5. Entity Inheritance

Contents

[1. Introducing Entity Inheritance 1](#_Toc31959)

[2. Demo: Extend One Entity 1](#_Toc6304)

[3. Demo: Extend One Non-entity 1](#_Toc151)

[4. Mapping Strategies 1](#_Toc7532)

[5. Demo: Single Table per Class Hierarchy 1](#_Toc29870)

[6. Demo: Joined Subclass Strategy 2](#_Toc8660)

[7. Demo: Table per Concrete Class Strategy 2](#_Toc3343)

[8. Demo: Conversion 2](#_Toc4362)

# Introducing Entity Inheritance

=>slides: Pg. 1

Hello. Welcome to this module about entity inheritance. We'll examine now the way we can transpose from the object‑oriented world to the relational world, one of the features that is particular only to OOP, the inheritance. So let's dive into it.

=>slides: Pg. 2

This module will examine how to inherit from entities and non‑entities and to define a class hierarchy, how to work with mapping strategies, and how to choose the appropriate mapping strategy for a particular situation, how to work with converters when the data representation is different from the Java class to the database record.

=>slides: Pg. 3

An entity class may inherit from another entity class. An abstract class can be specified as an entity. Of course, the OOP principle still applies, and the class cannot be directly instantiated. The abstract class will define persistent state to be inherited by its subclasses. An abstract entity class will be annotated with the Entity annotation, will be mapped as an entity, and can be the target of queries. JPA supports polymorphic associations and queries for entities.

=>slides: Pg. 4

An entity may inherit from a superclass that provides persistent entity state and mapping information, but which is not itself an entity. The purpose of such a mapped superclass is to define state and mapping information that is common to multiple entity subclasses. It is like a template for entities without its own persistence. The mapped superclass cannot be queried. The mapped superclass may define only unidirectional relationships.

=>slides: Pg. 5

The MappedSuperclass annotation is used to specify a class being a mapped superclass. The AttributeOverride and AssociationOverride annotations are used to override mappings for the concrete classes.

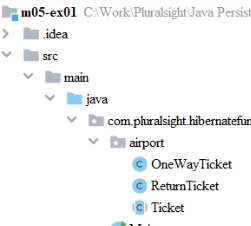
=>slides: Pg. 6

An entity can have as superclass a non‑entity. This one may be an abstract or a concrete class. The state of such a non‑entity superclass is not persistent. It is used to inherit only the behavior. The persistence framework will ignore the annotations on this kind of superclass.

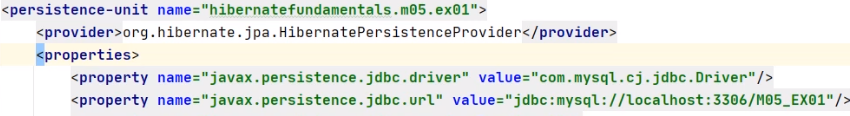
# Demo: Extend One Entity

=>slides: Pg. 7

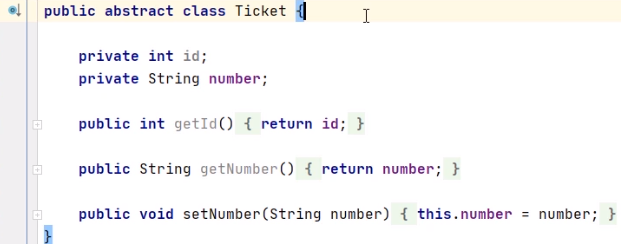
Let's move to the practical demonstration. We'll implement an example in which classes extend one entity and inherit persistent state.



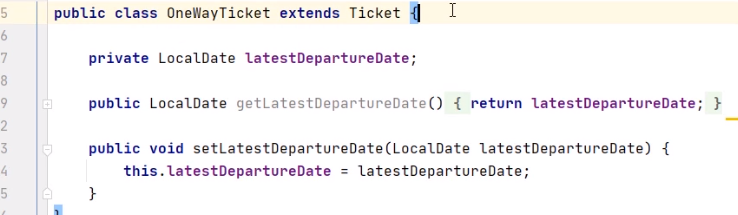
We start with our Maven project, for which the pom.xml



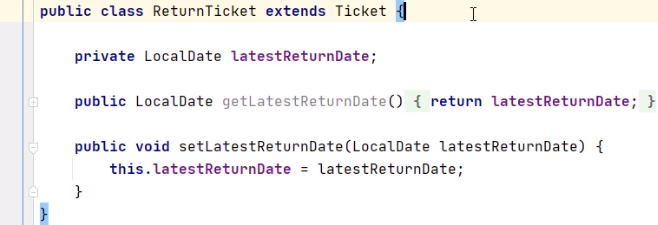
and the persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence unit and the database name.

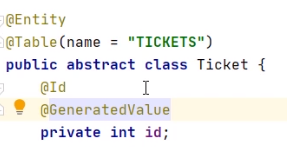


The project already contains the Ticket abstract class, which contains the ticket id, the ticket number, and accessor methods for them.



The OneWayTicket class extends the Ticket class, and it adds the latestDepartureDate field. We also have the getter and the setter method for it.

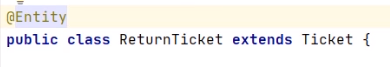
The ReturnTicket class extends the Ticket class, and it adds the latestReturnDate field. We also have the getter and the setter method for it.



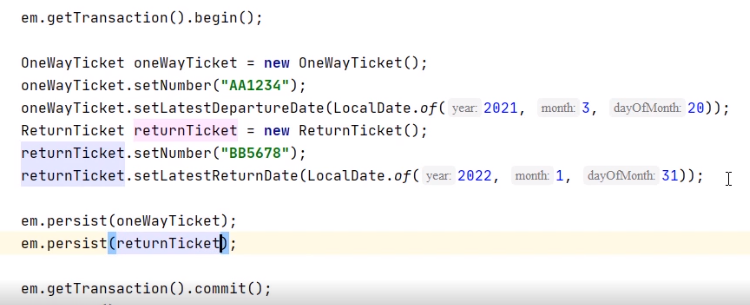
We move back to the Ticket class, and we annotate it with Entity and with Table name = TICKETS, indicating the corresponding table in the database. We annotate the id field with the Id annotation, indicating that it is a primary key, and with GeneratedValue, indicating that the value of this field is given by the database.



