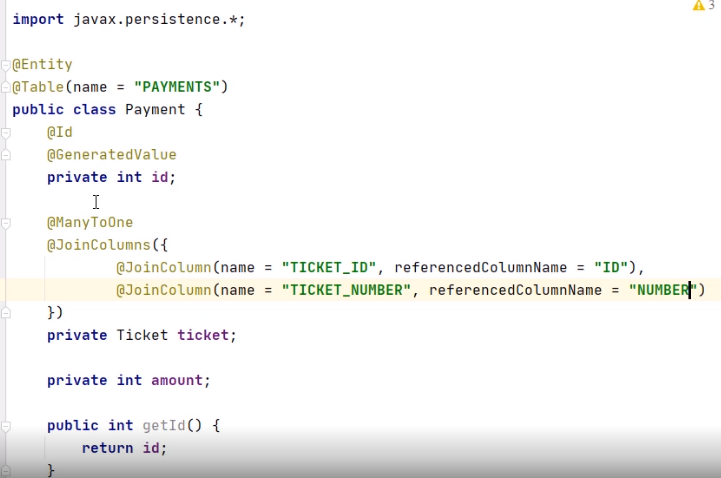
# Demo: Join Tables on Multiple Columns

=>slides: Pg. 17

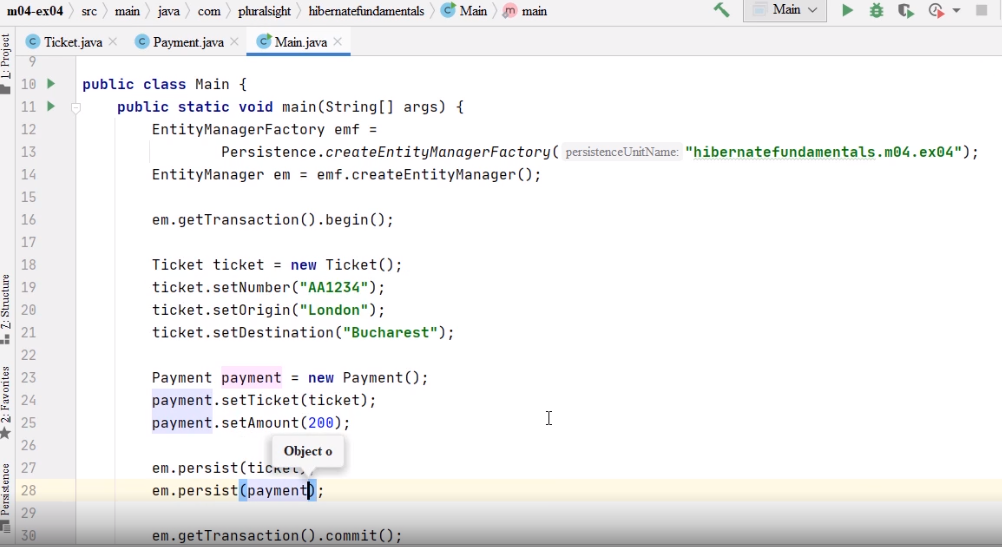
We move to the next practical demonstration. We'll see how to put in practice joining tables on multiple columns.



We start with a Maven project for which the pom.xml and the persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence unit and the database name. The project contains an entity ticket and an entity payment. There is a many‑to‑one relationship between payments and tickets. There may be many payments for the same ticket.



Using the JoinColumns annotation, we indicate the joined columns between the two entities. The first JoinColumn, TICKET\_ID, referencing the ID column from TICKETS, and the second JoinColumn TICKET\_NUMBER referencing the NUMBER column from TICKETS.



In the Main class, we have already configured the persistent skeleton, the creation of the objects, and the link between them. We still have to persist the two objects that we have created, the ticket and the payment corresponding to the ticket.



We move on the side of the database, and here we have already created the M04\_EX04 database and commuted to it with the USE command. And now we may run the Java program. We'll create the needed tables, we'll create the relationships between them, and we'll insert the information from our objects. We notice that there is a TICKETS table and the PAYMENTS table where we inserted the rows with the information from the objects. Let's check the content of the two expected tables.



