# JPA Mini Livro

## Primeiros passos e conceitos detalhados

Autor: Hebert Coelho de Oliveira

# Conteúdo

# Capítulo 1: Introdução

Vamos ver sobre JPA: o que é JPA, para que serve o persistence.xml, criar corretamente uma entidade, como realizar desde mapeamentos simples até os mapeamentos complexos de chaves primárias, criar relacionamentos entre as entidades, modos para persistência automática e outros.

# Capítulo 2: Quais os motivos que levaram à criação do JPA

Um dos grandes problemas da Orientação a Objetos é como mapear seus objetos para refletir o banco de dados. É possível ter uma classe com o nome de Carro mas seus dados estarem salvos em uma tabela chamada TB_CARRO. O nome da tabela seria apenas o começo dos problemas, se sua classe Carro tem o campo "*nome*" mas na tabela está como STR_NAME_CAR?

O framework mais básico do Java para acessar o banco de dados é o JDBC. Infelizmente, com o JDBC, é necessário um trabalho braçal para transformar o resultado que vem do banco de dados em uma classe.

Para obter objetos do tipo carros (no código abaixo está como Car) vindo do banco de dados utilizando JDBC seria necessário ter o código abaixo:

```
1 import java.sql.*;
2 import java.util.LinkedList;
3 import java.util.List;
4
5 public class MainWithJDBC {
6    public static void main(String[] args) throws Exception {
7        Class.forName("org.hsqldb.jdbcDriver");
8
9        Connection connection = // get a valid connection
10
11        Statement statement = connection.createStatement();
12
13        ResultSet rs = statement.executeQuery("SELECT \"Id\", \"Name\" FROM \"Car\"");
14
15        List<Car> cars = new LinkedList<Car>();
16
17        while(rs.next()){
18            Car car = new Car();
19            car.setId(rs.getInt("Id"));
20            car.setName(rs.getString("Name"));
21            cars.add(car);
22        }
23
24        for (Car car : cars) {
25            System.out.println("Car id: " + car.getId() + " Car Name: " +
car.getName());
26        }
27
28        connection.close();
29    }
30 }
```

No código acima é possível ver como é trabalhoso transformar os valores retornados pela consulta em objetos. Imagine uma classe com 30 campos… Para piorar imagine uma classe com 30 campos e relacionado com outra classe que também tenha 30 campos? Por exemplo, um Carro pode ter uma Lista de Pessoas e cada objeto da classe Pessoa teria 30 campos.

Outra desvantagem de uma aplicação que utiliza o JDBC puro é a sua portabilidade. A sintaxe de cada query pode variar entre bancos de dados. Para se limitar o número de linhas de uma consulta na Oracle se utiliza a palavra ROWNUM, já no SQL Server é utilizado a palavra TOP.

A portabilidade de uma aplicação que utiliza JDBC fica comprometida quando as queries nativas de cada banco são utilizadas. Existem soluções para esse tipo de problema, por exemplo, é ter cada consulta utilizada salva em um arquivo .sql fora da aplicação. Para cada banco diferente que a aplicação rodasse um arquivo .sql diferente seria utilizado.

É possível encontrar outras dificuldades ao longo do caminho: como atualizar os relacionamentos de modo automático, não deixar registros “órfãos” na tabela ou como utilizar hierarquia de um modo mais simples.

# Capítulo 3: O que é o JPA? O que são as implementações do JPA?

O JPA veio para solucionar todos os problemas listados na página anterior.

A proposta do JPA é que seja possível trabalhar diretamente com as classes e não ter que utilizar as consultas nativas de cada banco de dados; o JPA irá fazer esse trabalho pelo desenvolvedor.

O JPA nada mais é do que um conjunto de especificações (muitos textos, normas e interfaces do Java) de como uma implementação deve se comportar. Existem diversas implementações no mercado que seguem as especificações do JPA. Podemos citar o Hibernate, OpenJPA, EclipseLink e o “recente” Batoo.

As implementações tem a liberdade de adicionar anotações e códigos a mais que desejarem, mas tem que implementar o básico que o JPA requeira.

A primeira solução para o problema de portabilidade que o JPA apresenta é o modo de mapear os dados para dentro de cada classe. Como veremos mais a frente, é possível mapear os dados para cada coluna da classe mesmo que o nome do atributo seja diferente do nome da coluna no banco de dados.

O JPA criou uma linguagem de consulta chamada JPQL para realizar as consultas no banco de dados. A vantagem do JPQL é que a mesma consulta pode ser executada em todos os bancos de dados.

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

A consulta acima pode ser executada em JPQL conforme abaixo:

```
1 SELECT c FROM Car c
```

Note que o retorno da consulta é “c”, ou seja, um objeto do tipo carro e não os campos/valores existentes na tabela. O próprio JPA montará os objetos automaticamente.

Caso você queira ver como realizar diversos modos de query com JPA, acesse: http://uaihebert.com/?p=1137.

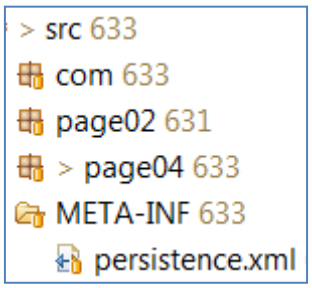
O JPA ficará responsável por traduzir cada JPQL para a sintaxe correta; o desenvolvedor não precisará tomar conhecimento de qual a sintaxe requerida para cada banco.

# Capítulo 4: Para que serve o persistence.xml? E suas configurações?

O arquivo do persistence.xml é o arquivo responsável por diversas configurações do sistema. Nele é possível definir configurações específicas da aplicação e do banco de dados, como também passar configurações específicas de cada implementação do JPA.