We annotate the OneWayTicket class with Entity, indicating that it has a corresponding table in the database.



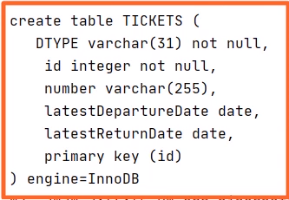
We annotate the ReturnTicketClass with Entity, indicating that it has a corresponding table in the database.



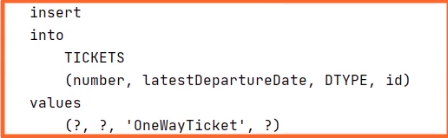
In the Main class, we have already configured the persistent skeleton, the creation and the configuration of one oneWayTicket object and of one returnTicket object. We need to persist the oneWayTicket and the returnTicket objects. We move on the side of the database.



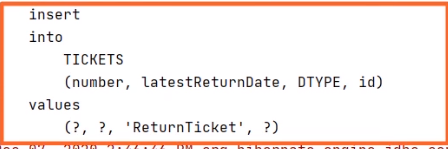
We have already created the M05\_EX01 database and committed to it with the USE command. And now, we may run the Java program. We'll create the needed table, TICKETS, and we'll insert the information from our objects.



We notice the creation of the TICKETS table that contains both the latestDepartureColumn specific to a OneWayTicket and the latestReturnDate column specific to a ReturnTicket. The table also has a discriminator, DTYPE column, that indicates the type of the ticket, one way or return. We notice the existence of the id and number columns inherited from the Ticket entity abstract class.



We notice the insertion of two rows. One is a not null latestDepartureDate column, and DTYPE equals to OneWayTicket.



One is a not null latestReturnDate column and the DTYPE equals the ReturnTicket. Let's check the content of the expected tables.



We execute SELECT all FROM TICKETS.

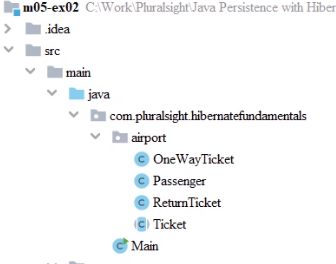


We expect both newly introduced tickets to be there. And yes, we have one OneWayTicket inside the table with a not null latestDepartureDate column and with a null latestReturnDate column, and we have one ReturnTicket inside the table with a not null latestReturnDate column and with a null latestDepartureDate column. We notice the existence of the id and number columns inherited from the TicketEntity abstract class. So this concludes our demonstration. We proved how we can extend one entity represented by an abstract class and inherit persistent state. In particular, we did this using tickets.

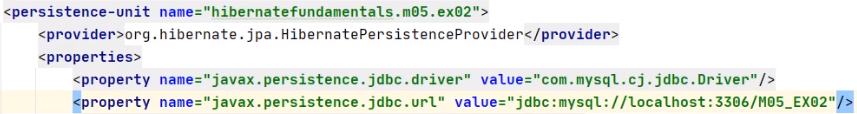
# Demo: Extend One Non-entity

=>slides: Pg. 8

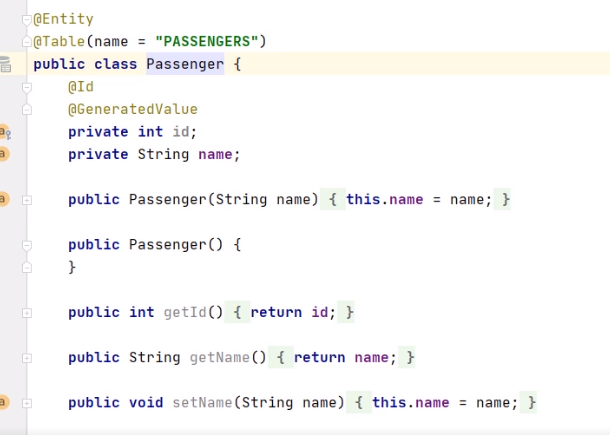
We move to the next practical demonstration. We'll implement an example in which classes extend one non‑entity and do not inherit any persistent state.



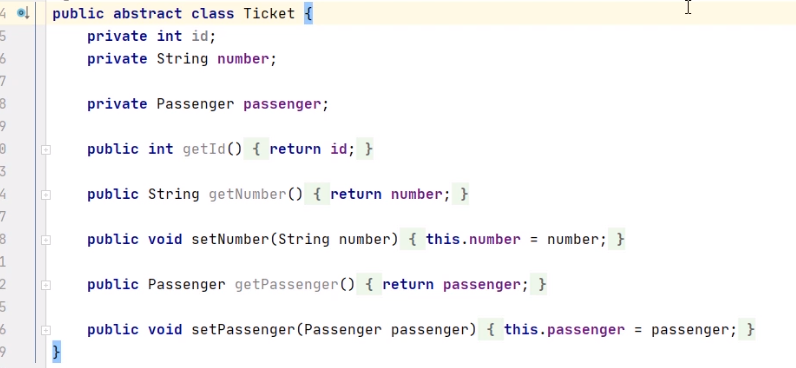
We start with our Maven project, for which the pom.xml



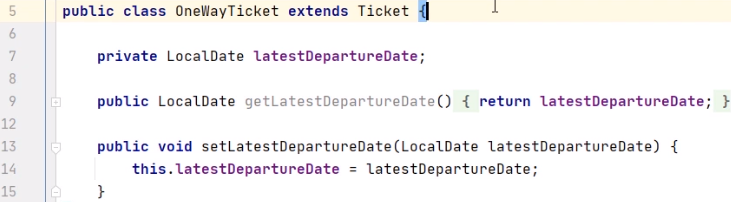
and the persistence.xml are configured similarly to what we presented in the previous demonstrations. We made sure we changed the persistence unit and the database name.