We execute a SELECT \* FROM TICKETS. We expect the newly introduced ticket to be there. And yes, we have it inside the table id 1 and NUMBER AA1234. We verify the PAYMENTS table.



We execute SELECT \* FROM PAYMENTS. We expect the newly introduced payment to be there. And, yes, we have it here. We notice that it has the TICKET\_ID column with the value 1 and the TICKET\_NUMBER column with the value AA1234, the same values as in the TICKETS table, so we are correctly making the join on these two columns. So, this concludes our demonstration. We proved how we can join tables on multiple columns, and we did this in particular between the tickets and the payments.

# Embeddable Classes

=>slides: Pg. 18

Let's examine the annotations for embeddable classes.

=>slides: Pg. 19

Embeddable classes are fine‑grained classes representing entity state. They do not have persistent identity of their own. They cannot be shared across persistent entities. An entity may have collections of embeddables, as well as single‑valued embeddable attributes. And they exist only as part of the state of the entity to which they belong.

=>slides: Pg. 20

Embeddable classes must be annotated as @Embeddable. The customization of embeddable classes' mapping can be done with the help of @AttributeOverride, which overrides the mapping for a particular field or property of an embeddable class, and with @AttributeOverrides, which overrides the mappings of multiple properties or fields.

=>slides: Pg. 21

JPA 2.0 supports having collections of basic types or embeddable classes for an entity. Most useful annotations are @ElementCollection, which defines collection of instances of a basic type or embeddable class, or @Collection Table, which specifies the table that is used for the mapping of collections of the basic or embeddable types.

=>slides: Pg. 22

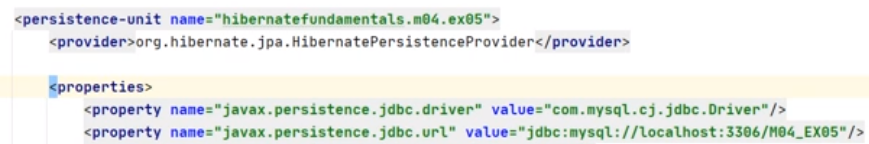
All Java collection types‑‑collection, list, set, map‑‑are supported. JPA 2.0 also supports mapping embeddable classes and basic types to map. The key and the value may be either a basic type or an embeddable class, at least one of the keys or values must be a basic type. The useful annotations are @MapKeyColumn, which specifies the column name for the map key if the key is a basic type, and @Column, which specifies the column for the map value if the value is a basic type.

# Demo: Embedding Classes in Entities

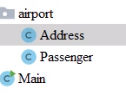
=>slides: Pg. 23

Let's move to the practical demonstration. We'll see how to put in practice embedding classes in entities.

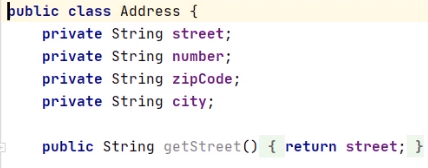




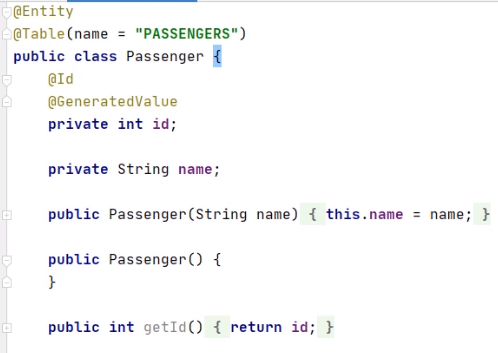
We start with a Maven project for which the pom.xml and the persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence‑unit and the database name.



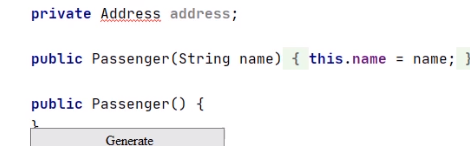
The project contains an entity Address



with the fields street, number, zipCode, and city, and the accessor methods for them.



