First steps and detailed concepts

Java Persistence API Mini Book

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Hebert Coelho
Byron Kiourtzoglou

Table of Contents

Reasons that led to the creation of JPA

One of the problems of Object Orientation is how to map the objects as the database requires. It is possible to have a class with the name Car but its data is persisted in a table named TB_CAR. The table name is just the beginning of the problem, what if the Car class has the attribute "*name*" but in the database you find the column STR_NAME_CAR?

The basic Java framework to access the database is JDBC. Unfortunately, with JDBC, a lot of hand work is needed to convert a database query result into Java classes.

In the code snippet bellow we demonstrate how to transform a JDBC query result into Car objects:

```
import java.sql.*;


import java.util.LinkedList;
import java.util.List;

public class MainWithJDBC {
    public static void main(String[] args) throws Exception {
        Class.forName("org.hsqldb.jdbcDriver");

        Connection connection = // get a valid connection

        Statement statement = connection.createStatement();

        ResultSet rs = statement.executeQuery("SELECT \"Id\", \"Name\" FROM \"Car\"");

        List<Car> cars = new LinkedList<Car>();

        while(rs.next()){
            Car car = new Car();
            car.setId(rs.getInt("Id"));
            car.setName(rs.getString("Name"));
            cars.add(car);
        }

        for (Car car : cars) {
            System.out.println("Car id: " + car.getId() + " Car Name: " + car.getName());
        }

        connection.close();
    }
}
```

As you can understand there is a lot of boilerplate code involved using this approach. Just imagine the Car class the have not two but thirty attributes… To make things even worse, imagine a class with 30 attributes and with a relationship to another class that has 30 attributes also e.g. a Car class may maintain a list of Person classes

representing the drivers of the specific car where the Person class may have 30 attributes to be populated.

Other disadvantages of JDBC is its portability. The query syntax will change from one database to another. For example with Oracle database the command ROWNUM is used to limit the amount of returned lines, whereas with SqlServer the command is TOP.

Application portability is a problematic issue when database native queries are used. There are several solutions to this kind of problem, e.g. a separate file with all query code could be stored outside the application deployable file. With this approach for each database vendor a sql specific file would be required.

It is possible to find other problems when developing with JDBC like: update a database table and its relationships, do not leave any orphans records or an easy way to use hierarchy.

What is JPA? What is a JPA Implementation?

JPA was created as a solution to the problems mentioned earlier.

JPA allows us to work with Java classes as it provides a transparent layer to each database specific details; JPA will do the hard work of mapping table to class structure and semantics for the developer.

An easy definition to JPA is: “A group of specifications (a lot of texts, regularizations and Java Interfaces) to define how a JPA implementation should behave”. There are many JPA implementations available both free and paid, e.g. Hibernate, OpenJPA, EclipseLink and the “new born” Batoo etc.

Most JPA implementations are free to add extra codes, annotations that are not present in the JPA specification, but must conform to the JPA specification scemantics.

The main JPA feature to address application portability is the ability to map database tables into the classes. In the following sections we will demonstrate how it is possible to map the column of a table into a Java class field regardless of the names of both the batabase column and Java class field.

JPA created a database language named JPQL to perform database queries. The advantage of JPQL is that the query can be executed in all databases.

```
SELECT id, name, color, age, doors FROM Car
```

The query above could be translated to the JPQL below:

```
SELECT c FROM Car c
```

Notice that the result of the query above is “c”, that means, a car object and not the fields/values found in the database table. JPA will create the object automatically.

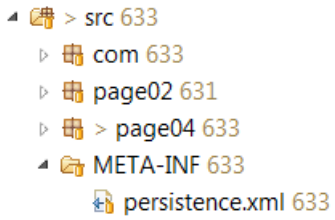
If you want to see several ways to run a database query with JPA click here.
JPA will be responsible to translate the JPQL query to the database native query; the developer will not need to worry about which is the required sql database syntax.

What is the persistence.xml file used for? And its configurations?

The persistence.xml file is responsible for all JPA evironment configuration; it can hold database, application and JPA implementation specific configuration.

In the persistence.xml code presented here we use the EclipseLink JPA implementation specific configuration, but the concept and the theory applied to the Java classes in this post can be applied when using any JPA implementation framewrok available.

The persistence.xml file must be located in a META-INF folder in the same path as the Java classes. Below is a picture that shows where the file should be placed:



This picture is valid when using Eclipse IDE. For Netbeans it is necessary to check its documentation as of where the correct place to put the file is.

If you got the error message: “Could not find any META-INF/persistence.xml file in the classpath” here are some tips that you can check:

- Open the generated WAR file and check if the persistence.xml is located at “/WEB-INF/classes/META-INF/”. If the file is automatically generated and is not there your error is at the WAR creation, you need to put the persistence.xml file where the IDE expects. If the artifact

creation is done manually check the creation script (ant, maven).
•If your project is deployed with EAR file check if the persistence.xml file is in the EJB jar root. If the file is automatically generated and is not there your error is at the WAR creation, you need to put the persistence.xml file where the IDE expects. If the artifact creation is manually check the creation script (ant, maven).
•If your project is a JSE project (desktop) you should check if the file is not in the folder “META-INF”. As default behavior the JPA will search in the JAR root for the folder and file: “/META-INF/persistence.xml”.
•Check if the file has the name: “persistence.xml”. The file must have the name with all letters in lowercase.

If JPA cannot find the file the error above will be displayed. As a general rule, it is a good practice to put the file as displayed above.

Check below a sample of persistence.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="MyPU" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

        <class>page20.Person</class>
        <class>page20.Cellular</class>
        <class>page20.Call</class>
        <class>page20.Dog</class>

        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:myDataBase" />
            <property name="javax.persistence.jdbc.user" value="sa" />
            <property name="javax.persistence.jdbc.password" value="" />
            <property name="eclipselink.ddl-generation" value="create-tables" />
            <property name="eclipselink.logging.level" value="FINEST" />
        </properties>
    </persistence-unit>

    <persistence-unit name="PostgresPU" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

        <class>page26.Car</class>
        <class>page26.Dog</class>
```

```
        <class>page26.Person</class>

        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost/JpaRelationships"
/>
            <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
            <property name="javax.persistence.jdbc.user" value="postgres" />
            <property name="javax.persistence.jdbc.password" value="postgres" />
            <!-- <property name="eclipselink.ddl-generation" value="drop-and-create-tables" /> -->
            <property name="eclipselink.ddl-generation" value="create-tables" />
            <!-- <property name="eclipselink.logging.level" value="FINEST" /> -->
        </properties>
    </persistence-unit>
</persistence>
```

About the code above:

•<persistence-unit name=”MyPU” => with this configuration it is possible to define the Persistence Unit name. The Persistence Unit can be understood as the JPA universe of your application. It contains information about all classes, relationships, keys and others configurations all relating the database to your application. It is possible to add more than one Persistence Unit at the same persistence.xml file as displayed in the code above.
•transaction-type=”RESOURCE_LOCAL” => Define the transaction type. Two values are allowed here: RESOURCE_LOCAL and JTA. For desktop applications RESOURCE_LOCAL should be use; for web application both values can be used, the correct value depends on the application design.
•<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider> => Defines the JPA implementation provider. The provider is the application JPA implementation. If your application is using Hibernate, the provider value should be “org.hibernate.ejb.HibernatePersistence” and for OpenJPA “org.apache.openjpa.persistence.PersistenceProviderImpl”.
•<class></class> => Used to declare the Java classes. In a JEE/JSE usually this is not required; e.g. for a Hibernate desktop application there is no need to declare the classes with this tags, but with EclipseLink and OpenJPA requires the presence of the class tag.
•<exclude-unlisted-classes>true</exclude-unlisted-classes> => this configurations defines that if a class is not listed in the persistence.xml it should not be handled as an entity (we will see more about entity in the next page) in the Persistence Unit. This configuration is very useful for applications with more than one Persistence Units, where an Entity should appear in one database but not in the other.
•It is possible to comment a code use the tag <!– –>
•<properties> => It is possible to add specific JPA implementations configurations. Values like driver, password and user it is normal to find it to all implementations; usually these values are written in the persistence.xml file for a RESOUCE_LOCAL application. To JTA applications the datasources are used by the containers. A datasource can be specified with the tags <jta-data-

source></jta-data-source><non-jta-data-source></non-jta-data-source>. Some JBoss versions require the data source declaration to be present even if the connection is local. Two configurations are worth mentioning:

Configurations	Implementation	Used for
eclipselink.ddl-generation	EclipseLink	Will activate the automatically database table creation, or just validate the database schema against the JPA configuration.
hibernate.hbm2ddl.auto	Hibernate	
openjpa.jdbc.SynchronizeMappings	OpenJPA	
eclipselink.logging.level	EclipseLink	Will define the LOG level of the JPA implementation.
org.hibernate.SQL.level=FINEST org.hibernate.type.level=FINEST	Hibernate	
openjpa.Log	OpenJPA	
On the internet you will find the allowed values to each implementation.		

Entity definitions. What are the Logic and Physical annotations?

For JPA to correctly map database tables into Java classes the concept of Entity was created. An Entity must be created to support the same database table structure; thus JPA handles table data modification through an entity.

For a Java class to be considerate an Entity it must follow the rules below:

- To be annotated with @Entity
- A public constructor without arguments
- The Java class will need to have a field with the @Id annotation

The class below follows all the requirements to be considerate an entity:

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Car {
```

```
    public Car(){

    }

    public Car(int id){
        this.id = id;
    }

    // Just to show that there is no need to have get/set when we talk about JPA Id
    @Id
    private int id;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

About the code above:

- The class has the annotation @Entity above its name
- There is an attribute considered the id of the class that is annotated with @Id. *Every entity must have an id*. Usually this field is a sequential integer field, but it can be a String or other allowed values
- Notice that there is no get/set for the id attribute. For JPA is concidered that an entity's id is immutable, so there is no need to edit the id value.
- The presence of a public constructor without arguments is mandatory. Other constructors may be added

As shown in the code snippet above we only used two annotations, one to define an entity and another to declare the "id" field; by default JPA will look for a table named CAR in the database with columns named ID and NAME. By default JPA will use the class name and the class attribute names to find the tables and its structures.

According to the book “Pro JPA 2” we can define the JPA annotations in two ways: Logical annotations and Physical annotations. The physical annotations will map the configuration of the database into the class. The logical annotations will define the application modeling. Check the code below:

```
import java.util.List;

import javax.persistence.*;

@Entity
@Table(name = "TB_PERSON_02837")
```

```
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    @Basic(fetch = FetchType.LAZY)
    @Column(name = "PERSON_NAME", length = 100, unique = true, nullable = false)
    private String name;

    @OneToMany
    private List<Car> cars;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

In the code snippet above the following annotations are shown: @Entity, @Id and @OneToMany (we will see about this annotation later); these are considered logical annotations. Notice that these annotations do not define anything related to database, but define how a Java class will behave as an Entity.

Additionally in the code above we demonstrate another set of annotations: @Table, @Column and @Basic. These annotations will create the relationship between the database table and the entity. It is possible to define the table name, column name and other information regarding the database. This kind of annotations are known as Physical annotations, it is their job to “connect” the database to the JPA code.

Its out of the scope of this mini book to present every annotation supported by JPA, but it is very easy to find this information on the internet; e.g.: *@Column(name = “PERSON_NAME”, length = 100, unique = true, nullable = false)*. The physical annotations are the easier to understand because it looks like database configuration.

Id Generation: Definition, using Identity or Sequence

As said in previous chapters, every entity must have an id. JPA has the option to automatically generate the entity id.

There are three options for automatic id generation:

- Identity
- Sequence
- TableGenerator

We must have in mind that every database has its own id generation mechanism. Oracle and Postgres databases use the Sequence approach, SqlServer and MySQL use the Identity approach. It is not possible to use an id generation approach in a server when the server does not support it.

The following Java types are allowed to be used for an id attribute: byte/Byte, int/Integer, short/Short, long/Long, char/Character, String, BigInteger, java.util.Date and java.sql.Date.

Identity

This is the simplest id generation approach. Just annotate the id field like below:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

The database controls the ID generation, JPA does not act on the id at all. Thus in order to retrieve the id from the databse an entity needs to be persisted first, and after the transaction commits, a query is executed so as to retrieve the generated id for the specific entity. This procedure leads to a small performance degradation, but not something troublesome.

The aforementioned id generation approach does not allow allocating ids in memory. We will examine what this ids allocation means in the following chapters.

Sequence

The Sequence approach can be configured like below:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;

@Entity
@SequenceGenerator(name = Car.CAR_SEQUENCE_NAME, sequenceName = Car.CAR_SEQUENCE_NAME, initialValue = 10,
allocationSize = 53)
public class Car {

    public static final String CAR_SEQUENCE_NAME = "CAR_SEQUENCE_ID";

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = CAR_SEQUENCE_NAME)
    private int id;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

About the code above:

- The @SequenceGenerator annotation will define that there will be a sequence in the database with the specified name (sequenceName attribute). The same sequence could be shared between entities but is not recomended. If we use the Car sequence in the House entity, the id values in the tables would not be sequential; when the first car is persisted the id would be 1, if another car is persisted the id would be 2; when the first house is persisted its id would be 3, this will happen if the House entity uses the same sequence as the Car entity.
- The attribute "name = Car.CAR_SEQUENCE_NAME" defines the name of the sequence inside the application. Sequence declaration should be done only once. All Entities that will use the same sequence will just have to use the specific sequence name. That is why we assign the sequence name value to a static attribute at the specific entity.
- The value "sequenceName = Car.CAR_SEQUENCE_NAME" reflects the sequence name at the database level.
- The "initialValue = 10" defines the first id value of the sequence. A developer must be cautious

with this configuration; if after inserting the first row the application is restarted, JPA will try to insert a new entity with the same value defined in the initialValue attribute. An error message will be displayed indicating that the id is already in use.
• "allocationSize = 53" represents the amount of ids that JPA will store in cache. Works like this: when the application is started JPA will allocate in memory the specified number of ids and will share these values to each new persisted entity. In the code above, the ids would go from 10 (initialValue) up to 63 (initialValue + allocationSize). When the number of allocated ids ends JPA will request from the database and allocate in memory 53 more ids. This act of allocating ids into memory is a good approach to optimize server memory, since JPA will not need to trigger the database with every insert so as to get the created id just like with the @Identity approach.
•The @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = CAR_SEQUENCE_NAME) annotation defines that the generation type as SEQUENCE and the name of the generator.

Id Generation: TableGenerator and Auto TableGenerator

The text below describes how the TableGenerator works:

•A table is used to store the id values
•This table has a column that will store the table name and the actual id value
•So far, this is the only Generation ID approach that allows database portability, without the need of altering the id generation approach. Imagine an application that is running with Postgres and uses a Sequence. If we were to use the same application with SqlServer too we would have to create two distinct artifacts (WAR/JAR/EAR) of it, one for each database – since SqlServer doen not support Sequences. With the TableGenerator approach the same artifact can be used for both databases.

Check the code below to see how the TableGenerator approach can be used:

```
import javax.persistence.*;

@Entity
public class Person {

    @Id
    @TableGenerator(name="TABLE_GENERATOR", table="ID_TABLE", pkColumnName="ID_TABLE_NAME",
pkColumnValue="PERSON_ID", valueColumnName="ID_TABLE_VALUE")
    @GeneratedValue(strategy = GenerationType.TABLE, generator="TABLE_GENERATOR")
```

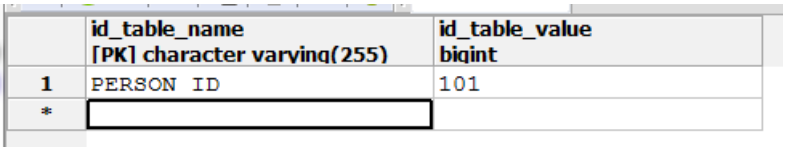
```
    private int id;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

The code above will use the table in the image below (it is possible to configure JPA to create the specific table automatically, just add the needed configuration as presented in the section about persistence.xml):



	id_table_name [PK] character varying(255)	id_table_value bigint
1	PERSON_ID	101
*		

About the code above:

- "name" => Is the TableGenerator id inside the application.
- "table" => Name of the table that will hold the values.
- "pkColumnName" => Column name that will hold the id name. In the code above the column name "id_table_name" will be used.
- "valueColumnName" => Column name that will hold the id value.
- "pkColumnValue" => Table name that will persist in the table. The default value is the entity name + id (entity attribute defined as id). In the code above will be PERSON_ID that is the same value described above.
- initialValue, allocationSize => These options can be used in the @TableGenerator annotation also. Check the Sequence approach section above to see how they should be used.
- It is possible to declare TableGenerator across different entities without the problem of loosing sequenced ids (like the Sequence problem that we saw earlyer).

The best practice to use the table generator declartion is in an orm.xml file. This file is used to override JPA configuration through annotations and is out of the scope of this mini book.

Auto

The Auto approach (automatically) allows JPA to choose an approach to use. This is the default value and can be used like below:

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO) // or just @GeneratedValue
private int id;
```

With the Auto approach any strategy can be used by JPA. JPA will choose between the 3 approaches discussed earlyer.

Simple Composite Key

A simple key is when the id uses just one field. A simple key is used like below:

```
@Id
private int id;
```

A composite key is needed when more than one attribute is required as an entity id. It is possible to find simple composite key and complex composite key. With a Simple composite key use only plain Java attributes for the id (e.g String, int, …). In the following sections we will disquas all about complex composite key semantics.

There are two ways to map a simple composite key, with @IdClass or @EmbeddedId.

@IdClass

Check the code below:

```
import javax.persistence.*;

@Entity
@IdClass(CarId.class)
public class Car {

    @Id
    private int serial;

    @Id
    private String brand;
```

```
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getSerial() {
        return serial;
    }

    public void setSerial(int serial) {
        this.serial = serial;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }
}
```

About the code above;

- @IdClass(CarId.class) => this annotation indicates that the CarId has inside of it the id attributes found in the Car entity.
- All fields annotated with @Id must be found in the IdClass.
- It is possible to use the @GeneratedValue annotation with this kind of composite key. For example in the code above it would be possible to use the @GeneratedValue with the "serial" attribute.