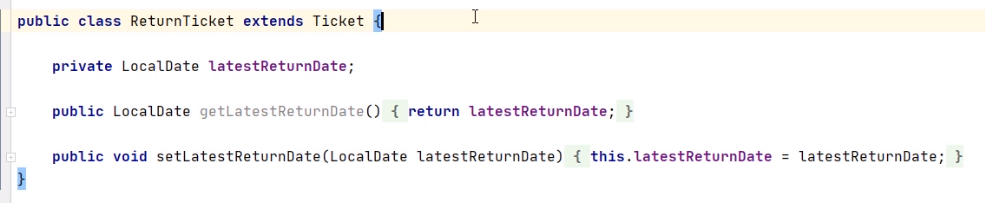
The project already contains the Passenger class, annotated as Entity and with the Table name = PASSENGERS annotation, indicating the corresponding table in the database. The id field is annotated with Id, indicating it is a primary key, and with GeneratedValue, indicating that the value of the field is provided by the database.



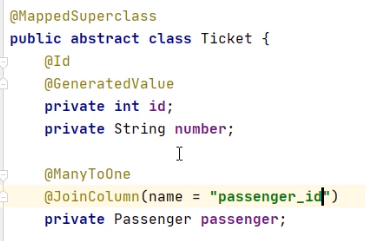
The project also contains the Ticket abstract class, which contains the ticket id, the ticket number, and accessor methods for them.



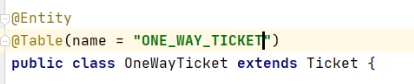
The OneWayTicket class extends the Ticket class, and it adds the latestDepartureDate field. We also have the getter and the setter method for it.



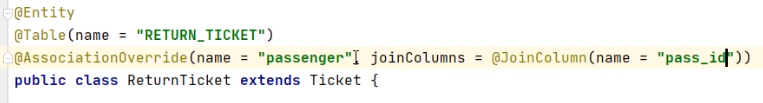
The ReturnTicket class extends the Ticket class, and it adds the latestReturnDate field. We also have the getter and the setter method for it.

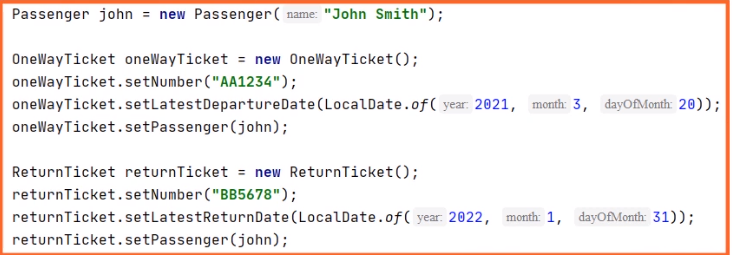


We move back to the Ticket class, and we annotate it with MappedSuperclass, indicating that it does not have its own persistence, but the defined state and mapping information common to multiple entity subclasses. We annotate the id field with the Id annotation, indicating that it is a primary key, and with GeneratedValue, indicating that the value of this field is given by the database. We annotate the passenger field with ManyToOne, indicating that many tickets may be owned by one passenger, and we also indicate the JoinColumn as passenger\_id.

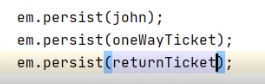


We annotate the OneWayTicket class with Entity, indicating that it has a corresponding table in the database. And we indicate the name of the table to be ONE\_WAY\_TICKET.

We annotate the ReturnTicket class with Entity, indicating that it has a corresponding table in the database, and we indicate the name of the table to be RETURN\_TICKET. We override the association with the passenger using the AssociationOverrideAnnotation. The JoinColumn will be pass\_id instead of the previously defined passengerid.



In the Main class, we have already configured the persistent skeleton and the creation and the configuration of a passenger object of one oneWayTicket object and of one returnTicket object.



We need to persist one by one the passenger john, the oneWayTicket, and the returnTicket objects. We move on the side of the database, and here we have already created the M05\_EX02 database and committed to it with the USE command. And now, we may run the Java program. We'll create the needed tables, PASSENGERS, ONE\_WAY\_TICKET, and RETURN\_TICKET, and we'll insert the information from our objects. Let's check the content of the expected tables.



We execute SELECT all FROM PASSESNGERS.



We expect the newly introduced passenger to be there. And yes, we have the passenger John Smith having the id 1.



We execute SELECT all FROM ONE\_WAY\_TICKET.



The AA1234 ticket should be there belonging to the passenger with the id 1 referenced through the passenger\_id column



We execute SELECT all FROM RETURN\_TICKET.



The BB5678 ticket should be there belonging to the passenger with the id 1 reference through the pass\_id column as we have overridden the associations through the AssociationOverride annotation So this concludes our demonstration. We proved how we can extend one non‑entity represented by an abstract class and make it the base class for multiple entity subclasses. In particular, we did this using tickets.

# Mapping Strategies

=>slides: Pg. 9

We'll examine now the mapping strategies and the ways to apply them in Hibernate.

=>slides: Pg. 10

There are three basic strategies that may be used when mapping a class or a class hierarchy to relational database. Single table per class hierarchy. One single table will be the result of the entire class hierarchy. Joined subclass strategy. Fields that are specific to a subclass are mapped to a separate table than the fields that are common to the parent class, and the join is performed to instantiate the subclass. Table per concrete entity class. Only the concrete classes will be transformed in tables, there will be no concrete table for the super class.

=>slides: Pg. 11

The class hierarchy that needs to be persisted is shown here. There is a ticket superclass and it has two subclasses, OneWayTicket and ReturnTicket. We will apply the three strategies we mentioned and will then examine the resulted entities diagram.

=>slides: Pg. 12

Applying the single table per class hierarchy will get one single table, tickets, that will be the result of the entire class hierarchy that we have shown. The tickets table contains the union of all fields from the superclass and from the subclasses. ID, number, origin, destination, and price are taken from the ticket superclass. LATEST\_DEPARTURE\_DATE is taken from OneWayTicket, and LATEST\_RETURN\_DATE is taken from ReturnTicket. Consequently, when inserting a one way ticket, latest return date will be null, and when inserting a return ticket, latest departure date will be null.