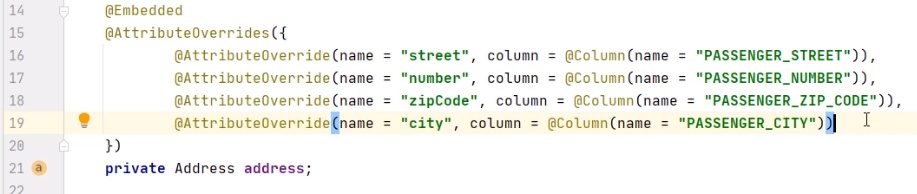
The project also contains an entity Passenger with the fields id, primary key as generated value, and name. The entity corresponds to the PASSENGERS table. It has a constructor with the name argument and a constructor without arguments, and accessor methods for the fields.



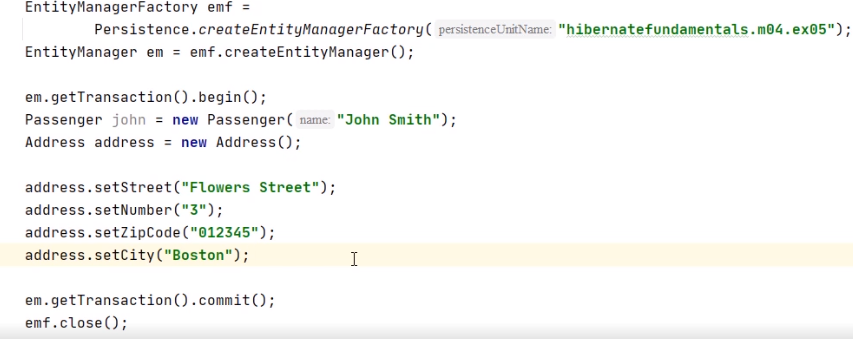
We add here a private Address address field, the address of the given passenger. We will generate the getter and the setter for this newly introduced field.

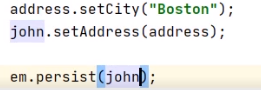


We annotate the Address class with Embeddable. It does not have a persistent identity on its own, and it is designed to be embedded in another entity.



Back in the Passenger class, we annotate the address field with Embedded, indicating that we have this address embedded in the Passenger. We'll need to provide meta information through annotations to override the mappings of multiple properties or fields, as we first used for this, the AttributeOverrides annotation. Then we override for attributes one by one. The AttributeOverride annotation is used to override the mapping of one field or column. We override the mapping of the street field to the column with the name PASSENGER\_STREET, we override the mapping of the number field to the column with the name PASSENGER\_NUMBER, we override the mapping of the zipCode field to the column with the name PASSENGER\_ZIP\_CODE, and we override the mapping of the city field to the column with the name PASSENGER\_CITY.



In the Main class, we have already configured the Persistence skeleton and the creation of the objects. 

We set John's address to be one that we created here, and we persist the john object. We move on the side of the database,



and here we have already created the M04\_EX05 database and commuted to it with the USE command. And now we may run the Java program. We'll create the needed table, and we'll insert the information from our objects. We noticed that there is one passenger table where we inserted the role with the information from both the passenger and the address object. Let's check the content of the expected table.



We execute SELECT \* FROM PASSENGERS. We expect the newly introduced passenger to be there together with the address information.



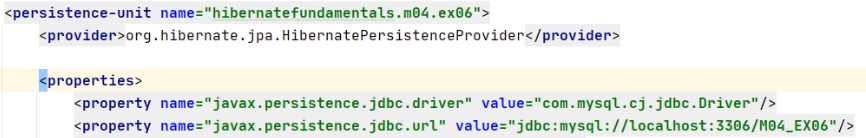
And, yes, we have it inside the table. We notice that the row contains both the information from the passenger object, id and number, and the embedded information from the address object, PASSENGER\_STREET, PASSENGER\_NUMBER, PASSENGER\_ZIP\_CODE, and PASSENGER\_CITY. So, this concludes our demonstration. We proved how we can embed classes in entities, and we did this, in particular, between the passengers and the addresses.