Check below the CarId class code:

```
import java.io.Serializable;

public class CarId implements Serializable{

    private static final long serialVersionUID = 343L;

    private int serial;
    private String brand;

    // must have a default construcot
    public CarId() {

    }
```

```
    public CarId(int serial, String brand) {
        this.serial = serial;
        this.brand = brand;
    }

    public int getSerial() {
        return serial;
    }

    public String getBrand() {
        return brand;
    }

    // Must have a hashCode method
    @Override
    public int hashCode() {
        return serial + brand.hashCode();
    }

    // Must have an equals method
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof CarId) {
            CarId carId = (CarId) obj;
            return carId.serial == this.serial && carId.brand.equals(this.brand);
        }

        return false;
    }
}
```

The class CarId has the fields listed as @Id in the Car entity.

To use a class as ID it must follow the rules below:

- A public constructor with no arguments must be found
- Implements the Serializable interface
- Overwrite the hashCode/equals method

To do a query against a database to find an entity given a simple composite key just do like below:

```
EntityManager em = // get valid entity manager

CarId carId = new CarId(33, "Ford");

Car persistedCar = em.find(Car.class, carId);

System.out.println(persistedCar.getName() + " - " + persistedCar.getSerial());
```

To use the find method it is necessary to provide the id class with the required information.

@Embeddable

The other approach to use the composite key is presented below:

```
import javax.persistence.*;

@Entity
public class Car {

    @EmbeddedId
    private CarId carId;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public CarId getCarId() {
        return carId;
    }

    public void setCarId(CarId carId) {
        this.carId = carId;
    }
}
```

About the code above:

- The id class was written inside the Car class.
- The @EmbeddedId is used to define the class as an id class.
- There is no need to use the @Id annotation anymore.

The class id will be like below:

```
import java.io.Serializable;

import javax.persistence.Embeddable;

@Embeddable
public class CarId implements Serializable{

    private static final long serialVersionUID = 343L;

    private int serial;
    private String brand;
```

```
    // must have a default construcot
    public CarId() {

    }

    public CarId(int serial, String brand) {
        this.serial = serial;
        this.brand = brand;
    }

    public int getSerial() {
        return serial;
    }

    public String getBrand() {
        return brand;
    }

    // Must have a hashCode method
    @Override
    public int hashCode() {
        return serial + brand.hashCode();
    }

    // Must have an equals method
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof CarId) {
            CarId carId = (CarId) obj;
            return carId.serial == this.serial && carId.brand.equals(this.brand);
        }

        return false;
    }
}
```

About the code above:

- The @Embeddable annotation allows the class to be used as id.
- The fields inside the class will be used as ids.

To use a class as ID it must follow the rules below:

- A public constructor with no arguments must be found
- Implements the Serializable interface
- Overwrite the hashCode/equals method

It is possible to do queries with this kind of composite key just like the @IdClass presented above.

Complex Composite Key

A complex composite key is composed of other entities – not plain Java attributes.

Imagine an entity DogHouse where it uses the Dog as the id. Take a look at the code below:

```
import javax.persistence.*;

@Entity
public class Dog {
    @Id
    private int id;

    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
import javax.persistence.*;

@Entity
public class DogHouse {

    @Id
    @OneToOne
    @JoinColumn(name = "DOG_ID")
    private Dog dog;

    private String brand;

    public Dog getDog() {
        return dog;
    }

    public void setDog(Dog dog) {
        this.dog = dog;
    }

    public String getBrand() {
```

```
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }
}
```

About the code above:

- The @Id annotation is used in the entity DogHouse to inform JPA that the DogHouse will have the same id as the Dog.
- We can combine the @Id annotation with the @OneToOne annotation to dictate that there is an explicit relationship between the classes. More information about the @OneToOne annotation will be available later on.

Imagine a scenario where it is required to access the DogHouse id without passing through the class Dog (dogHouse.getDog().getId()). JPA has a way of doing it without the need of the Demeter Law pattern:

```
import javax.persistence.*;

@Entity
public class DogHouseB {

    @Id
    private int dogId;

    @MapsId
    @OneToOne
    @JoinColumn(name = "DOG_ID")
    private Dog dog;

    private String brand;

    public Dog getDog() {
        return dog;
    }

    public void setDog(Dog dog) {
        this.dog = dog;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public int getDogId() {
        return dogId;
```

```
    }

    public void setDogId(int dogId) {
        this.dogId = dogId;
    }
}
```

About the code above:

- There is an explicit field mapped with @Id.
- The Dog entity is mapped with the @MapsId annotation. This annotation indicates that JPA will use the Dog.Id as the DogHouse.DogId (just like before); the dogId attribute will have the same value as the dog.getId() and this value will be attributed at run time.
- dogId field does not need to be explicitely mapped to a database table column. When the application starts JPA will attribute the dog.getId() to the dogId attribute.

To finish this subject, let us see one more topic. How can we map an entity id with more than one entities?

Check the code below:

```
import javax.persistence.*;

@Entity
@IdClass(DogHouseId.class)
public class DogHouse {

    @Id
    @OneToOne
    @JoinColumn(name = "DOG_ID")
    private Dog dog;

    @Id
    @OneToOne
    @JoinColumn(name = "PERSON_ID")
    private Person person;

    private String brand;

    // get and set
}
```

About the code above:

- Notice that both entities (Dog and Person) were annotated with @Id.
- The annotation @IdClass is used to indicate the need of a class to map the id.

```
import java.io.Serializable;

public class DogHouseId implements Serializable{
```

```
    private static final long serialVersionUID = 1L;

    private int person;
    private int dog;

    public int getPerson() {
        return person;
    }

    public void setPerson(int person) {
        this.person = person;
    }

    public int getDog() {
        return dog;
    }

    public void setDog(int dog) {
        this.dog = dog;
    }

    @Override
    public int hashCode() {
        return person + dog;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof DogHouseId){
            DogHouseId dogHouseId = (DogHouseId) obj;
            return dogHouseId.dog == dog && dogHouseId.person == person;
        }

        return false;
    }
}
```

About the code above:

- The class has the same amount of attributes as the number of attributes in the class DogHouse annotated with @Id
- Notice that the attributes inside the DogHouseId have the same name as the attributes inside the DogHouse annotated with @Id. This is mandatory for JPA to correctly utilize id functionality. For example If we named the Person type attribute inside the DogHouse class as "dogHousePerson" the name of the Person type attribute inside the DogHouseId class would have to change also to "dogHousePerson".

To use a class as ID it must follow the rules below:

- A public constructor with no arguments must be found

- Implements the Serializable interface
- Overwrite the hashCode/equals method

How to get an EntityManager

There are two ways to get an EntityManager. One is with injection and the other through a factory.

The easiest way to get an EntityManager is with injection, the container will inject the EntityManager. Below is how the injection code works:

```
@PersistenceContext(unitName = "PERSISTENCE_UNIT_MAPPED_IN_THE_PERSISTENCE_XML")
private EntityManager entityManager;
```

It is needed to annotate the EntityManager field with: “@PersistenceContext(unitName = “PERSISTENCE_UNIT_MAPPED_IN_THE_PERSISTENCE_XML”)”. The injection option will only work for JEE applications, running inside applications servers like JBoss, Glassfish… To achieve the injection without problems the persistence.xml must be in the right place, and must have (if needed) a datasource defined.

The EntityManager injection, until today, will work only with a server that supports an EJB container. Tomcat and other WEB/Servlet only containers will not inject it.

When the application is a JSE (desktop) application or when a web application wants to handle the database connection manually just use the code bellow:

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("PERSISTENCE_UNIT_MAPPED_IN_THE_PERSISTENCE_XML");
EntityManager entityManager = emf.createEntityManager();

entityManager.getTransaction().begin();

// do something

entityManager.getTransaction().commit();
entityManager.close();
```

Notice that it is required to get an instance of the EntityManagerFactory first, which will be linked to a PersistenceUnit created in the persistence.xml file. Through the EntityManagerFactory is possible to get an instance of the EntityManager.

Mapping two or more tables in one entity

A class may have information in more than one table.

To map an Entity that has its data in more than one table, just do like below:

```
import javax.persistence.*;

@Entity
@Table(name="DOG")
@SecondaryTables({
    @SecondaryTable(name="DOG_SECONDARY_A", pkJoinColumns={@PrimaryKeyJoinColumn(name="DOG_ID")}),
    @SecondaryTable(name="DOG_SECONDARY_B", pkJoinColumns={@PrimaryKeyJoinColumn(name="DOG_ID")})
})
public class Dog {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String name;
    private int age;
    private double weight;

    // get and set

}
```

About the code above:

- The annotation @SecundaryTable is used to indicate at which table the Entity data can be found. If the data is found in just one secondary table just the @SecundaryTable annotation is enough.
- The annotation @SecundaryTables is used to group up several tables in one class. It groups several @SecundaryTable annotations.

Mapping Hierarchy: MappedSuperclass

Sometimes is needed to share methods/attributes between classes; thus create a class hierarchy. In case the super class – the one with the shared methods/attributes - is not an entity, it is required to be marked as MappedSuperclass.

Check below how the MappedSuperclass concept can be applied:

```
import javax.persistence.MappedSuperclass;

@MappedSuperclass
```

```
public abstract class DogFather {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

```
@Entity
@Table(name = "DOG")
public class Dog extends DogFather {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String color;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

About the code above:

- The DogFather class is annotated with the @MappedSuperclass annotation. With this annotation all DogFather child classes will have its attributes persisted in the database, but DogFather will not be mapped to a database table.
- A MappedSuperclass can be an abstract or a concrete class.
- The Dog class has the ID generator, only Dog is an entity. DogFather is not an entity.

Some recommendations about MappedSuperclass:

- A MappedSuperclass cannot be annotated with @Entity\@Table. It is not a class that will be

persisted. Its attributes/methods will be reflected in its child classes.
•It is always a good practice to create it as abstract. A MappedSuperclass will not be used in the queries.
•Cannot be persisted, it is not an entity.

When do we use it?

If we do not need to use the super class in database queries, it would be a good idea to use a MappedSuperclass. In the opposite case it is a good idea to use Entity hierarchy (see the following sections for details).

Mapping Hierarchy: Single Table

With JPA it is possible to find different approaches for persisting class hierarchies. In an Object Orientated language like Java it is very easy to find hierarchy between classes that will have their data persisted.

The Single Table strategy will store all hierarchy data into one table. Check the code below:

```
import javax.persistence.*;

@Entity
@Table(name = "DOG")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DOG_CLASS_NAME")
public abstract class Dog {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String name;

    // get and set
}
```

```
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("SMALL_DOG")
public class SmallDog extends Dog {
    private String littleBark;

    public String getLittleBark() {
        return littleBark;
    }

    public void setLittleBark(String littleBark) {
        this.littleBark = littleBark;
    }
```

```
}
```

```
import javax.persistence.*;

@Entity
@DiscriminatorValue("HUGE_DOG")
public class HugeDog extends Dog {
    private int hugePooWeight;

    public int getHugePooWeight() {
        return hugePooWeight;
    }

    public void setHugePooWeight(int hugePooWeight) {
        this.hugePooWeight = hugePooWeight;
    }
}
```

About the code above:

- @Inheritance(strategy = InheritanceType.SINGLE_TABLE) => this annotation should be placed in the highest place of the hierarchy (the “father” class), also known as “root”. This annotation will define the hierarch pattern to be followed, in the annotation above the hierarchy mapping strategy is set to Single Table.
- @DiscriminatorColumn(name = “DOG_CLASS_NAME”) => will define the name of the column that will link a database table row to a class. Check in the image below how the data is stored.
- @DiscriminatorValue => Will set the value to be persisted in the column defined in the annotation @DiscriminatorColumn. Check in the image below how the data is stored.
- Notice that the id is only defined in the root class. It is not allowed to a child class to declare an id.

id [PK] integer	dog_class_name character varying(31)	name character varyi	littlebark character v	hugepoowe integer
1	SMALL DOG	Red	hau	
2	SMALL DOG	Green	hiu	
3	SMALL DOG	Black	hie	
4	HUGE DOG	Yellow		3
5	HUGE DOG	Brown		3
6	HUGE DOG	Snow		3

It is also possible to define the class discriminator column as integer:

- @DiscriminatorColumn(name = “DOG_CLASS_NAME”, discriminatorType = DiscriminatorType.INTEGER) => define the field as integer

•@DiscriminatorValue(“1″) => The value to be persisted in the Entity must change, a number will be persisted instead a text.

Mapping Hierarchy: Joined

When a hierarchy is using the Joined strategy each entity will have its data stored in the respective table. Instead of using just one table for all hierarchy classes, each entity will have its own table.

Check the code below to see how Joined strategy can be used:

```
import javax.persistence.*;

@Entity
@Table(name = "DOG")
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "DOG_CLASS_NAME")
public abstract class Dog {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String name;

    // get and set
}
```

```
import javax.persistence.*;

@Entity
@DiscriminatorValue("HUGE_DOG")
public class HugeDog extends Dog {
    private int hugePooWeight;

    public int getHugePooWeight() {
        return hugePooWeight;
    }

    public void setHugePooWeight(int hugePooWeight) {
        this.hugePooWeight = hugePooWeight;
    }
}
```

```
import javax.persistence.*;

@Entity
@DiscriminatorValue("SMALL_DOG")
public class SmallDog extends Dog {
    private String littleBark;
```

```
    public String getLittleBark() {
        return littleBark;
    }

    public void setLittleBark(String littleBark) {
        this.littleBark = littleBark;
    }
}
```

About the code above:

- The annotation @Inheritance(strategy = InheritanceType.JOINED) now has the Joined value.
- Check below how the tables are:

Dog table

id [PK] integer	dog_class_name character varying(31)	name character varying(255)
1	SMALL DOG	Red
2	SMALL DOG	Green
3	SMALL DOG	Black
4	HUGE DOG	Yellow
5	HUGE DOG	Brown
6	HUGE DOG	Snow

HugeDog table

id [PK] integer	hugepooweight integer
4	6
5	3
6	4

SmallDog table

id [PK] integer	littlebark character varying(255)
1	hau
2	hiu
3	hie

Notice in the images above how the data is persisted to each table. Every entity has its information persisted across unique tables; for this strategy JPA will use one table per entity regardlessthe entity being concrete or abstract.

The Dog table maintains all data common to all the classes of the hierarchy; The Dog table maintains a column indicating to which entity a row belongs to.

Mapping Hierarchy: Table per Concrete Class

The Table Per Concrete strategy will create a table per concrete entity. If an abstract entity is found in the hierarchy this data will be persisted in the concrete children entities database tables.

Check the code below:

```
import javax.persistence.*;

@Entity
@Table(name = "DOG")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Dog {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String name;

    // get and set
}
```

```
import javax.persistence.Entity;

@Entity
public class HugeDog extends Dog {
    private int hugePooWeight;

    public int getHugePooWeight() {
        return hugePooWeight;
    }

    public void setHugePooWeight(int hugePooWeight) {
        this.hugePooWeight = hugePooWeight;
    }
}
```

```
import javax.persistence.Entity;

@Entity
```

```
public class SmallDog extends Dog {
    private String littleBark;

    public String getLittleBark() {
        return littleBark;
    }

    public void setLittleBark(String littleBark) {
        this.littleBark = littleBark;
    }
}
```

About the code above:

•@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) => define the hierarchy type as table per class.
•The annotation @DiscriminatorColumn(name = “DOG_CLASS_NAME”) is not used in the children classes anymore. Each concrete class will have its own data, JPA will not spread entity data across tables.
•There is no need for the annotation @DiscriminatorValue for the same principal as above.
•Check below the database tables:

HugeDog Table

id [PK] integer	hugepooweight integer	name character varying(255)
4	6	Yellow
5	3	Brown
6	4	Snow

SmallDog Table

id [PK] integer	littlebark character varying(255)	name character varying(255)
1	hau	Red
2	hiu	Green
3	hie	Black

Notice that the attributes of the Dog entity are persisted in the HugeDog and SmallDog table.

Pros/Cons of each hierarchy mapping approach

Unfortunately there is no "better" approach to follow, each approach has its advantages and disadvantages. It is necessary to analyze the pros/cons of each approach and decide which is better for the application:

Approach	Pros	Cons
SINGLE_TABLE	Easier to understand the table model. Just one table is required.	Cannot have “non null” fields. Imagine that SmallDog has the hairColor attribute non null in the database. When HugeDog get persisted an error message from the database will be receive, it will inform that hairColor cannot be null.
	The entities attributes can be found all in one table.	
	As general rule has a good performance.	
JOINED	Each entity will have its own database table to store the data.	The insert has a higher performance cost. An insert will be done to each database table used in the hierarchy. Imagine a hierarchy like C extends B extends A, when C is persisted 3 inserts will be executed – one to each mapped entity.
	It will follow the OO patterns applied in the application code.	The join number executed in the database queries will increase in deeper hierarchies.
TABLE_PER_CLASS	When the query is executed to bring just one entity the performance is better. A database table will have only one entity data.	Columns will be repeated. The attributes found in the abstract entities will repeated in the concrete child entities.
		When a query brings more than one entity of the hierarchy this query will have a higher cost. It will be used UNION or one query per table.

Embedded Objects

Embedded Objects is a way to organize entities that have different data in the same table. Imagine a database table that maintains "person" information (e.g. name, age) and address data (street name, house number, city etc).

Check the picture below:

id [PK] integer	name character varying(255)	age integer	house_address character varying(255)	house_number integer	house_color character varying(255)
1	John	22	Street A	22	Red
2	Mary	33	Street B	33	Black
3	Joseph	44	Street C	44	Green

It is possible to see data related to a person and their corresponding address also. Check the code below to see an example of how to implement the Embedded Objects concept with the Person and Address entities:

```
import javax.persistence.*;

@Embeddable
public class Address {
    @Column(name = "house_address")
    private String address;

    @Column(name = "house_color")
    private String color;

    @Column(name = "house_number")
    private int number;

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    // get and set
}
```

```
import javax.persistence.*;