=>slides: Pg. 13

Using the single table per class hierarchy strategy, all the classes in the hierarchy will be mapped to a single table. The table has a column that serves as a discriminator column. The values of this column identify the specific surplus. As advantage, the strategy provides good support for polymorphic relationships. As disadvantage, it requires that the columns corresponding to states specific to the subclasses are nullable.

=>slides: Pg. 14

We examine now the annotations to use for the mapping strategies. Inheritance defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity class that is the root of the entity class hierarchy. The default strategy is inheritanceType.singleTable. The DiscriminatorColumn is only specified in the root of our entity class hierarchy. If the annotation is missing the name of the DiscriminatorColumn, its default name will be the type, and its type will be string. DiscriminatorValue specifies the value of the DiscriminatorColumn for entities of the given type. It can only be specified on the concrete entity classes. If the annotation is not specified and the DiscriminatorColumn is used, the default value of the discriminator will be the class name.

=>slides: Pg. 15

For the joined subclass strategy, the fields that are specific to a subclass are mapped to a separate table, as we see here. The ONE\_WAY\_TICKETS table contains the LATEST\_DEPARTURE\_DATE column, and the RETURN\_TICKETS table contains the LATEST\_RETURN\_DATE column.

=>slides: Pg. 16

Applying the joined subclass strategy, the root of the class hierarchy is represented by a single table. Each subclass is represented by a separate table that contains fields that are specific to that subclass. The join between the tables is made through the primary key column of the superclass table, which serves as foreign key to the primary key of the subclass table. As advantage, the strategy provides good support for polymorphic relationships. As disadvantage, it requires one or more join operations to be performed between the entity modeling the subclass and the entity modeling the superclass. Joins may generally lead to bad performance.

=>slides: Pg. 17

Using the table per concrete class strategy, only the concrete classes will be transformed in tables. Here, we see that there is no tickets table, and we have two tables, ONE\_WAY\_TICKETS and RETURN\_TICKETS. The fields that were originally found in the ticket superclass are now duplicated. ID, number, origin, destination, and price are to be found both in the ONE\_WAY\_TICKETS and RETURN\_TICKETS tables. In addition, ONE\_WAY\_TICKETS contains the LATEST\_DEPARTURE\_DATE column and RETURN\_TICKETS contains the LATEST\_RETURN\_DATE column.

=>slides: Pg. 18

Using the table per concrete class strategy, each subclass is mapped to a separate table. All properties of the class, including hierarchy inherited properties, are mapped to columns of the table for the class. This strategy provides poor support for polymorphic relationships. It typically requires an SQL union for queries addressing the class hierarchy. It may lead to possible problems using ID generation strategies. The IDs may be duplicated between the tables that result from the same hierarchy.

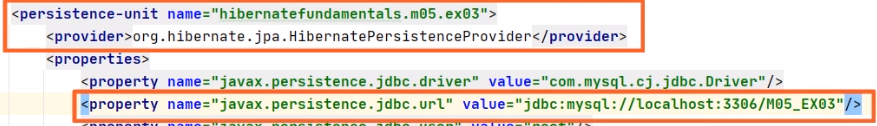
=>slides: Pg. 19

One more problem to examine now is the one referring to storing the values that are represented in Java in a different way as in the database. For example, a Java boolean may arrive in a database as 0/1 values or as strings True/False, or Yes/No. JPA 2.1 provides the interface, javax.persistence.AttributeConverter, having two methods for converting the entity attribute state into database column representation and vice‑versa. Once we implement our own converter, we can use it to specify the conversion of the field or property.

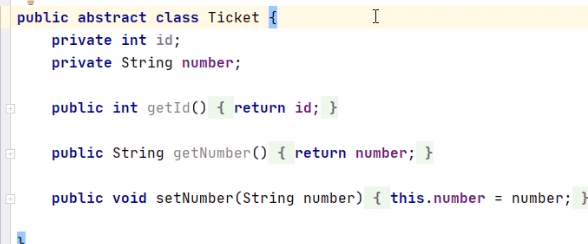
# Demo: Single Table per Class Hierarchy

=>slides: Pg. 20

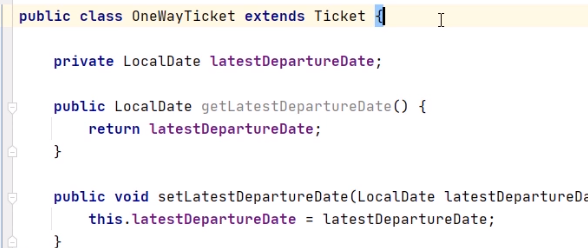
We'll move to another practical demonstration. We'll implement an example in which the class hierarchy will be mapped using single table per class hierarchy. We'll start with our Maven project, for which the pom.xml and the persistence.xml are configured similarly to what we presented in the previous demonstrations.



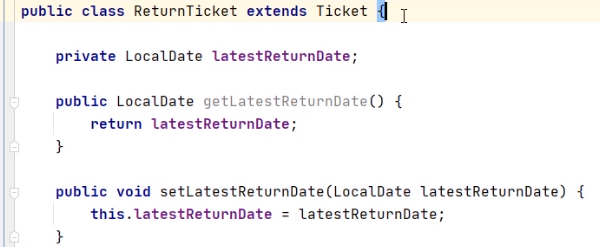
We made sure we changed the persistence unit and the database name.



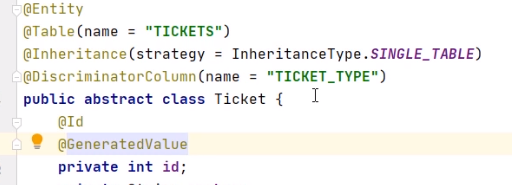
The project also contains the Ticket abstract class, which contains the ticket Id, the ticket number, and accessor methods for them.



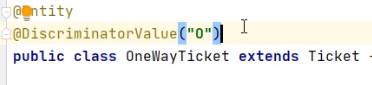
The OneWayTicket class extends the Ticket class and it adds the latestDepartureDate field. We also have the getter and the setter method for it.