# Demo: Embedding Collections of Classes in Entities

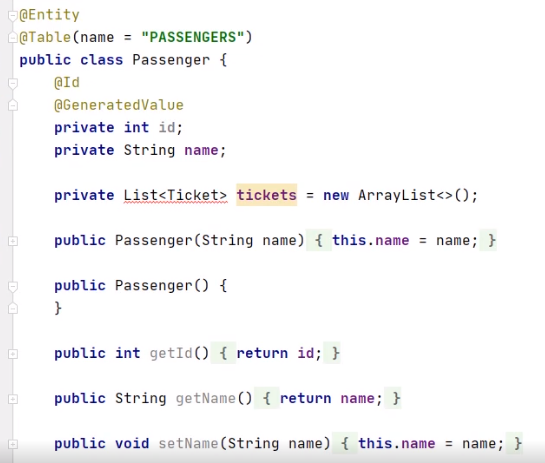
=>slides: Pg. 24

We'll move to the next practical demonstration. We'll see how to put in practice embedding collections of classes in entities.

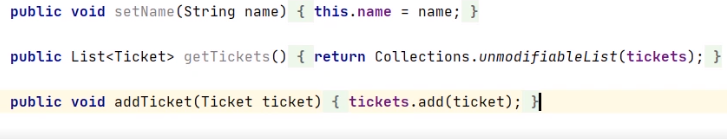




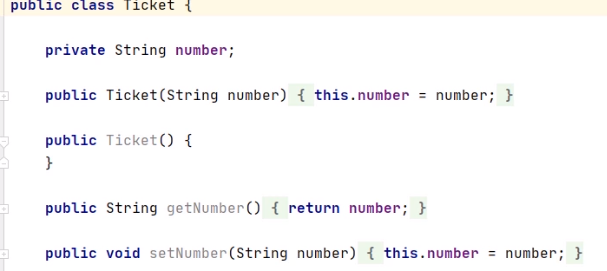
We start with a Maven project for which the pom.xml and the persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence‑unit and the database name.



The project contains an entity, PASSENGER, with the field's ID, primary key as GeneratedValue, name, and a list of tickets. The entity corresponds to the table PASSENGERS. It has a constructor with the name argument and a constructor without arguments. It also has accessor methods for the fields.



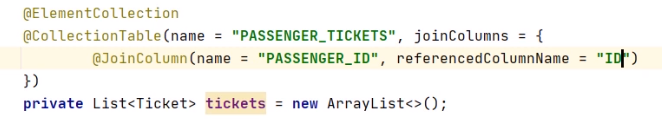
The project also contains an entity,



Ticket, with a number field, a constructor with a number argument, a constructor without arguments, and accessor methods.



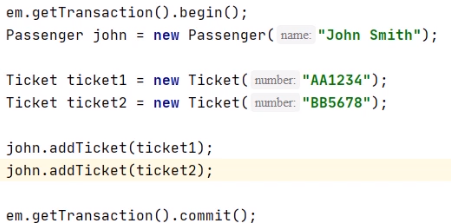
We annotate the Ticket class with @Embeddable. It does not have a persistent identity on its own. It is designed to be embedded in another entity.

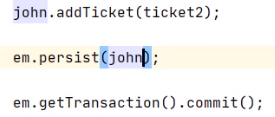


Back in the Passenger class, we annotate the List<Ticket> field with @ElementCollection, mapping the embeddable tickets nonentities. The @CollectionTable annotation specifies the table that is used for the mapping of collections of the embeddable ticket type. We indicate the name of the table to be PASSENGER\_TICKETS, the @JoinColumn of the @CollectionTable, which references the primary table of the entity to be PASSENGER\_ID, and the name of the referenced column in the parent Ticket table to be ID.



We move on the side of the database, and here we have already created the M04\_EX06 database and commuted to it with the USE command.



In the Main class, we have already configured the Persistence skeleton and the creation of the objects. We create the Passenger, John Smith, and two tickets, and we add these ones on John's list of tickets. 

We also need to persist that John object. And now we may run the Java program. We'll create the needed tables, PASSENGERS and PASSENGER\_TICKETS, and we'll insert the information from our objects. Let's check the content of the expected tables.



We execute SELECT \* FROM PASSENGERS.



We expect the newly introduced passenger to be there. And, yes, we have John Smith inside the table. 