@Entity
@Table(name = "person")
public class Person {

    @Id
    private int id;
    private String name;
    private int age;
```

```
    @Embedded
    private Address address;

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    // get and set
}
```

About the code above:

- The @Embeddable annotation (Address class) allows the class to be used inside an entity, notice that Address is not an entity. It is just a class to help organize the database data.
- A @Column annotation is used in the Address class to indicate the table database column name.
- The @Embedded annotation (Person entity) indicates that JPA will map all the fields that are inside the Address class as belonging to the Person entity.
- The Address class can be used in other entities. There are ways to override the @Column annotation at runtime.

ElementCollection – how to map a list of values into a class

Sometimes it is needed to map a list of values to an entity but those values are not entities themselves, e.g. person has emails, dog has nicknames etc.

Check the code below that demonstrates this situation:

```
import java.util.List;
import java.util.Set;

import javax.persistence.*;

@Entity
@Table(name = "person")
public class Person {

    @Id
    @GeneratedValue
    private int id;
```

```
    private String name;

    @ElementCollection
    @CollectionTable(name = "person_has_emails")
    private Set<String> emails;

    @ElementCollection(targetClass = CarBrands.class)
    @Enumerated(EnumType.STRING)
    private List<CarBrands> brands;

    // get and set
}
```

```
public enum CarBrands {
    FORD, FIAT, SUZUKI
}
```

About the code above:

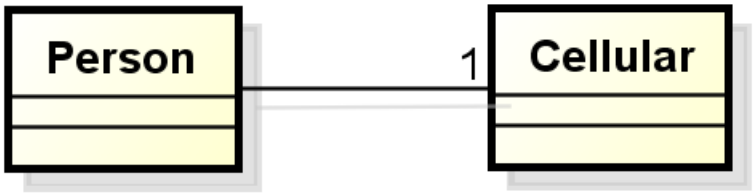
- Notice that two lists of data are used: Set<String>, List<CarBrand>. The @ElementCollection annotation is used not with entities but with “simple” attributes (e.g. String, Enum etc).
- The @ElementCollection annotatio is used to allow an attribute to repeat itself several times.
- The @Enumerated(EnumType.STRING) annotation is used with the @ElementCollection annotation. It defines how the enum will be persisted in the database, as String or as Ordinal (click here to more information).
- @CollectionTable(name = “person_has_emails”) => configure JPA to which table the information will be stored. When this annotation is not present JPA will create a database table named after the class and attribute name by default. For example with the “List<CarBrand> brands” attribute the database table would be named as "person_brands".

OneToOne unidirectional and bidirectional

It is very easy to find entities with relationships. A person has dogs, dogs have fleas, fleas has… hum… never mind.

Unidirectional

A one to one relationship is the easiest to understand. Imagine that a Person has only one Cellular and only Person will “see” the Cellular, the Cellular will not see Person. Check the image below:



Check how the Person class will be:

```
import javax.persistence.*;

@Entity
public class Person {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @OneToOne
    @JoinColumn(name="cellular_id")
    private Cellular cellular;

    // get and set
}
```

```
import javax.persistence.*;

@Entity
public class Cellular {

    @Id
    @GeneratedValue
    private int id;

    private int number;

    // get and set
}
```

About the code above:

•In a unidirectional relationship just one side of the relationship knows (“sees”) the other. Notice that Person knows Cellular but Cellular does not know Person. It is possible to do person.getCellular() but it is not possible to do cellular.getPerson().
•In the Person entity it is possible to use the annotation @OneToOne. This annotation indicates to JPA that there is a relationship between the entities.

Every relationship needs one of the entities to be the “relationship owner”. Being the relationship owner is nothing more than to have the foreign key in the database table. In the code above you can see that the annotation @JoinColumn has been used. This annotation indicates that the foreign key will be located in the person database table, making the Person entity owner of the relationship.

Bidirectional

To transform this relationship in a bidirectional one we just have to edit the Cellular entity. Check the class below:

```
import javax.persistence.*;

@Entity
public class Cellular {

    @Id
    @GeneratedValue
    private int id;

    private int number;

    @OneToOne(mappedBy="cellular")
    private Person person;

    // get and set
}
```

About the code above:

- The same annotation @OneToOne to the Person attribute is used in the Cellular entity.
- The parameter “*mappedBy*” was used in the @OneToOnea nnotation. This parameter indicates that the entity Person is the owner of the relationship; the foreign key must exist inside the person table, and not the Cellular table.

A developer must have in mind that for JPA to work in an optimal way it is a good practice to leave one side of the relationship as the owner. If the annotation @OneToOne found in the Cellular entity was without the “mappedBy” parameter, JPA would handle the Cellular entity as the owner of the relationship too. It is not a good idea to leave either sides of a relationship without defining "mappedBy", or both with "mappedBy" setted.

There is no such “auto relationship”

For a Bidirectional relationship to work correctly it is necessary to do like below:

```
person.setCellular(cellular);
cellular.setPerson(person);
```

JPA uses the Java concept of class reference, a class must maintain a reference to another one if there will be a join between them. JPA will not create a relationship automatically; to have the relationship in both sides it is needed to do like above.

OneToMany/ManyToOne unidirectional and bidirectional

The one to many relationship is used when an entity has a relationship with a list of other entities, e.g. a Cellular may have several Calls but a Call can have only one Cellular. The OneToMany relationship is represented with a list of values, there is more than one entity associate with it.

Let us start with the ManyToOne side:

```
import javax.persistence.*;

@Entity
public class Call {

    @Id
    @GeneratedValue
    private int id;

    @ManyToOne
    @JoinColumn(name = "cellular_id")
    private Cellular cellular;

    private long duration;

    // get and set
}
```

About the code above:

- The annotation @ManyToOne is used.
- Notice that annotation @JoinColumn is used to define who the owner of the relationship is.
- The @ManyToOne side will always be the owner of the relationship. There is no way of using the mappedBy attribute inside the @ManyToOne annotation.

To do a bidirectional relationship we need to edit the Cellular entity (created in the previous sections). Check the code below:

```
import javax.persistence.*;
```

```
@Entity
public class Cellular {

    @Id
    @GeneratedValue
    private int id;

    @OneToOne(mappedBy = "cellular")
    private Person person;

    @OneToMany(mappedBy = "cellular")
    private List<Call> calls;

    private int number;

    // get and set
}
```

About the code above:

- The @OneToMany annotation is used. This annotation must be placed in a collection.
- The mappedBy directtive is used to define the Call entity as the relationship owner.

Every relationship needs one of the entities to be the “relationship owner”. Being the relationship owner is nothing more than to have the foreign key in the database table. In the code above you can see that the annotation @JoinColumn has been used. This annotation indicates that the foreign key will be located in the call database table, making the Call entity owner of the relationship.

There is no such “auto relationship”

For a Bidirectional relationship to work correctly it is necessary to do like below:

```
call.setCellular(cellular);
cellular.setCalls(calls);
```

JPA uses the Java concept of class reference, a class must maintain a reference to another one if there will be a join between them. JPA will not create a relationship automatically; to have the relationship in both sides it is needed to do like above.

ManyToMany unidirectional and bidirectional

In a ManyToMany example a person may have several dogs and a dog may have several persons (imagine a dog that lives in a house with 15 persons).

In a ManyToMany approach it is necessary to use an extra database table to store the ids that relate the database

tables of every entity of the relationship. Thus for the specific example we will have a person table, a dog table and a relationship table named person_dog. The person_dog table would only maintain the person_id and dog_id values that represent which dog belongs to which person.

See the database table images below:

Person table

id [PK] integer	name character v	cellular_id integer
1	Mary	2

Dog table

id [PK] integer	name character v
4	Spike
5	Snow

person_dog table

person_id [PK] integer	dog_id [PK] integer
1	4
1	5

Notice that the person_dog has only ids.

Check the Person entity below:

```
import java.util.List;

import javax.persistence.*;

@Entity
public class Person {

    @Id
    @GeneratedValue
    private int id;
```

```
    private String name;

    @ManyToMany
    @JoinTable(name = "person_dog", joinColumns = @JoinColumn(name = "person_id"), inverseJoinColumns =
@JoinColumn(name = "dog_id"))
    private List<Dog> dogs;

    @OneToOne
    @JoinColumn(name = "cellular_id")
    private Cellular cellular;

    // get and set
}
```

About the code above:

- The @ManyToMany annotation is used.
- The @JoinTable annotation is used to set the relationship table between the entities; “name” sets the table name; “joinColumn” defines the column name in the table of the relationship owner; “inverseJoinColumns” defines the column name in the table of the non-relationship owner.

The Person entity has unidirectional relationship with the Dog entity. Check how the Dog entity would look like in a bidirectional relationship:

```
import java.util.List;

import javax.persistence.*;

@Entity
public class Dog {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @ManyToMany(mappedBy="dogs")
    private List<Person> persons;