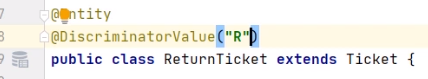
The ReturnTicket class extends the Ticket class, and it adds the latestReturnDate field. We also have the getter and the setter method for it.



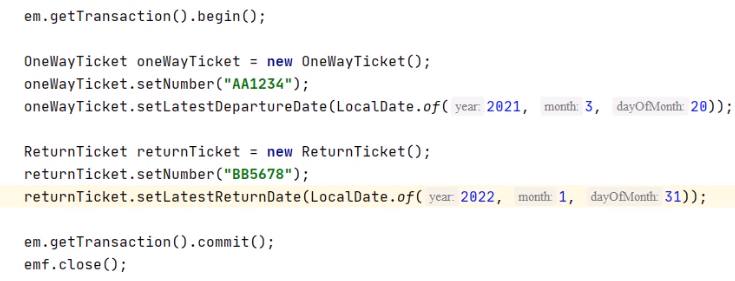
We move back to the Ticket class, and we annotate it with Entity, indicating that it has a corresponding table in the database, and we indicate the name of the table to be TICKETS. We'll used the Inheritance annotation to indicate the single table per class hierarchy strategy. We'll indicate the DiscriminatorColumn in the single table to be created to have the name TICKET\_TYPE. We annotate the Id field with the ID annotation, indicating that it is a primary key and with GeneratedValue, indicating that the value of this field is given by the database.



We annotate the OneWayTicket class with Entity, and we'll use the DiscriminatorValue annotation with the TICKET\_TYPE value for one‑way tickets to be O.



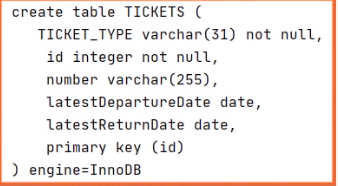
We annotate the ReturnTicket class with Entity, and we'll use the DiscriminatorValue annotation indicating the TICKET\_TYPE value for return tickets to be R.



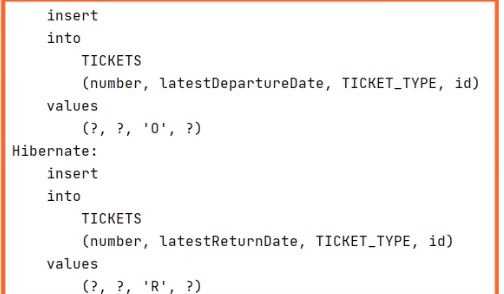
In the main class, we have already configured the persistence skeleton and the creation and the configuration of one OneWayTicket object and of one ReturnTicket object. The effective persistence part has to be added.



We need to persist one by one, the OneWayTicket and the ReturnTicket objects. We move on the site of the database, and here we have already created the M05\_EX03 database and commuted to it with the USE command. We may run the Java program. We'll create the needed table TICKETS, and we'll insert the information from our objects.



We notice the creation of a single TICKETS table containing the latestDepartureDate column specific to one‑way tickets and the latestReturnDate column specific to return tickets. The discriminator column TICKET\_TYPE is not null and will make the difference between one‑way tickets and return tickets.



We also noticed to insert commands, one with the latestDepartureDate field and the TICKET\_TYPE value equal to 0, one with the latestReturnDate field and the TICKET\_TYPE value equal to R. Let's check the content of the expected table.



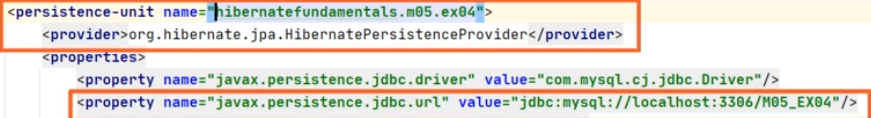
We execute SELECT \* FROM TICKETS.



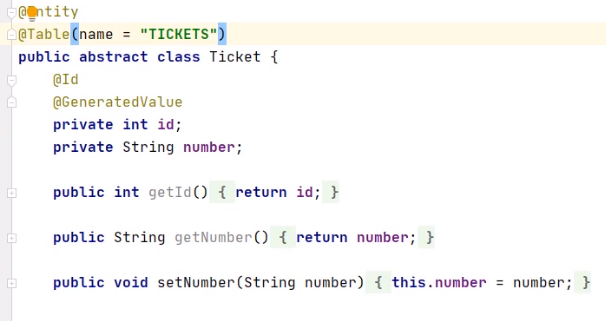
We expect the two tickets to be there. Yes, the AA1234 ticket is here with the value of the discriminator TICKET\_TYPE column O and using only the latestDepartureDate column. Also, the BB5678 ticket is here with the value of the discriminator TICKET\_TYPE column R and using only the latestReturnDate column. So this concludes our demonstration. We proved how we can create a single table for a class hierarchy with one super class and two subclasses representing tickets.

# Demo: Joined Subclass Strategy

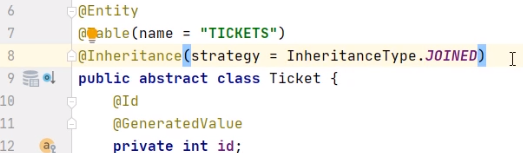
=>slides: Pg. 21

We move to the next practical demonstration. We'll implement an example in which the class hierarchy will be mapped using joined subclass strategy. 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. 

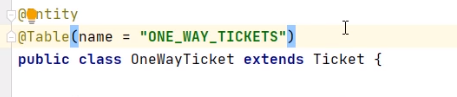
We made sure we changed the persistence‑unit and the database name.



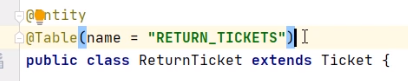
The project contains the Ticket abstract class, which contains the Ticket id, the Ticket number, and accessor methods for them. In addition to the previous demonstrations, we have already annotated the class with Entity and with Table, the name of the table being TICKETS. The id field is already annotated with Id and GeneratedValue.