We execute SELECT \* FROM PASSENGER\_TICKETS. We expect the newly introduced tickets to be there.



And, yes, we have the tickets with numbers AA1234 and BB5678 belonging to the PASSENGER\_ID 1. This is John Smith. So this concludes our demonstration. We proved how we can embed collections of classes in entities. And we did this in particular between the passengers and the tickets.

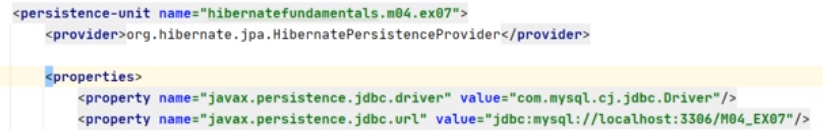
# Demo: Embedding Maps of Classes in Entities

=>slides: Pg. 25

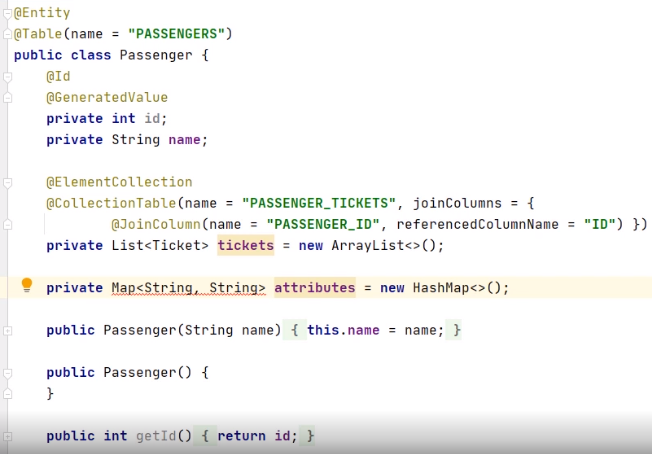
We move to the next practical demonstration. We'll see how to put in practice embedding maps of classes in entities.

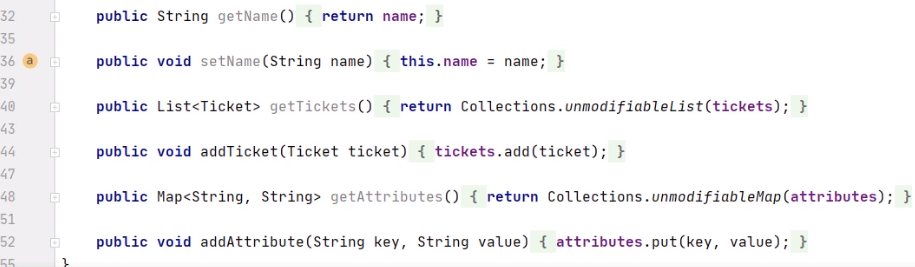
We start with our Maven project,



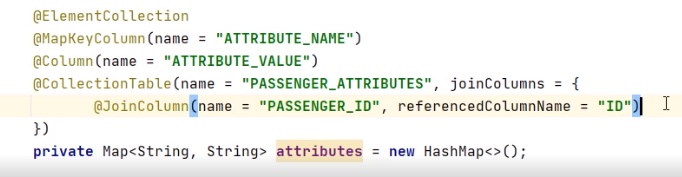
for which the pom.xml and the persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence‑unit and the database name. The project already contains the entity, PASSENGER,



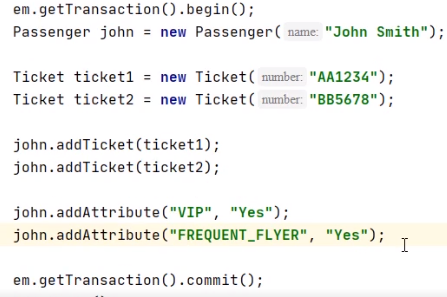
with the fields id, primary key as GeneratedValue, name, and the list of tickets.



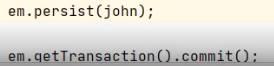
We added the attributes of the passenger that will be kept as pairs of strings, the name of the attribute, and the value of the attribute. So, they belong to a map. The entity corresponds to the Passenger's table. It has a constructor with the name argument and the constructor without arguments. We also have the accessor methods, including accessor methods for the attributes field.