    // get and set
}
```

As you can see the @ManyToMany annotation is configured with the "mappedBy" option, which establishes the Person entity as the relationship owner.

Every relationship needs that one of the entities be the “relationship owner”. For the ManyToMany association the relationship owner entity is the one that is dictated by the "mappedBy" option usually copnfigured with the

@ManyToMany annotation at the non-owner entity of the relation. If the "mappedBy" option is not found in any of the related entities JPA will define both entities as the relationship owners. The "mappedBy" option must point to the associated entity's attribute name and not the associated entity's name.

There is no such “auto relationship”

For a Bidirectional relationship to work correctly it is necessary to do like below:

```
person.setDog(dogs);
dog.setPersons(persons);
```

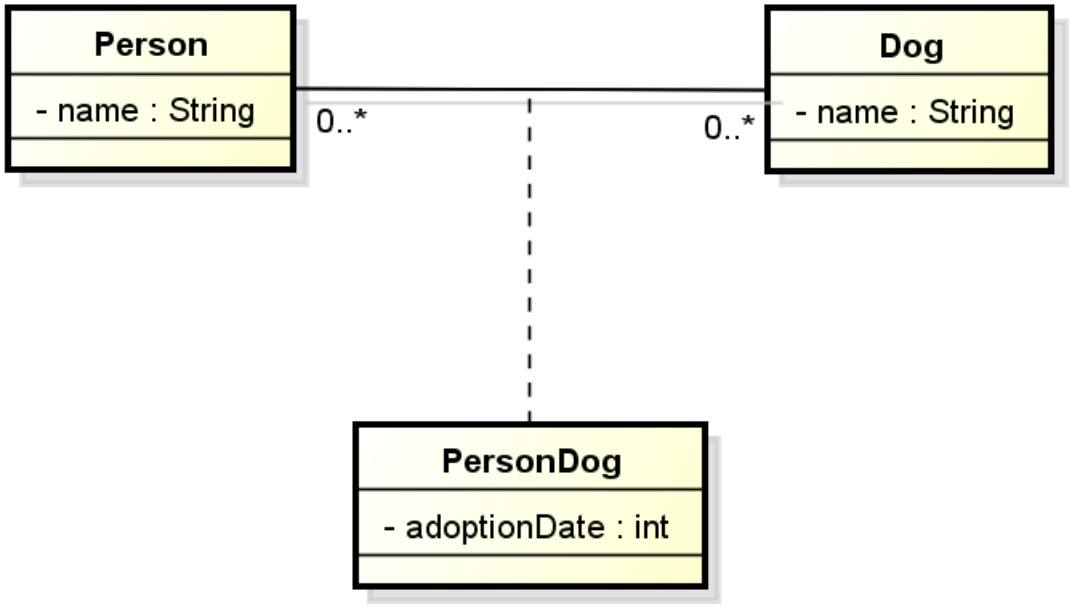
JPA uses the Java concept of class reference, a class must maintain a reference to another one if there will be a join between them. JPA will not create a relationship automatically; to have the relationship in both sides it is needed to do like above.

ManyToMany with extra fields

Imagine a Person entity that has a ManyToMany relationship with the Dog entity; every time that a dog is adopted by a person the application should register the adoption date. This value must be stored in the relationship and not an an attribute on person nor dog.

To handle this kind of situationwe use the "Associative Class" alias "Associative Entity" approach. With this approach it is possible to store extra data when the ManyToMany relationship is created.

The image below shows how this entity can be mapped:



To map this extra field, implement like the following code:

```
import java.util.List;

import javax.persistence.*;

@Entity
public class Person {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @OneToMany(mappedBy = "person")
    private List<PersonDog> dogs;

    // get and set
}
```

```
import java.util.List;

import javax.persistence.*;

@Entity
public class Dog {

    @Id
    @GeneratedValue
    private int id;
```

```
    private String name;

    @OneToMany(mappedBy = "dog")
    private List<PersonDog> persons;

    // get and set
}
```

The code above is using the @OneToMany relationship with the mappedBy option. Notice that there is no @ManyToMany relationship between the entities, but there is an entity PersonDog that unite both entities.

Below is the PersonDog code:

```
import java.util.Date;

import javax.persistence.*;

@Entity
@IdClass(PersonDogId.class)
public class PersonDog {

    @Id
    @ManyToOne
    @JoinColumn(name="person_id")
    private Person person;

    @Id
    @ManyToOne
    @JoinColumn(name="dog_id")
    private Dog dog;

    @Temporal(TemporalType.DATE)
    private Date adoptionDate;

    // get and set
}
```

In the code above you can see the relationships between PersonDog, Dog and Person, and an extra attribute to store the adoption date. There is an id class to store the relationship ids named "PersonDogId":

```
import java.io.Serializable;

public class PersonDogId implements Serializable {

    private static final long serialVersionUID = 1L;

    private int person;
    private int dog;

    public int getPerson() {
        return person;
    }
```

```
    public void setPerson(int person) {
        this.person = person;
    }

    public int getDog() {
        return dog;
    }

    public void setDog(int dog) {
        this.dog = dog;
    }

    @Override
    public int hashCode() {
        return person + dog;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof PersonDogId){
            PersonDogId personDogId = (PersonDogId) obj;
            return personDogId.dog == dog && personDogId.person == person;
        }

        return false;
    }
}
```

One thing to notice here is that person and dog attributes must have the same name - "person" and "dog" here - between the PersonDogId and PersonDog entities. It is how JPA works. More information about complex keys can be found at earlier sections of this document.

How the Cascade functionality works? How should a developer use the OrphanRemoval? Handling the org.hibernate.TransientObjectException

It is very common two or more entities to receive updates in the same transaction. When editing a person's data for example, we could change their name, address, age, car color etc. These changes should trigger updates to three different entities: Person, Car and Address.

The updates above could be done as shown in the code snippet below:

```
car.setColor(Color.RED);
car.setOwner(newPerson);
car.setSoundSystem(newSound);
```

If the code below were to be executed the exception org.hibernate.TransientObjectException would be thrown:

```
EntityManager entityManager = // get a valid entity manager

Car car = new Car();
car.setName("Black Thunder");

Address address = new Address();
address.setName("Street A");

entityManager.getTransaction().begin();

Person person = entityManager.find(Person.class, 33);
person.setCar(car);
person.setAddress(address);

entityManager.getTransaction().commit();
entityManager.close();
```

With the EclipseLink JPA implementation the following message would be fired: "*Caused by: java.lang.IllegalStateException: During synchronization a new object was found through a relationship that was not marked cascade PERSIST*".

What means that an entity is transient? Or what a relationship not marked with cascade persist is all about?

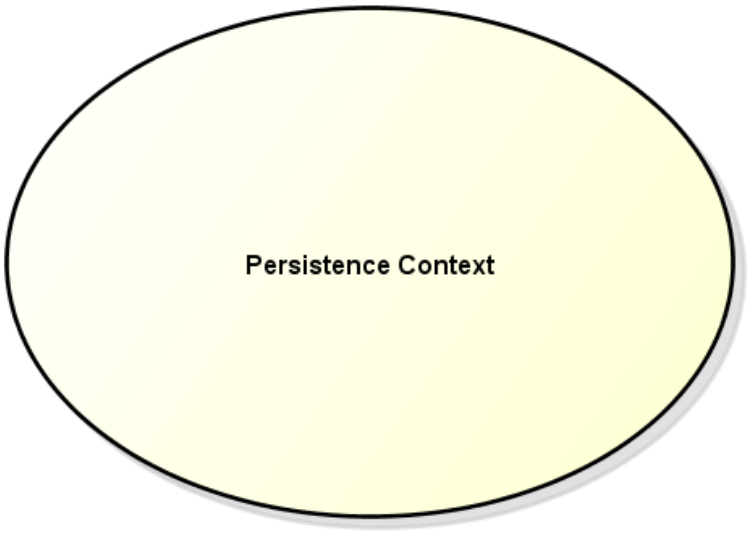
JPA works as a tracker for every entity that participates in a transaction. An entity that participates in a transaction is an entity that will be created, updated, deleted. JPA needs to know where that entity came from and where it is going to. When a transaction is opened every entity brought from the database is "attached". With "attached" we mean that the entity is inside a transaction, being monitored by JPA. An entity remains attached until the transaction is closed (rules for JSE applications or transactions without EJB Stateful Session Beans with Persistence Scope Extended are different); to be attached an entity needs to come from a database (by query, entity manager find method…), or receive some contact inside the transaction (merge, refresh).

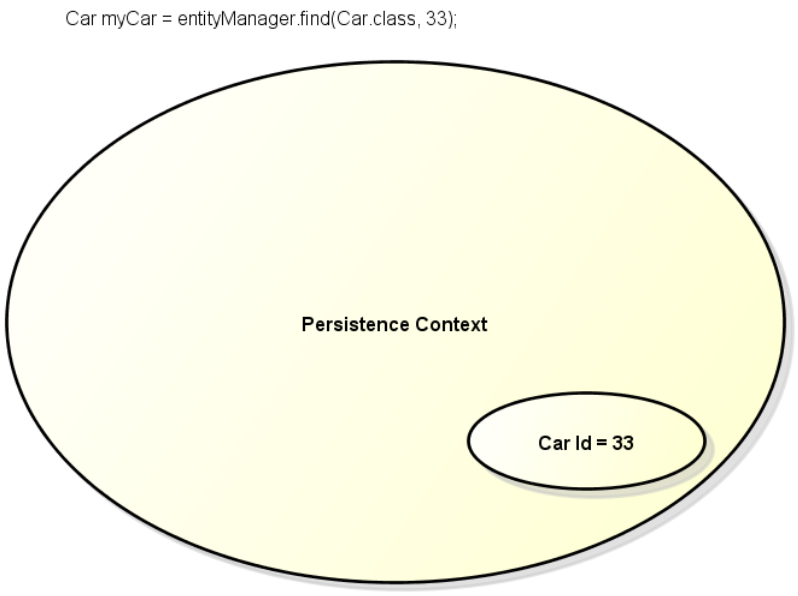
Notice the code below:

```
entityManager.getTransaction().begin();
Car myCar = entityManager.find(Car.class, 33);
myCar.setColor(Color.RED);
entityManager. getTransaction().commit();
```

The transaction is opened, an update is made in the entity and the transaction is committed. It was not required to do an explicit update to the entity, with the transaction committing all updates made to the entity will be persisted to the database. The updates made to the Car entity were persisted to the database because the entity is attached, any update to an attached entity will be persisted in the database after the transaction commits() or a flush call is made.

As shown In the code snippet above "myCar" entity was brought from the database inside a transaction, thus JPA will have the "myCar" entity attached into that Persistence Context. It is possible to define a Persistence Context as a place where JPA will put all attached entities to that transaction, or as a big bag. The images below show this concept:

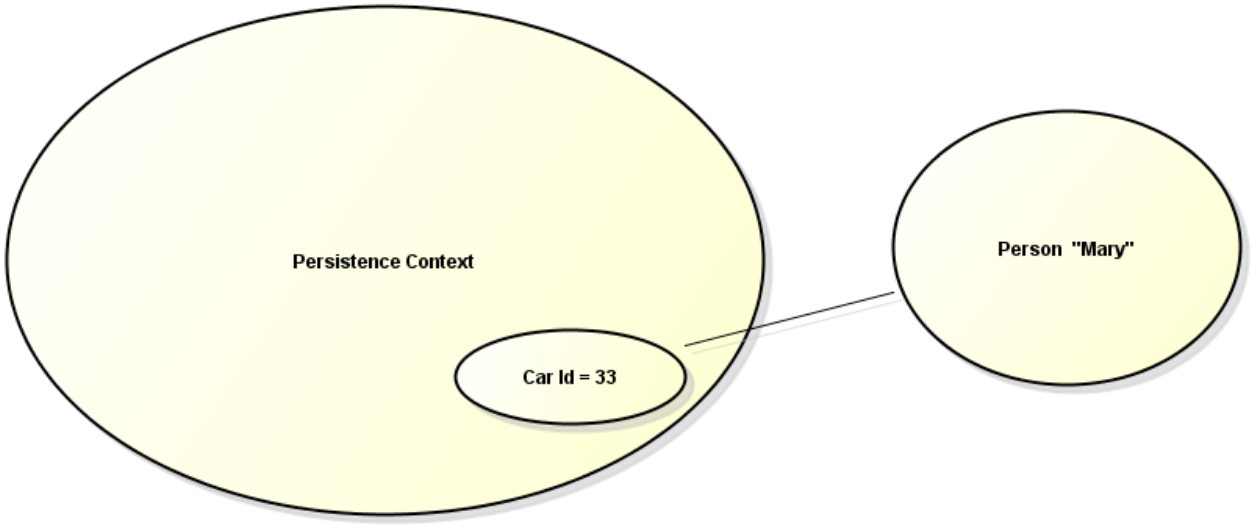




Notice in the image above that the Car entity is added to the Persistence Context only after a database query is performed to retrieve it from the database. Every update in the Car entity will be monitored by JPA. Once the transaction is finished (or the flush command invoked), JPA will persist this changes to the database.

The aforementioned problem occures when we update a relationship between two entities. Check the code and the image below:

```
entityManager.getTransaction().begin();
Person newPerson = new Person();
newPerson.setName("Mary");
Car myCar = entityManager.find(Car.class, 33);
myCar.setOwner(newPerson);
entityManager. getTransaction().commit();
```



The Car entity establishes a relationship with the Person entity. The problem is that the Person entity is outside of the Persistence Context, notice that the person entity was not brought from the database nor was attached to the transaction. Upon commit JPA cannot recognize that Person is a new entity, which does not exist in the database. Even if the Person entity existed in the database, since it came from outside a transaction (e.g. a JSF ManagedBean, Struts Action) it will be considerate unmanagable in this Persistence Context. An object outside the Persistence Context is known as “detached”.

This detached entity situation may happen in any JPA operations: INSERT, UPDATE, DELETE…

To help in these situations JPA created the option Cascade. This option can be defined in the annotations: @OneToOne, @OneToMany and @ManyToMany. The enum javax.persistence.CascadeType has all Cascade options available.

Check the cascade options below:

- •CascadeType.DETACH
- •CascadeType.MERGE
- •CascadeType.PERSIST
- •CascadeType.REFRESH
- •CascadeType.REMOVE
- •CascadeType.ALL

The Cascade applies the behavior to repeat the action defined in the relationship. See the code below:

```
import javax.persistence.*;

@Entity
public class Car {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @OneToOne(cascade = CascadeType.PERSIST)
    private Person person;

    // get and set
}
```

In the persistence code above it is defined that the @OneToOne Person relationship will have the Cascade.PERSIST action executed every time the command entityManager.persist(car) is executed; thus for every persist action invoked in a Car entity JPA will invoke persist in the Person relationship also.

The Cascade advantage is that the propagation of an action is automatic, just need to configure it in a relationship.

Check below the Cascade options:

Type	Action	Triggered by
CascadeType.DETACH	When the entity is removed from the Persistence Context (it will cause the entity to be detached) this action will be reflected in the relationship.	Finished Persistence Context or by command: entityManager.detach(), entityManager.clear().
CascadeType.MERGE	When an entity has any	When the entity is updated and

	data updated this action will be reflected in the relationship.	the transaction finishes or by command: entityManager.merge().
CascadeType.PERSIST	When a new entity is persisted in the database this action will be reflected in the relationship.	When a transaction finishes or by command:entityManager.persist().
CascadeType.REFRESH	When an entity has its data synchronized with the database this action will be reflected in the relationship.	By command: entityManager.refresh().
CascadeType.REMOVE	When an entity is deleted from the database this action will be reflected in the relationship.	By command: entityManager.remove().
CascadeType.ALL	When any of the actions above is invoked by the JPA or by command, this action will be reflected in the relationship.	By any command or action described above.

Once the cascade is defined, the code below should run without error:

```
entityManager.getTransaction().begin();
Person newPerson = new Person();
newPerson.setName("Mary");
Car myCar = entityManager.find(Car.class, 33);
myCar.setOwner(newPerson);
entityManager. getTransaction().commit();
```

Observations about Cascade:

- A developer must be cautious when using CascadeType.ALL in a relationship. When the entity is deleted its relationship would be deleted as well. In the sample code above, if the cascade type in the Car entity was set to the ALL option, when a Car entity was deleted from the database the Person entity would be deleted also.
- CascadeType.ALL (or individual cascades) can cause low performance in every action triggered in the entity. If for example an entity has a lot of lists of referenced entities a merge() action invoked on the specific entity could cause all lists to be merged too.
- car.setOwner(personFromDB) => if the "personFromDB" entity exists in the DB but is detached

for the persistence context, Cascade will not help. When the command entityManager.persist(car) is executed JPA will run the persist command for every relationship defined with CascadeType.PERSIST (e.g. entityManager.persist(person)). If the Person entity already exists in the database JPA will try to insert the same record again and an error message will be thrown. In this case only “attached” entities can be used, the best way to do it is by using the getReference() method that we present here in more detail.

In order for JPA to trigger the cascade it is necessary always to execute the action in the entity that has the cascade option defined. Check the code below:

```
import javax.persistence.*;

@Entity
public class Car {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @OneToOne(cascade = CascadeType.PERSIST)
    private Person person;

    // get and set
}
```

```
import javax.persistence.*;

@Entity
public class Person {

    @Id
    private int id;

    private String name;

    @OneToOne(mappedBy="person")
    private Car car;

    // get and set
}
```

The correct way to trigger the cascade in the entities above is:

```
entityManager.persist(car);
```

JPA will search inside the Car entity if there is a Cascade option that should be triggered. If the persist was executed like below the transient error message would be thrown:

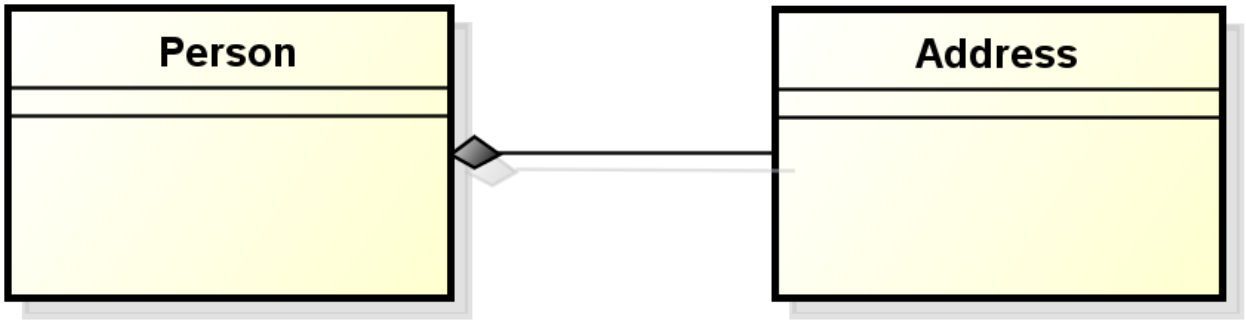
```
entityManager.persist(person);
```

Remember: the cascade will only be triggered by JPA when the action is executed in the entity that has the Cascade configured, in the sample above the Car class defined the Cascade to Person; only the Car class will trigger the Cascade.

OrphanRemoval

The OrphanRemoval option works almost like the CascadeType.REMOVE. The OrphanRemoval is usually applied in cases where an entity just exists inside another entity.

Imagine a situation where an Address entity will only exist inside a Person entity:



If a new Person entity is persisted to the database an Address entity will be created too. Conceptually the Address entity is created only when a new Person entity is created and when the Person entity is deleted the Address entity is deleted also, just like by using the CascadeType.REMOVE option.

The OrphanRemoval has almost the same functionality as the CascadeType.REMOVE, but conceptually it must be applied at class composition level.

Look at the code below:

```
import javax.persistence.*;

@Entity
public class Address {
```

```
    @Id
    @GeneratedValue
    private int id;

    private String name;

    // get and set
}
```

```
import javax.persistence.*;

@Entity
public class Person {

    @Id
    private int id;

    private String name;

    @OneToMany(orphanRemoval=true)
    private List<Address> address;

    // get and set
}
```

Imagine that a specific Address entity is assigned to a specific Person entity and only that Person entity has access to it.

As stated above OrphanRemoval is ***almost*** like CascadeType.REMOVE. The difference is that the dependent entity will be deleted from the database if the attribute is set to null as showin at the code snippet below:

```
person.setAddress(null);
```

When the Person entity gets updated in the database, the Address entity will be deleted from the database. The person.setAddress(null) would cause the Address entity to be orphan.

The OrphanRemoval option is only available in the annotations: @OneToOne and @OneToMany.

How to delete an entity with relationships. Clarify which relationships are raising the exception

When an entity is deleted from the database a constraint error may appear, something like:“*java.sql.SQLIntegrityConstraintViolationException*”. This error might happen if we try to delete a Person entity that is related to a Car entity and the CascadeType.REMOVE is not defined for the specific association (see the last chapter about Cascade definition).

In order to perform the removal of the Person entity we could do:

- •CascadeType.REMOVE => Once the entity is deleted its child entity (Car) will be deleted too. Check the previous section to see how to do it.
- •OrphanRemoval => Can be applied but have a different behavior than CascadeType.REMOVE. Check the last chapter to see how to do it.
- •Set the relationship to null before excluding it:

```
        person.setCar(null);
        entityManager.remove(person);
```

Unfortunately locating the actual entity that is the cause of the specific exception is not straightforward at all. What can be done is catch the class name displayed in the error message of the exception and go on from there. This solution is not accurate because the error message is a String with a long text containing the database table name. Nevertheless this message may change by each distinct JPA implementation, JDBC driver, local database language etc.

Creating one EntityManagerFactory by application

When controlling database transactions programmatically one EntityManagerFactory per application is usually used. This is the optimal approach since loading an EntityManagerFactory has a high performance cost; JPA will analyze the database, validate entities and perform several other tasks when creating a new EntityManagerFactory. Thus it is unviable to create a new EntityManagerFactory per transaction.

Below is a code that can be used:

```
import javax.persistence.*;

public abstract class ConnectionFactory {
    private ConnectionFactory() {
    }

    private static EntityManagerFactory entityManagerFactory;

    public static EntityManager getEntityManager(){
        if (entityManagerFactory == null){
            entityManagerFactory = Persistence.createEntityManagerFactory("MyPersistenceUnit");
        }

        return entityManagerFactory.createEntityManager();
    }
}
```

Understanding how the Lazy/Eager option works

An entity may have an attribute with a huge size (e.g. a large file) or a big list of entities e.g a person may have several cars or a picture with 150 MB of size.

Check the code below:

```
import javax.persistence.*;

@Entity
public class Car {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @ManyToOne
    private Person person;

    // get and set
}
```

```
import java.util.List;

import javax.persistence.*;

@Entity
public class Person {

    @Id
    private int id;

    private String name;

    @OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
    private List<Car> cars;

    @Lob
    @Basic(fetch = FetchType.LAZY)
    private Byte[] hugePicture;

    // get and set
}
```

About the code above:

- Notice that the collection is assigned a FetchType.LAZY option.
- Notice that the image Byte[] hugePicture is assigned a FetchType.LAZY option also.

When a field is marked with a lazy fetch type option it is implied that JPA should not load the specific attribute “naturally”. When the command entityManager.find(Person.class, person_id) is used to retrieve the specific person from the database the "cars" list and the "hugePicture" field will not be loaded. The amount of data returned from the database will be smaller. The advantage of this approach is that the query has less performance impact and the data traffic over the network is smaller.

This data can be accessed when the relevant get methods of the Person entity are invoked. When the command person.getHugePicture() is executed a new query will be fired against the database searching this information.

Every simple attribute will have the FetchType.EAGER option enabled by default. To mark a simple attribute as LAZY fetched just add the annotation @Basic with the Lazy option selected just like above.

The relationship between entities has a default behavior too:

- Relationships finish with “One” will be EAGERly fetched: @OneToOne and @ManyToOne.
- Relationships finish with “Many” will be LAZYly fetched: @OneToMany and @ManyToMany.

To change the default behavior you should do like demonstrated above.

When we use the LAZY behavior the exception “Lazy Initialization Exception” may happen. This error happens when the LAZY attribute is accessed without any opened connection. Check this post to see how to solve this problem: Four solutions to the LazyInitializationException.

The problem with EAGER fetch type in all list/attributes is that the database query may increase a lot. For example if every Person entity has a list of thousant of actions logs, imagine the cost to bring hundred or even thousants of Person entities from the database! A developer must be very cautious when choosing which kind of fetch strategy will be used to every attribute/relationship.

Handling the “cannot simultaneously fetch multiple bags” error

This error happens when JPA retrieves an entity from the database and this entity has more than one list fields marked for EAGERly fetching.

Check the code below:

```
import java.util.List;
```

```
import javax.persistence.*;

@Entity
public class Person {

    @Id
    private int id;

    private String name;

    @OneToMany(mappedBy = "person", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    private List<Car> cars;

    @OneToMany(fetch = FetchType.EAGER)
    private List<Dog> dogs;

    // get and set
}
```

When a query to get the Person entity above is fired the following error message is displayed: “javax.persistence.PersistenceException: org.hibernate.HibernateException: cannot simultaneously fetch multiple bags”, a list can be known as bag.

This error happens because Hibernate (the specific JPA implementation farmework used for this example) tries to bring the same result amount for each list. If the generated SQL returns 2 lines for the "dog" entity list and one for the "car" entity list, Hibernate will repeat the "car" query to make the result equal. Check the images below to understand what is happening when the entityManager.find(Person.class, person_id) command is executed:

Tabela DOG

id	name	age
1	Red	2
2	Black	3

Tabela CAR

id	name	color
1	Thunder	Green

entityManager.find(Person.class, 33)

person.id	dog.id	dog.name	dog.age	car.id	car.name	car.color
33	1	Red	2	1	Thunder	Green
33	2	Black	3	1	Thunder	Green

The problem arises when repeated results appear and break the correct query result. The red cells in the image above are the repeated results.

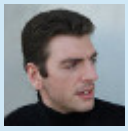
There are four solutions for this situation:

- To use java.util.Set instead of other collection types => with this easy change the error can be avoided.
- To use EclipseLink => it is a radical solution, but for JPA only users that use only JPA annotations this change will have minimal impact.
- To use FetchType.LAZY instead of EAGER => this solution is a temporary solution, because if a query fetches data from two collections this error may occur again. For example the following query could trigger the error: “select p from Person p join fetch p.dogs d join fetch p.cars c”. Additionally when using this approach the LazyInitializationException error may happen.
- To use @LazyCollection or @IndexColumn of the Hibernate implementation in the collection => it is a very good idea to understand how the @IndexColumn works and its effects when used, its behavior will change varying to which side of the relationship it is added (the explanation of this annotation is outside the scope of this mini book).

ABOUT THE AUTHOR

Hebert Coelho is a senior Java Development, with 4 certifications and a published book about JSF (portuguese only). Founder of the blog uaiHebert.com visited from more than 170 different countries.

ABOUT THE EDITOR



Byron Kiourtzoglou is a master software engineer working in the IT and Telecom domains. He is an applications developer in a wide variety of applications/services. He is currently acting as the team leader and technical architect for a proprietary service creation and integration platform for both the IT and Telecom industries in addition to a in-house big data real-time analytics solution. He is always fascinated by SOA, middleware services and mobile development. Byron is co-founder and Executive Editor at Java Code Geeks.