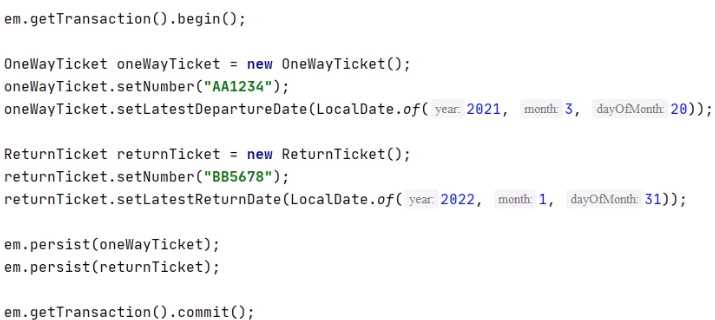
For this demonstration, we'll change the Inheritance strategy to be JOINED, indicating that the result will consist in tables between which we need to make joins.



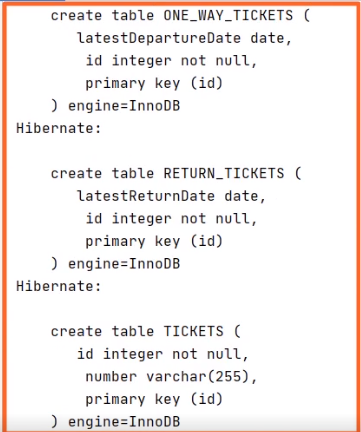
The ONE\_WAY\_TICKET class extends the Ticket class and it adds the latestDepartureDate field. We also have the getter and the setter method for it. We have already annotated the class with Entity and with Table, the name of the table being ONE\_WAY\_TICKETS.

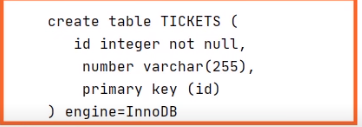


The ReturnTicket class extends the Ticket class and it adds the latestReturnDate field. We also have the getter and the setter method for it. We have already annotated the class with Entity and with Table, the name of the table being RETURN\_TICKETS.

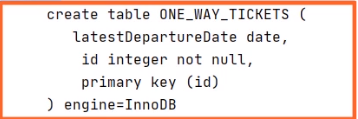


In the Main class, we have already configured the persistence skeleton, the creation and the configuration of one oneWayTicket object and of one returnTicket object. We have already made the persistence of the two objects. We move on the side of the database, and here we have already created the M05\_EX04 database and commuted to it with the USE command. And now we may run the Java program. We'll create the needed tables, TICKETS, ONE\_WAY\_TICKETS, and RETURN\_TICKETS. We'll alter these tables in order to make the join between them, and we'll insert the information from our objects.

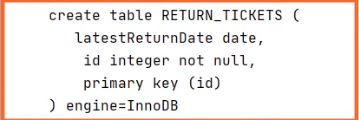


We notice the creation of three tables, TICKETS, ONE\_WAY\_TICKETS, and RETURN\_TICKETS. 

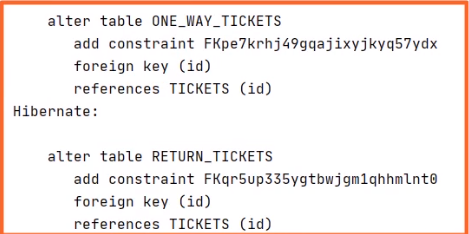
TICKETS contains the common id and number columns,



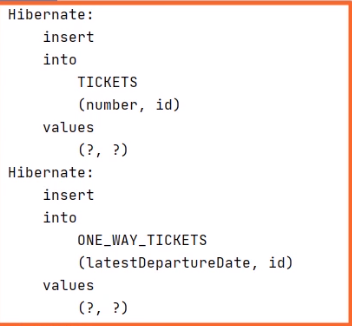
ONE\_WAY\_TICKETS contains the id joining column and the latestDepartureDate column specific to ONE\_WAY\_TICKETS,



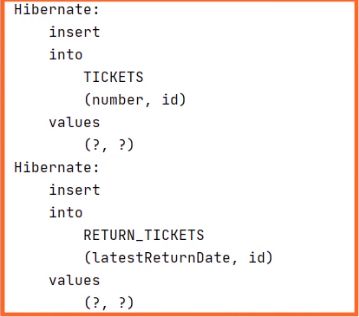
while RETURN\_TICKETS contains the id joining column and the latestReturnDate column specific to RETURN\_TICKETS.



We noticed two alter table commands that add the foreign key constraints between ONE\_WAY\_TICKETS and TICKETS, then between RETURN\_TICKETS and TICKETS.



Our first pair of insert commands will add the information from the ONE\_WAY\_TICKET with the common part in the TICKETS table and the specific part in the ONE\_WAY\_TICKETS table.



A second pair of insert commands will add the information from the RETURN\_TICKET with the common part in the TICKETS table and the specific part in the RETURN\_TICKETS table. Let's check the content of the expected tables. We execute SELECT \* FROM TICKETS.



We expect two rows to be found here, and they should contain the common parts of the tickets. And yes, we have a row with the id 1 and the number AA1234, and a row with the id 2 and the number BB5678. We execute SELECT \* FROM ONE\_WAY\_TICKETS.



We expect one row to be here, and it should contain the specific part of the ONE\_WAY\_TICKET. And, yes, we have a row with the id 1 and the latestDepartureDate. We execute SELECT \* FROM RETURN\_TICKETS.

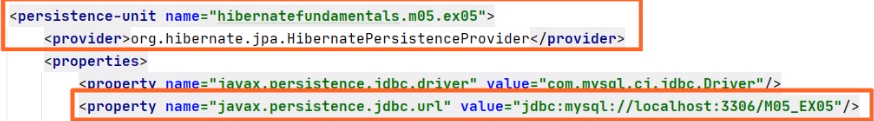


We expect one row to be here, and it should contain the specific part of the RETURN\_TICKET. And, yes, we have a row with the id 2 and the latestReturnDate. So, this concludes our demonstration. We proved how we can use the JOINED subclass strategy for a class hierarchy with one super class and two subclasses representing Tickets.