We annotate the attributes map field with ElementCollection, mapping the attribute's non‑entities. The MapKeyColumn annotation specifies the name of the key field to be ATTRIBUTE\_NAME. The column annotation specifies the name of the value field to be ATTRIBUTE\_VALUE. The CollectionTable annotation specifies the table that is used for the mapping of the attributes. We indicate the name of the table to be PASSENGER\_ATRRIBUTES, the JoinColumn of the CollectionTable, which references the primary key of the entity, to be PASSENGER\_ID, and the name of the reference column in the parent passenger's table to be ID.



In the Main class, we have already configured the persistent skeleton and the creation of the objects. We create the passenger, John Smith, and we add two attributes to John. The values of the VIP attribute is Yes, and the value of the FREQUENT\_FLYER attribute is Yes.



We need to persist that John object.



We move on the side of the database, and here, we have already created the M04\_EX07 database and commuted to it with the USE command. And now, we may run the Java program. We'll create the needed tables, PASSENGERS, PASSENGER\_TICKETS, and PASSENGER\_ATRRIBUTES, and we insert the information from our objects. We focus for this example to the PASSENGERS and PASSENGER\_ATTRIBUTES tables. Let's check the content of the expected tables.







We execute SELECT \* FROM PASSENGERS. We expect the newly‑introduced passenger to be there. And yes, we have John Smith inside the table. We execute SELECT \* FROM PASSENGER\_ATTRIBUTES. We expect the newly‑introduced attributes to be there. And yes, we will have two rows, the passenger with the ID 1 having the attribute name VIP and the ATTRIBUTE\_VALUE Yes and the passenger with the ID 1 having the ATTRIBUTE\_NAME FREQUENT FLYER and the ATTRIBUTE\_VALUE, Yes. So, this concludes our demonstration. We proved how we can embed maps of classes in entities, and we did this, in particular, between the passengers and the attributes.

=>slides: Pg. 27

To conclude this module, let's have a look at what we have covered. We analyzed the types and directions of relationships, we used annotations to define relationships, we used embedded classes to represent the state of an entity and demonstrated various ways to do it, and we used entities having collections of embedded classes or maps of embedded classes. Our next module will demonstrate how to work with entity inheritance.

# Queries and Relationship Direction

=>slides: Pg. 26-27

Let’s take a look at the unidirectional mapping on the OrderItem entity first. The OrderItem entity represents the many side of the relationship and the OrderItem table contains the foreign key of the record in the Order table.

As you can see in the following code snippet, you can model this association with an attribute of type Order and a @ManyToOne annotation. The Order order attribute models the association, and the annotation tells Hibernate how to map it to the database.

That is all you need to do to model this association. By default, Hibernate generates the name of the foreign key column based on the name of the relationship mapping attribute and the name of the primary key attribute. In this example, Hibernate would use a column with the name order\_id to store the foreign key to the Order entity.

You can now use this association in your business code to get the Order for a given OrderItem and to add or remove an OrderItem to or from an existing Order.

# Cascading Events

=>slides: Pg. 28- 31

In fact, we just need to customize the relationship mapping. By default, no persist event is cascaded, but by adding a OneToMany annotation with this cascade attribute set to PERSIST, then the job is done. Here, the CD entity propagates the PERSIST event to the Musician relationship. The cascade element can take an array of events. Here, the relationship cascades the PERSIST, REMOVE, and MERGE events from the EntityManager. And if we need to cascade all the events, we use ALL. What we have to keep in mind is that by default events are not propagated to other entities that have a relationship. Cascading an event is only possible if the mapping of the relationship is customized. This can be done on the annotation OneToOne, OneToMany, ManyToOne, and ManyToMany. They all have a cascade attribute that takes an array of events to be cascaded. These events are PERSIST, REMOVE, MERGE, and ALL, which gathers all the events. For example, if we cascade a REMOVE event calling an EntityManager REMOVE method on CD, it will also remove all the musicians. But as you can see, there is no find event. So what happens when we find a CD? Are the musicians also retrieved from the database? Well, that depends on how we fetch our relationships.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*