O código do persistence.xml apresentado aqui consta as configurações relacionadas ao EclipseLink, mas os conceitos e os códigos presentes nas classes Java deste post funcionam com qualquer implementação.

O arquivo persistence.xml deve estar localizado na pasta META-INF no mesmo diretório das classes. Abaixo um exemplo de onde esse arquivo deve estar localizado no Eclipse:




Essa imagem é valida para o Eclipse. Para o Netbeans é necessário verificar qual o correto lugar na documentação. No momento não o tenho instalado para fornecer essa informação.

Caso você tenha a mensagem de erro: "*Could not find any META-INF/persistence.xml file in the classpath*" você pode verificar o seguinte:

- Abra o arquivo WAR gerado e veja se o arquivo persistence.xml se encontra dentro de "*/WEB-INF/classes/META-INF/*"; Se não estiver seu erro está na criação do WAR. Caso o empacotamento seja automático você precisa colocar o arquivo no lugar que a IDE espera que ele esteja. Se seu empacotamento for manual (ant, maven) verifique o script de geração do arquivo.
- Se seu projeto for um projeto EAR verifique se o arquivo se encontra na raiz do jar EJB dentro da pasta "*META-INF*"; Se não estiver seu erro está na criação do WAR. Caso o empacotamento seja automático você precisa colocar o arquivo no lugar que a IDE espera que ele esteja. Se seu empacotamento for manual (ant, maven) verifique o script de geração do arquivo.

- Caso a aplicação seja JSE (desktop) verifique se não está na raiz das classes dentro da pasta "*META-INF*" no JAR. Por default o JPA irá procurar na raiz do JAR pelo endereço "*META-INF/*persistence.xml".
- Verifique se o arquivo está escrito todo em letras minúsculas "persistence.xml". O arquivo tem que ter o nome todo com letras minúsculas.

Caso o arquivo não seja localizado o JPA irá exibir a mensagem de erro citada acima. Como regra geral, o melhor lugar para se colocar o persistence.xml é como mostrado na imagem acima.

Veja abaixo, um exemplo de persistence.xml:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <persistence version="2.0"
4    xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
6
7    <persistence-unit name="MyPU" transaction-type="RESOURCE_LOCAL">
8        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
9
10        <class>page20.Person</class>
11        <class>page20.Cellular</class>
12        <class>page20.Call</class>
13        <class>page20.Dog</class>
14
15        <exclude-unlisted-classes>true</exclude-unlisted-classes>
16
17        <properties>
18            <property name="javax.persistence.jdbc.driver"
value="org.hsqldb.jdbcDriver" />
19            <property name="javax.persistence.jdbc.url"
value="jdbc:hsqldb:mem:myDataBase" />
20            <property name="javax.persistence.jdbc.user" value="sa" />
21            <property name="javax.persistence.jdbc.password" value="" />
22            <property name="eclipselink.ddl-generation" value="create-tables" />
23            <property name="eclipselink.logging.level" value="FINEST" />
24        </properties>
25    </persistence-unit>
26
27    <persistence-unit name="PostgresPU" transaction-type="RESOURCE_LOCAL">
28        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
29
30        <class>page26.Car</class>
31        <class>page26.Dog</class>
32        <class>page26.Person</class>
33
34        <exclude-unlisted-classes>true</exclude-unlisted-classes>
35
36        <properties>
37            <property name="javax.persistence.jdbc.url"
value="jdbc:postgresql://localhost/JpaRelationships" />
```