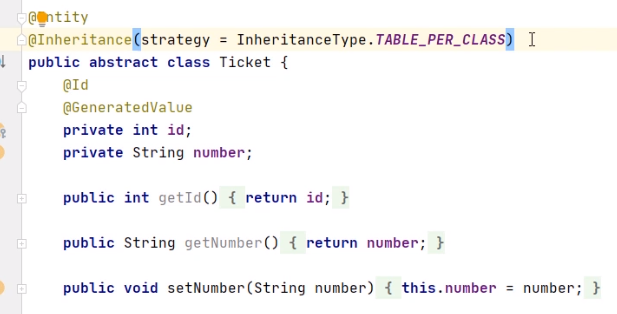
# Demo: Table per Concrete Class Strategy

=>slides: Pg. 22

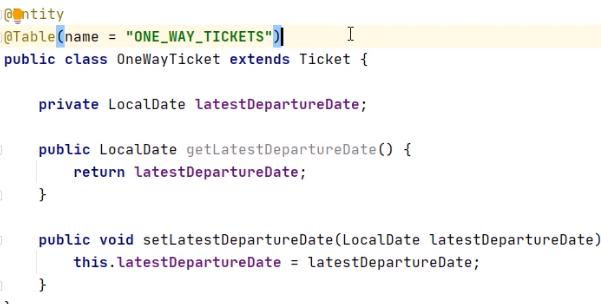
We move to another practical demonstration. We'll implement an example in which the class hierarchy will be mapped using table per concrete class strategy. We start with our Maven project, for which the pom.xml and the persistence.xml are configured similarly to what we presented in the previous demonstrations.



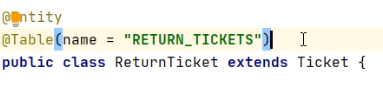
We made sure we changed the persistence‑unit and the database name.



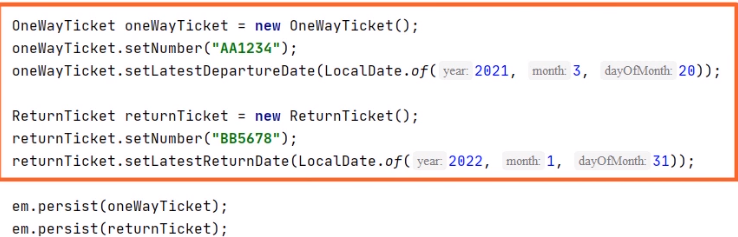
The project contains the Ticket abstract class, which contains the ticket id, the ticket number, and the accessor methods for them. We have already annotated the class with @Entity. The ID field is already annotated with @Id and @GeneratedValue. For this demonstration, we change the inheritance strategy to be TABLE\_PER\_CLASS, indicating that the result will consist in tables corresponding to each concrete class in the hierarchy.



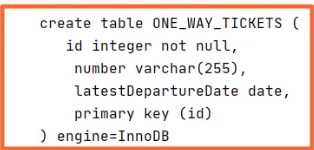
The OneWayTicket class extends the Ticket class and it adds the latestDepartureDate field. We also have the getter and the setter method for it. We have already annotated the class with @Entity and with @Table, the name of the table being ONE\_WAY\_TICKETS.



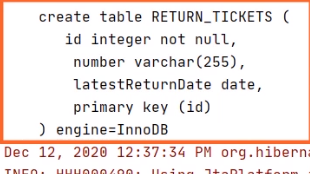
The ReturnTicket class extends the Ticket class, and it adds the latestReturnDate field. We also have the getter and the setter method for it. We have already annotated the class with @Entity and with @Table, the name of the table being RETURN\_TICKETS.



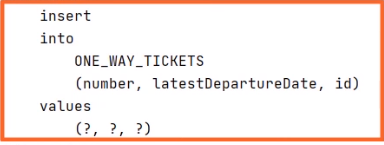
In the Main class, we have already configured the persistent skeleton, the creation and the configuration of one oneWayTicket object and of one returnTicket object. We have already made the persistence of the two objects. We move on the side of the database, and here, we have already created the M05\_EX05 database and commuted to it with the USE command. And now, we may run the Java program. We'll create the needed tables, ONE\_WAY\_TICKETS and RETURN\_TICKETS, and we'll insert the information from our objects.



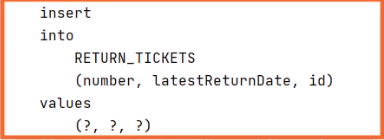
ONE\_WAY\_TICKETS contains the common id and number columns and the latestDepartureDate column, specific to ONE\_WAY\_TICKETS.



RETURN\_TICKETS contains the common id and number columns and the latestReturnDate column, specific to RETURN\_TICKETS.



A first insert command will add the information from the one‑way ticket with both the common part and the specific part in the ONE\_WAY\_TICKETS table.



A second insert command will add the information from the return ticket with both the common part and the specific part in the RETURN\_TICKETS table.



We execute SELECT \* FROM ONE\_WAY\_TICKETS.



We expect one row to be here, and it should contain both the common and the specific part of the one‑way ticket. And yes, we have a row with the id 1 number AA1234 and the latestDepartureDate. We execute SELECT \* FROM RETURN\_TICKETS.



We expect one row to be here, and it should contain both the common and the specific part of the return ticket. And yes, we have a row with the id 2 number BB5678 and the latestReturnDate. So, this concludes our demonstration. We proved how we can use the table for concrete class strategy using a class hierarchy with one superclass and two subclasses representing Tickets.

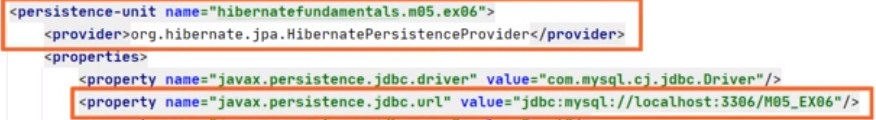
# Demo: Conversion

=>slides: Pg. 23

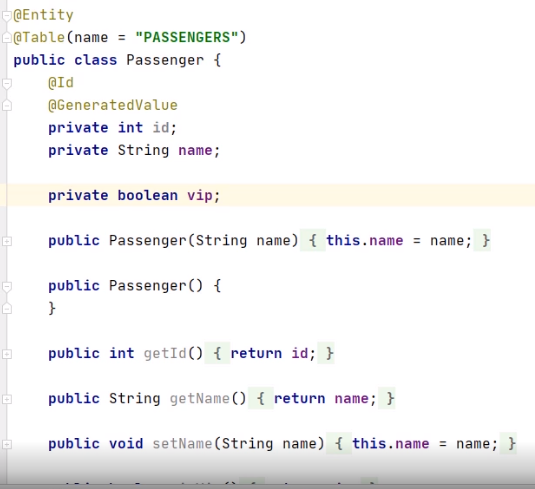
We'll end the demonstrations implementing the conversion between different data representations in Java and in the database.



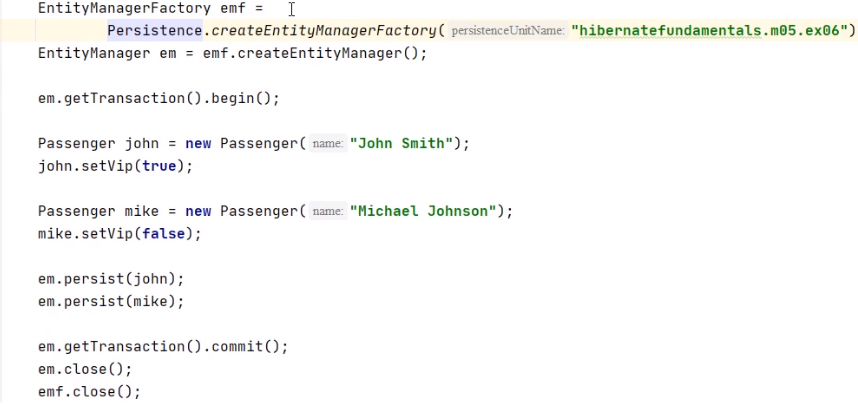
We'll start with our Maven project, for which the pom.xml and the persistence.xml are configured similarly to what we presented in the previous demonstrations.



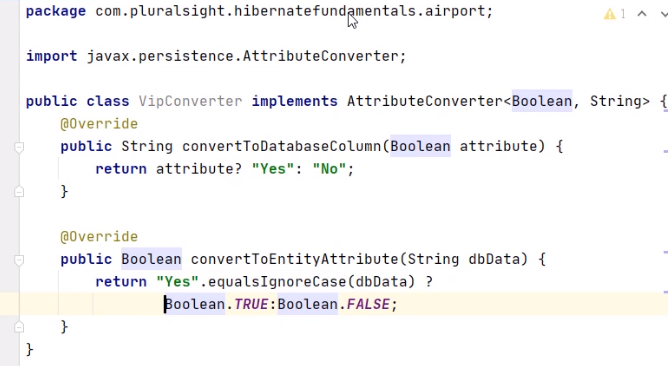
We made sure we changed the persistence‑unit and the database name.



The project contains the Passenger class, which contains the id, the name, the boolean vip, and the accessor methods for them. The vip field is boolean in the class, but it will be kept in string yes/no format in the database. We have already annotated the class with entity, the corresponding table is PASSENGERS. The id field is already annotated with @Id and @GeneratedValue.



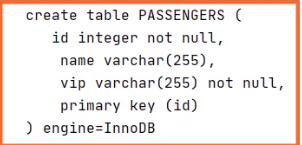
In the Main class, we have already configured the persistent skeleton, the creation and the configuration of two Passenger objects. We have already made the persistence of the two objects. We need to create the converter class that will convert from the boolean object representation, the string database representation.



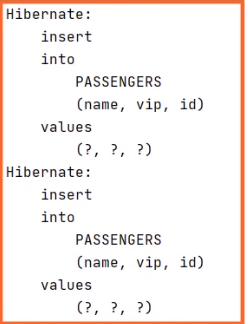
We call this class VIPConverter. We'll make this VIPConverter implement the AttributeConverter interface. This interface is generally fired by Boolean and String, meaning that it will convert from a boolean to a string and vice versa. Implementing this interface, we need to implement the two methods that it declares, convertToDatabaseColumn and convertToEntityAttribute. The convertToDatabaseColumn method will convert the boolean object representation to the string database representation. If the Boolean argument is true, it will return Yes, otherwise it will return No. And we'll rename the aBoolean argument of the method to attribute, as it represents an attribute of the object. The convertToEntityAttribute method will convert the string database representation to the boolean object representation. We rename the argument of the method to dbData. If the argument is Yes, no matter the case, we'll return Boolean.TRUE. Otherwise, we'll return Boolean.FALSE.



We go back to the Passenger class, and we'll annotate the boolean vip field with @Convert argument converter = VipConverter.class, indicating that it will be this converter to be used for the read and write operations from and to the database. We move on the side of the database, and here, we have already created the M05\_EX06 database and commuted to it with the USE command. And now, we may run the Java program. We'll create the needed table, PASSENGERS, and we'll insert the information from our objects using the VipConverter that we have just created.



The PASSENGERS table contains the ID and the name column, while the VIP column, corresponding to a boolean field is of type varchar of 255 in the database.



We also have to insert commands for the two passengers that we persist in the database. We execute SELECT \* FROM PASSENGERS. We expect two rows to be here, and they should contain the two previously persisted passengers.



And yes, we have a row with the id 1 named John Smith and the vip value being Yes, as it was a Boolean.TRUE value in the program. We also have a row with the id 2 named Michael Johnson and the vip value being No, as it was a Boolean.FALSE value in the program. So, this concludes our demonstration. We proved how we can use converters between different values in the program and in the database, and we created our own VipConverter class to transform a boolean to a string and vice versa.

=>slides: Pg. 24

At the end of this module, let's have a look at what we found out here. We built class hierarchies and inherited both from entities and from non‑entities. We applied the mapping strategies, more exactly the single table per class hierarchy, the joined subclass strategy, and the table per concrete class strategy, analyzing the particularities, the pluses, and the minuses of each one. We worked with converters for converting data representation that is different from the Java class to the database record. Our next module will work in depth with the entity manager. See you there.

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