```
38            <property name="javax.persistence.jdbc.driver"
value="org.postgresql.Driver" />
39            <property name="javax.persistence.jdbc.user" value="postgres" />
40            <property name="javax.persistence.jdbc.password" value="postgres" />
41            <!-- <property name="eclipselink.ddl-generation" value="drop-and-create-
tables" /> -->
42            <property name="eclipselink.ddl-generation" value="create-tables" />
43            <!-- <property name="eclipselink.logging.level" value="FINEST" /> -->
44        </properties>
45    </persistence-unit>
46 </persistence>
```

Vamos analisar detalhadamente cada linha citada acima:

- <persistence-unit name=”MyPU” =>Com essa configuração definimos o nome do Persistence Unit. O Persistence Unit pode ser definido como o universo JPA da sua aplicação. Ele contém dados de todas as classes, relacionamentos, chaves e outras informações relacionados ao banco de dados. É possível também adicionar mais de um PersistenceUnit no mesmo arquivo conforme demonstrado acima.
- transaction-type=”RESOURCE_LOCAL” => Define qual o tipo de transação. Atualmente dois valores são permitidos: RESOURCE_LOCAL e JTA. Em aplicações desktop é utilizado o RESOURCE_LOCAL e para aplicações web pode se utilizar ambos, dependendo da arquitetura do sistema.
- <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider> => Define qual será o “provider” da implementação JPA utilizada. O provider na verdade é qual implementação será utilizada. Para o Hibernate é utilizado “*org.hibernate.ejb.HibernatePersistence*” e para o OpenJPA é utilizado “org.apache.openjpa.persistence.PersistenceProviderImpl”.
- <class></class> => Serve para declarar as classes. A necessidade dessa tag estar presente varia de acordo com a implementação. Por exemplo, no JSE ao utilizar o Hibernate não é obrigado listar todas as classes presentes no sistema, já com EclipseLink e OpenJPA é necessário declarar as classes.
- <exclude-unlisted-classes>true</exclude-unlisted-classes> => Essa configuração define que qualquer entidade, que não se encontra listada no arquivo persistence.xml, não deve ser mapeada dentro do Persistence Unit (próximo capítulo explicará o que é uma entidade). Essa configuração é muito útil quando são necessários dois ou mais Persistence Units no mesmo projeto onde uma entidade pode apenas existir em um Persistence Unit e não em outro.
- É possível deixar um código comentado utilizando os valores <!– –>
- <properties> => É possível passar parâmetros específicos a serem utilizados pela implementação. Valores como driver, senha e usuário é normal de encontrar em todas as implementações; geralmente esses valores são escritos em um persistence.xml quando temos uma aplicação que utiliza RESOURCE_LOCAL. Para transações JTA são utilizados datasources que contêm as configurações necessárias. Para configurar um datasource é necessário usar uma das duas anotações a seguire: <jta-data-source></jta-data-source> <non-jta-data-source></non-jta-data-source>. Alguns servidores, como o JBoss, tem como

requisito que mesmo um Persistence Unit de RESOURCE_LOCAL seja mapeado com as tags de datasource. Duas configurações valem apena destacar:

<table>
<tr><td>Configuração</td><td>Implementação</td><td>Serve Para</td></tr>
<tr><td>eclipselink.ddl-generation</td><td>EclipseLink</td><td rowspan="3">Ativará a criação/atualização automática as tabelas ou até mesmo validar se o esquema do banco de dados é valido com relação ao mapeamento realizado com o JPA nas classes.</td></tr>
<tr><td>hibernate.hbm2ddl.auto</td><td>Hibernate</td></tr>
<tr><td>openjpa.jdbc.SynchronizeMappings</td><td>OpenJPA</td></tr>
<tr><td>eclipselink.logging.level</td><td>EclipseLink</td><td rowspan="3">Configurações que definem o nível de conteúdo a ser impresso pelo log.</td></tr>
<tr><td>org.hibernate.SQL.level<br>org.hibernate.type.level</td><td>Hibernate</td></tr>
<tr><td>openjpa.Log</td><td>OpenJPA</td></tr>
<tr><td colspan="3">Você irá encontrar na internet os valores permitidos para cada configuração de cada implementação.</td></tr>
</table>

# Capítulo 5: Definições de uma “Entity”. O que são anotações Lógicas e Físicas?

Para que o JPA possa mapear corretamente cada tabela do banco de dados para dentro de uma classe Java foi criado o conceito de Entidade (ou Entity em inglês). Uma Entity deve refletir exatamente a estrutura da tabela no banco de dados para que o JPA possa tomar conta do restante do trabalho.

Para que uma classe Java possa ser considerada uma Entity ela seguir os seguintes requisitos:

- Ter a anotação @Entity
- Ter um construtor público sem parâmetros
- Ter um campo anotado como @Id

Veja abaixo uma classe que segue todos os requisitos necessários:

```
1 import javax.persistence.Entity;
2 import javax.persistence.Id;
3
4 @Entity
5 public class Car {
6
7    public Car(){
8
9    }
10
11    public Car(int id){
12        this.id = id;
13    }
14
15    // Just to show that there is no need to have get/set when we talk about JPA Id
16    @Id
17    private int id;
18
19    private String name;
20
21    public String getName() {
22        return name;
23    }
24
25    public void setName(String name) {
26        this.name = name;
27    }
28 }
```

Sobre o código acima:

- A classe é anotada com @Entity logo acima de seu nome
- Existe um atributo que definimos como o id da nossa classe apenas anotando com @Id.*Sim, toda Entity tem que ter um campo que sirva como ID*. Em geral esse campo é um número sequencial, mas pode ser uma String e outros tipos de valores
- Um detalhe curioso é que não existe get/set para o ID. Na visão do JPA um ID é imutável então não existe a necessidade de alteração do ID
- É obrigatória à presença de um construtor que não aceite de parâmetros. Outros construtores podem ser adicionados.

De acordo com o código exibido acima são exibidas apenas anotações para definir uma Entity; o JPA irá procurar por uma tabela chamada CAR no banco de dados e por campos chamado ID e NAME. Por padrão o JPA utiliza o nome da classe e o nome de seus atributos para encontrar a tabela e sua estrutura no banco.

Segundo o livro “Pro JPA 2” podemos definir as anotações do JPA de dois modos, anotações físicas e anotações lógicas. As anotações físicas são aquelas que determinam o relacionamento entre a classe e o banco de dados. As anotações lógicas definem a modelagem da classe com relação ao sistema. Veja o código abaixo:

```
1  import java.util.List;
2
3  import javax.persistence.*;
4
5  @Entity
6  @Table(name = "TB_PERSON_02837")
7  public class Person {
8
9      @Id
10     @GeneratedValue(strategy = GenerationType.AUTO)
11     private int id;
12
13     @Basic(fetch = FetchType.LAZY)
14     @Column(name = "PERSON_NAME", length = 100, unique = true, nullable = false)
15     private String name;
16
17     @OneToMany
18     private List<Car> cars;
19
20     public String getName() {
21         return name;
22     }
23
24     public void setName(String name) {
25         this.name = name;
26     }
27 }
```

No código acima é possível encontrar as anotações: @Entity, @Id e @OneToMany (estudaremos mais adiante sobre essa anotação); essas anotações são consideradas como Anotações Lógicas. Note que essas anotações não definem nada relacionado ao banco de dados. Eles definem apenas configurações para o comportamento da classe como Entity.

No código acima também é possível ver as anotações @Table, @Column e @Basic. Essas anotações definem informações relacionadas ao banco de dados. Note que é possível definir o nome da tabela, nome da coluna e diversas outras configurações relacionados ao banco de dados físico. Esse tipo de anotação é conhecido por Anotações Físicas, pois estão diretamente ligadas ao banco de dados.

Não serão abordadas nesse post as possíveis opções para cada anotação, mas é bastante fácil de encontrar na internet a utilização para opção. Por exemplo: a anotação*@Column(name = "PERSON_NAME", length = 100, unique = true, nullable = false)*. As opções das anotações físicas são mais fáceis de entender pois parecem bastante com as configurações de um banco de dados.

# Capítulo 6: Gerando Id: Definição, Id por Identity ou Sequence

Como foi dito na página 5, toda Entity é obrigada a ter um ID. O JPA vem com a opção de geração de ID de um modo automático e prático.

Atualmente existem três modos para geração de um ID:

- Identity
- Sequence
- TableGenerator

É necessário entender que cada banco adota um modo de geração automática de id. Atualmente o Oracle e o Postgres trabalham com Sequence, o Sql Server e o MySQL trabalham com Identity. Não é possível forçar o banco de dados a trabalhar com uma geração de ids automática que ele não suporta.

Os tipos permitidos para ids simples são: byte/Byte, int/Integer, short/Short, long/Long, char/Character, String, BigInteger, java.util.Date e java.sql.Date.

## Identity

Esse é o tipo de geração automática mais simples que existe. Basta anotar o atributo id como abaixo:

```
1 import javax.persistence.Entity;
2 import javax.persistence.GeneratedValue;
3 import javax.persistence.GenerationType;
4 import javax.persistence.Id;
5
6 @Entity
7 public class Person {
8
9     @Id
10    @GeneratedValue(strategy = GenerationType.IDENTITY)
11    private int id;
12
13    private String name;
14
15    public String getName() {
16        return name;
17    }
18
19    public void setName(String name) {
```

```
20        this.name = name;
21    }
22 }
```

Quem controla qual será o próximo ID é o próprio banco de dados, sem atuação do JPA. É necessário que primeiro a entidade seja persistida, e após a realização o commit, uma consulta ao banco será realizada para descobrir qual será o ID da entidade em questão. Pode inclusive haver uma pequena perca de desempenho, mas nada alarmante.

Esse tipo de esquema de geração não permite que blocos de Ids sejam alocados para facilitar no desempenho. A alocação de ids blocos será explicada a seguir.

## Sequence

O tipo de geração de id por Sequence funciona conforme abaixo:

```
1 import javax.persistence.Entity;
2 import javax.persistence.GeneratedValue;
3 import javax.persistence.GenerationType;
4 import javax.persistence.Id;
5 import javax.persistence.SequenceGenerator;
6
7 @Entity
8 @SequenceGenerator(name = Car.CAR_SEQUENCE_NAME, sequenceName = Car.CAR_SEQUENCE_NAME,
initialValue = 10, allocationSize = 53)
9 public class Car {
10
11    public static final String CAR_SEQUENCE_NAME = "CAR_SEQUENCE_ID";
12
13    @Id
14    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = CAR_SEQUENCE_NAME)
15    private int id;
16
17    private String name;
18
19    public String getName() {
20        return name;
21    }
22
23    public void setName(String name) {
24        this.name = name;
25    }
26 }
```

Sobre o código acima:

- A anotação @SequenceGenerator define que existirá uma Sequence no banco com o nome descrito nela (atributo sequenceName). Essa Sequence pode ser utilizada por mais classes mas não é aconselhável. Caso a Sequence da classe Car também fosse utilizada pela classe House o valor dos Ids não seriam sequenciais. Ao cadastrar o primeiro carro, o id seria um. Ao cadastrar a primeira casa, o id seria dois.
- O valor name = Car.CAR_SEQUENCE_NAME define por qual nome a Sequence será "conhecida" dentro da aplicação. Em qualquer classe que for utilizada não é necessário declará-la novamente, basta utilizar o mesmo nome. Por isso que foi criado um atributo estático com esse valor.
- O valor sequenceName = Car.CAR_SEQUENCE_NAME aponta diretamente para a Sequence no banco de dados.
- O valor initialValue = 10 define qual será o valor do primeiro ID. É preciso ter bastante cuidado ao se utilizar essa configuração; após inserir o primeiro valor e reiniciar a aplicação, o JPA tentará novamente inserir um objeto com o mesmo initialValue.
- O valor allocationSize = 53 define qual o tamanho ids que ficarão alocados na memória do JPA. Funciona do seguinte modo: ao iniciar a aplicação o JPA irá alocar em sua memória a quantidade de ids determinada nesse valor e irá distribuir para cada nova entidade que for salva no banco. Nesse caso, os ids de iriam de do initialValue até + 53. E uma vez que esgote os 53 ids em memória o JPA iria buscar mais 53 ids e guardar na memória novamente. Essa alocação de ids em blocos permite um melhor desempenho pois o JPA não precisará ir ao banco para buscar qual foi o ID gerado para aquele objeto.
- A anotação @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = CAR_SEQUENCE_NAME) define o tipo de geração como Sequence e aponta para o nome do gerador que é definido na anotação @SequenceGenerator.

# Capítulo 7: Gerando Id: TableGenerator e Auto

## TableGenerator

A opção TableGenerator funciona do seguinte modo:

- Uma tabela é utilizada para armazenar o valor de cada chave.
- Essa tabela contém uma coluna com o nome da tabela e outra coluna com o valor atual da chave.
- Essa é a única solução (até o atual momento) que permite a portabilidade da aplicação entre banco de dados, sem a necessidade de alteração da geração de ID. Suponha que uma aplicação que utilize Sequence para gerar ID no banco de dados do Postgres e a mesma aplicação irá rodar em um banco SQL Server; essa aplicação terá que gerar um artefato diferente para cada banco de dados. Utilizando o TableGenerator a aplicação poderá ser utilizada em qualquer banco independente de qual o tipo de geração de ID.

Veja o código abaixo de como utilizar o TableGenerator:

```
1 import javax.persistence.*;
2
3 @Entity
4 public class Person {
5
6    @Id
7    @TableGenerator(name="TABLE_GENERATOR", table="ID_TABLE",
pkColumnName="ID_TABLE_NAME", pkColumnValue="PERSON_ID",
valueColumnName="ID_TABLE_VALUE")
8    @GeneratedValue(strategy = GenerationType.TABLE, generator="TABLE_GENERATOR")
9    private int id;
10
11    private String name;
12
13    public String getName() {
14        return name;
15    }
16
17    public void setName(String name) {
18        this.name = name;
19    }
20 }
```

O código acima vai utilizar a seguinte tabela no banco de dados (é possível que o JPA crie a tabela automaticamente, veja a configuração na página sobre o persistence.xml):

| | id_table_name<br>[PK] character varying(255) | id_table_value<br>bigint |
|---|---|---|
| 1 | PERSON_ID | 101 |
| * | | |

Sobre a anotação @TableGenerator é possível destacar:

- "name" => É o id do TableGenerator dentro da aplicação.
- "table" => Nome da tabela que irá conter as chaves de cada tabela.
- "pkColumnName" =>Nome da coluna que irá conter o nome de id. No código acima a coluna com o nome id_table_name será gerada.
- "valueColumnName" => Nome da coluna que irá conter os valores de cada id.
- "pkColumnValue" => Nome da tabela que será salvo. O valor default é o nome da classe + id. Em nosso caso, é PERSON_ID que é o mesmo descrito no código acima.
- initialValue, allocationSize => Apesar de não estarem presentes no exemplo acima, é possível utilizar essas configurações igual ao demonstrado na geração por Sequence, veja na página anterior.
- É possível declarar o TableGenerator em outras entidades sem afetar o sequencial de uma entidade como visto com Sequence (veja na outra página).

A melhor prática para essa abordagem é criar o table generator dentro do arquivo orm.xml, esse arquivo é utilizado para sobrescrever as configurações do JPA e está fora do escopo desse post.

## Auto

O modo Auto (de automático) permite que o JPA decida qual estratégia utilizar. Esse é o valor padrão e para utilizá-lo basta fazer:

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

O JPA irá escolher o padrão de geração do ID a ser seguido. Os padrões podem ser qualquer um dos 3 tipos já citados.

# Capítulo 8: Utilizando Chave Composta Simples

Uma chave simples é quando tempos apenas um campo como ID. Um id simples funciona como abaixo:

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

Uma chave composta é necessária quando precisamos de mais de um atributo para ser o identificador da entidade. Existem chaves compostas que são simples ou complexas. São chamadas de chaves compostas simples quando apenas os tipos do Java são utilizados como id (String, int, …). Na próxima página se encontra modos de mapear chave compostas complexas.

Existem dois modos de utilizar uma chave composta simples, utilizando a anotação @IdClass ou @EmbeddedId.

## @IdClass

Veja o código abaixo:

```
1 import javax.persistence.*;
2
3 @Entity
4 @IdClass(CarId.class)
5 public class Car {
6
7    @Id
8    private int serial;
9
10    @Id
11    private String brand;
12
13    private String name;
14
15    public String getName() {
```

```
16        return name;
17    }
18
19    public void setName(String name) {
20        this.name = name;
21    }
22
23    public int getSerial() {
24        return serial;
25    }
26
27    public void setSerial(int serial) {
28        this.serial = serial;
29    }
30
31    public String getBrand() {
32        return brand;
33    }
34
35    public void setBrand(String brand) {
36        this.brand = brand;
37    }
38 }
```

Sobre código acima:

- @IdClass(CarId.class) => Essa anotação indica que a classe CarId contém os campos considerados com Id.
- Todos os campos que são marcados com Id devem estar descritos na classe CarId.
- É possível também utilizar @GeneratedValue na chave composta neste tipo de chave composta. No exemplo acima é possível adicionar @GeneratedValue ao atributo serial.

Veja abaixo o código da classe CarId:

```
1 import java.io.Serializable;
2
3 public class CarId implements Serializable{
4
5    private static final long serialVersionUID = 343L;
6
7    private int serial;
8    private String brand;
9
10    // must have a default construcot
11    public CarId() {
12
13    }
14
15    public CarId(int serial, String brand) {
16        this.serial = serial;
```

```
17          this.brand = brand;
18      }
19
20      public int getSerial() {
21          return serial;
22      }
23
24      public String getBrand() {
25          return brand;
26      }
27
28      // Must have a hashCode method
29      @Override
30      public int hashCode() {
31          return serial + brand.hashCode();
32      }
33
34      // Must have an equals method
35      @Override
36      public boolean equals(Object obj) {
37          if (obj instanceof CarId) {
38              CarId carId = (CarId) obj;
39              return carId.serial == this.serial && carId.brand.equals(this.brand);
40          }
41
42          return false;
43      }
44  }
```

A classe CarId contém os campos listados na entidade Car que foram marcados como @Id.

Para que uma classe possa ser utilizada como id, ela deve:

- Ter um construtor público sem argumentos
- Implementar a interface Serializable
- Sobrescrever os métodos hashCode/equals

E para buscar um carro no banco de dados utilizando chave composta, basta fazer como abaixo:

```
1 EntityManager em = // get valid entity manager
2
3 CarId carId = new CarId(33, "Ford");
4
5 Car persistedCar = em.find(Car.class, carId);
6
7 System.out.println(persistedCar.getName() + " - " + persistedCar.getSerial());
```

## @Embeddable

O outro modo de mapear uma chave composta é:

```
1 import javax.persistence.*;
2
3 @Entity
4 public class Car {
5
6    @EmbeddedId
7    private CarId carId;
8
9    private String name;
10
11    public String getName() {
12        return name;
13    }
14
15    public void setName(String name) {
16        this.name = name;
17    }
18
19    public CarId getCarId() {
20        return carId;
21    }
22
23    public void setCarId(CarId carId) {
24        this.carId = carId;
25    }
26 }
```

Sobre o código acima:

- A classe de ID foi colocada dentro da classe Car.
- A anotação @EmbeddedId é utilizada para identificar a classe de ID
- Não existe mais a necessidade de utilizar a anotação @Id dentro da classe.

A classe id ficará como abaixo:

```
1 import java.io.Serializable;
2
3 import javax.persistence.Embeddable;
4
5 @Embeddable
6 public class CarId implements Serializable{
7
8    private static final long serialVersionUID = 343L;
9
10    private int serial;
11    private String brand;
```

```
12
13      // must have a default construcot
14      public CarId() {
15
16      }
17
18      public CarId(int serial, String brand) {
19          this.serial = serial;
20          this.brand = brand;
21      }
22
23      public int getSerial() {
24          return serial;
25      }
26
27      public String getBrand() {
28          return brand;
29      }
30
31      // Must have a hashCode method
32      @Override
33      public int hashCode() {
34          return serial + brand.hashCode();
35      }
36
37      // Must have an equals method
38      @Override
39      public boolean equals(Object obj) {
40          if (obj instanceof CarId) {
41              CarId carId = (CarId) obj;
42              return carId.serial == this.serial && carId.brand.equals(this.brand);
43          }
44
45          return false;
46      }
47  }
```

Sobre o código acima:

- A anotação @Embeddable é utilizada para habilitar a classe para ser chave composta.
- Os campos dentro da classe já serão considerados como ids.

Para que uma classe possa ser utilizada como id, ela deve:

- Ter um construtor público sem argumentos
- Implementar a interface Serializable
- Sobrescrever os métodos hashCode/equals

Para realizar consultas é possível utilizar o mesmo método de consulta que no exemplo do @IdClass.

# Capítulo 9: Utilizando Chave Composta Complexa

Podemos definir uma chave composta complexa quando o id de uma classe é composto de outras entidades.

Imagine por exemplo a entidade DogHouse (casa de cachorro) onde ela utiliza como id o id do próprio cachorro. Veja o código abaixo:

```
1 import javax.persistence.*;
2
3 @Entity
4 public class Dog {
5    @Id
6    private int id;
7
8    private String name;
9
10    public int getId() {
11        return id;
12    }
13
14    public void setId(int id) {
15        this.id = id;
16    }
17
18    public String getName() {
19        return name;
20    }
21
22    public void setName(String name) {
23        this.name = name;
24    }
25 }
```

```
1 import javax.persistence.*;
2
3 @Entity
4 public class DogHouse {
5
6     @Id
7     @OneToOne
8     @JoinColumn(name = "DOG_ID")
9     private Dog dog;
10
11    private String brand;
12
13    public Dog getDog() {
14        return dog;
15    }
```

```
16
17    public void setDog(Dog dog) {
18        this.dog = dog;
19    }
20
21    public String getBrand() {
22        return brand;
23    }
24
25    public void setBrand(String brand) {
26        this.brand = brand;
27    }
28 }
```

Sobre o código acima:

- A anotação @Id foi utilizada na entidade Dog para informar ao JPA que o id da entidade DogHouse devem ser o mesmo.
- Junto com a anotação do @Id encontra-se a anotação @OneToOne para indicar que além do id em comum, existe um relacionamento entre as classes. Mais informações sobre a anotação @OneToOne ainda nesse post.

Imagine um caso onde seja necessário acessar o id da classe DogHouse sem precisar passar pela classe Dog (dogHouse.getDog().getId()). O próprio JPA já providencia um modo de fazer sem a necessidade de utilizar da Lei de Demeter: http://uaihebert.com/?p=62

```
1  import javax.persistence.*;
2
3  @Entity
4  public class DogHouseB {
5
6     @Id
7     private int dogId;
8
9     @MapsId
10    @OneToOne
11    @JoinColumn(name = "DOG_ID")
12    private Dog dog;
13
14    private String brand;
15
16    public Dog getDog() {
17        return dog;
18    }
19
20    public void setDog(Dog dog) {
21        this.dog = dog;
22    }
23
24    public String getBrand() {
```

```
25        return brand;
26    }
27
28    public void setBrand(String brand) {
29        this.brand = brand;
30    }
31
32    public int getDogId() {
33        return dogId;
34    }
35
36    public void setDogId(int dogId) {
37        this.dogId = dogId;
38    }
39 }
```

Sobre o código acima:

- Existe um campo int anotado com @Id e não mais a classe Dog.
- A entidade Dog foi anotada com @MapsId. Essa anotação indica ao JPA que será utilizado o id da entidade Dog como id da classe DogHouse e que esse id desse ser refletido no atributo mapeado com @Id.
- O JPA não necessita que o atributo dogId esteja no banco de dados. O que acontece é que em tempo de execução ele irá entender que o dogId é igual a dog.getId().

Para finalizar esse assunto, como seria uma entidade com id composto com mais de uma entidade? Veja o código abaixo:

```
1  import javax.persistence.*;
2
3  @Entity
4  @IdClass(DogHouseId.class)
5  public class DogHouse {
6
7     @Id
8     @OneToOne
9     @JoinColumn(name = "DOG_ID")
10    private Dog dog;
11
12    @Id
13    @OneToOne
14    @JoinColumn(name = "PERSON_ID")
15    private Person person;
16
17    private String brand;
18
19    // get and set
20 }
```

Sobre o código acima:

- Veja que as duas entidades (Dog e Person) foram marcadas com a anotação @Id.
- É utilizada a anotação @IdClass para determinar uma classe de chave composta.

```
1  import java.io.Serializable;
2
3  public class DogHouseId implements Serializable{
4
5      private static final long serialVersionUID = 1L;
6
7      private int person;
8      private int dog;
9
10     public int getPerson() {
11         return person;
12     }
13
14     public void setPerson(int person) {
15         this.person = person;
16     }
17
18     public int getDog() {
19         return dog;
20     }
21
22     public void setDog(int dog) {
23         this.dog = dog;
24     }
25
26     @Override
27     public int hashCode() {
28         return person + dog;
29     }
30
31     @Override
32     public boolean equals(Object obj) {
33         if(obj instanceof DogHouseId){
34             DogHouseId dogHouseId = (DogHouseId) obj;
35             return dogHouseId.dog == dog && dogHouseId.person == person;
36         }
37
38         return false;
39     }
40 }
```

Sobre o código acima:

- Veja que a classe contém dois atributos inteiros apontando para as entidades do relacionamento.
- Note que o nome dos atributos são exatamente iguais aos nomes dos atributos. Se o atributo na entidade DogHouse fosse Person dogHousePerson, o nome dentro da classe de id teria que ser dogHousePerson ao invés de person.

Para que uma classe possa ser utilizada como id, ela deve:

- Ter um construtor público sem argumentos
- Implementar a interface Serializable
- Sobrescrever os métodos hashCode/equals

# Capítulo 10: Modos para obter um EntityManager

Existem dois modos de obter um EntityManager. Um é por injeção e outro através de um Factory.

O modo de injeção é o mais simples, quem irá injetar os dados é o próprio servidor. Abaixo segue um modo de como injetar um EntityManager:

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

Só foi preciso utilizar a anotação "*@PersistenceContext(unitName = "PERSISTENCE_UNIT_MAPPED_IN_THE_PERSISTENCE_XML")*". Entenda que para que a injeção funcione é necessário que sua aplicação rode em um ambiente JEE, ou seja, em um servidor de aplicativos como JBoss, Glassfish… Para que um Entity Manager seja injetado corretamente o arquivo persistence.xml deve estar no local correto, e deve-se ter (se necessário) um datasource corretamente configurado.

A injeção de EntityManager, atualmente, só funciona em servidores que suportam EJB. Tomcat e outros não irão realizar a injeção.

Para aplicações desktop ou quando se deseja controlar a transação manualmente mesmo em servidores JEE basta fazer conforme abaixo:

```
1 EntityManagerFactory emf =
Persistence.createEntityManagerFactory("PERSISTENCE_UNIT_MAPPED_IN_THE_PERSISTENCE_XML")
;
2 EntityManager entityManager = emf.createEntityManager();
3
4 entityManager.getTransaction().begin();
5
6 // do something
7
8 entityManager.getTransaction().commit();
9 entityManager.close();
```

Note que primeiro é necessário criar uma instância do EntityManagerFactory. Ele irá apontar para o PersistenceUnit mapeado no persistence.xml. Através do EntityManagerFactory é possível criar uma nova instância do EntityManager.

# Capítulo 11: Mapeando duas ou mais tabelas em uma entitade

Uma classe pode conter informações espalhadas por duas ou mais tabelas.

Para mapear uma Entity que tem dados em duas tabelas diferentes utilize as anotações abaixo:

```
1  import javax.persistence.*;
2
3  @Entity
4  @Table(name="DOG")
5  @SecondaryTables({
6     @SecondaryTable(name="DOG_SECONDARY_A",
pkJoinColumns={@PrimaryKeyJoinColumn(name="DOG_ID")}),
7     @SecondaryTable(name="DOG_SECONDARY_B",
pkJoinColumns={@PrimaryKeyJoinColumn(name="DOG_ID")})
8  })
9  public class Dog {
10    @Id
11    @GeneratedValue(strategy = GenerationType.AUTO)
12    private int id;
13
14    private String name;
15    private int age;
16    private double weight;
17
18    // get and set
19
20 }
```

Sobre o código acima:

- A anotação @SecondaryTable é utilizada para indicar em qual tabela os dados secundários serão encontrados. Caso os dados estejam em apenas uma tabela secundária, apenas essa anotação será suficiente.
- A anotação @SecondaryTables é utilizada para agrupar várias tabelas em uma classe. Ela agrupa várias anotações @SecondaryTable.

# Capítulo 12: Mapeando Heranças – MappedSuperclass

Caso seja necessário compartilhar atributos/métodos entre classes de uma herança, mas a super classe que terá os dados não é uma entidade, é necessário que essa classe tenha o conceito de uma MappedSuperclass.

Veja abaixo como ficaria o código de uma MappedSuperclass:

```
1  import javax.persistence.MappedSuperclass;
2
3  @MappedSuperclass
4  public abstract class DogFather {
5     private String name;
6
7     public String getName() {
8         return name;
9     }
10
11    public void setName(String name) {
12        this.name = name;
13    }
14
15 }
```

```
1  @Entity
2  @Table(name = "DOG")
3  public class Dog extends DogFather {
4
5     @Id
6     @GeneratedValue(strategy = GenerationType.AUTO)
7     private int id;
8
9     private String color;
10
11    public int getId() {
12        return id;
13    }
14
15    public void setId(int id) {
16        this.id = id;
17    }
18
19    public String getColor() {
20        return color;
21    }
22
```

```
23    public void setColor(String color) {
24        this.color = color;
25    }
26 }
```

Sobre o código acima:

- Na classe DogFather foi definida a anotação @MappedSuperclass. Com essa anotação toda a classe que herdar da classe DogFather herdará todos seus atributos. Esses atributos serão refletidos no banco de dados.
- A MappedSuperclass pode ser tanto abstrata quanto concreta.
- A classe Dog contém o gerador de ID, apenas Dog é uma Entity. DogFather não é uma entidade.

Existem algumas dicas e normas para utilização do MappedSuperclass:

- Uma MappedSuperclass não pode ser anotada com @Entity\@Table. Ela não é uma classe que será persistida. Seus atributos/métodos serão refletidos nas classes filhas.
- É boa prática sempre defini-la como abstrata. Ela é uma classe que não será consultada diretamente por JPQL ou Queries.
- Não podem ser persistidas, elas não são Entities.

Quando utilizar?

Se a classe não tiver a necessidade de ser acessada diretamente nas consultas ao banco de dados, pode-se usar a MappedSuperclass. Caso essa classe venha ter seu acesso direto por pesquisas, é aconselhável utilizar herança de Entity (veja nas próximas páginas).

# Capítulo 13: Mapeando Heranças – Single Table

No JPA é possível encontrar diversas abordagens quanto à herança. Em uma linguagem a Orientação a Objetos como o Java é comum encontrar heranças entre as classes onde todos os dados devem ser persistidos.

A estratégia Single Table utilizará apenas uma tabela para armazenar todas as classes da herança. Veja abaixo como utilizar:

```
1  import javax.persistence.*;
2
3  @Entity
4  @Table(name = "DOG")
5  @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
6  @DiscriminatorColumn(name = "DOG_CLASS_NAME")
7  public abstract class Dog {
8     @Id
9     @GeneratedValue(strategy = GenerationType.AUTO)
10    private int id;
11
12    private String name;
13
14    // get and set
15 }
```

```
1  import javax.persistence.Entity;
2
3  @Entity
4  @DiscriminatorValue("SMALL_DOG")
5  public class SmallDog extends Dog {
6    private String littleBark;
7
8     public String getLittleBark() {
9        return littleBark;
10    }
11
12    public void setLittleBark(String littleBark) {
13        this.littleBark = littleBark;
14    }
15 }
```

```
1  import javax.persistence.*;
2
3  @Entity
4  @DiscriminatorValue("HUGE_DOG")
5  public class HugeDog extends Dog {
```

```
6      private int hugePooWeight;
7
8      public int getHugePooWeight() {
9          return hugePooWeight;
10     }
11
12     public void setHugePooWeight(int hugePooWeight) {
13         this.hugePooWeight = hugePooWeight;
14     }
15 }
```

Sobre o código acima:

- @Inheritance(strategy = InheritanceType.SINGLE_TABLE) => essa anotação deve ser utilizada na classe de hierarquia mais alta (classe pai), também conhecida por "*root*". Essa anotação define qual será o padrão de hierarquia a ser utilizado, sendo que seu valor default é SINGLE_TABLE.
- @DiscriminatorColumn(name = "DOG_CLASS_NAME") => define qual o nome da coluna que irá conter a descrição de qual tabela a linha da tabela no banco de dados irá pertencer. Veja a imagem abaixo para ver como ficará a estrutura da tabela.
- @DiscriminatorValue => Define qual o valor a ser salvo na coluna descrita na anotação @DiscriminatorColumn. Veja a imagem abaixo para ver como ficará a estrutura da tabela.
- Note que o ID é definido apenas na classe que está mais acima da hierarquia. Não é permitido reescrever o id de uma classe na hierarquia.

| 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 |
| | | | | |

É possível utilizar o campo de descrição da classe com um inteiro também:

- @DiscriminatorColumn(name = "DOG_CLASS_NAME", discriminatorType = DiscriminatorType.INTEGER) => basta definir o tipo da coluna como Inteiro.
- @DiscriminatorValue("1″) => O valor a ser salvo deve ser alterado na Entity também. O número será salvo ao invés do texto.

# Capítulo 14: Mapeando Heranças – Joined

Ao utilizar o tipo de mapeamento por Joined é definido que cada entidade terá seus dados em uma tabela específica. Ao invés de existir apenas uma tabela com todos os dados, será utilizada uma tabela por classe para armazenar os dados.

Veja abaixo como funciona o modelo de herança por Joined:

```
1  import javax.persistence.*;
2
3  @Entity
4  @Table(name = "DOG")
5  @Inheritance(strategy = InheritanceType.JOINED)
6  @DiscriminatorColumn(name = "DOG_CLASS_NAME")
7  public abstract class Dog {
8
9     @Id
10    @GeneratedValue(strategy = GenerationType.AUTO)
11    private int id;
12
13    private String name;
14
15    // get and set
16 }
```

```
1  import javax.persistence.*;
2
3  @Entity
4  @DiscriminatorValue("HUGE_DOG")
5  public class HugeDog extends Dog {
6     private int hugePooWeight;
7
8     public int getHugePooWeight() {
9         return hugePooWeight;
10    }
11
12    public void setHugePooWeight(int hugePooWeight) {
13        this.hugePooWeight = hugePooWeight;
14    }
15 }
```

```
1  import javax.persistence.*;
2
3  @Entity
4  @DiscriminatorValue("SMALL_DOG")
```

```
5  public class SmallDog extends Dog {
6      private String littleBark;
7
8      public String getLittleBark() {
9          return littleBark;
10     }
11
12     public void setLittleBark(String littleBark) {
13         this.littleBark = littleBark;
14     }
15 }
```

Sobre o código acima:

- A anotação @Inheritance(strategy = InheritanceType.JOINED) está utilizando o valor JOINED.
- Veja abaixo como ficaram as tabelas criadas no banco de dados:

Tabela da classe Dog

| 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 |
| | | |

Tabela da classe HugeDog

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

Tabela da classe SmallDog

| id<br>[PK] integer | littlebark<br>character varying(255) |
|---|---|
| 1 | hau |
| 2 | hiu |
| 3 | hie |
| | |

Note nas imagens como foram persistidos os dados. Cada entidade teve sua informação distribuída em tabelas distintas; para essa estratégia o JPA utilizará uma tabela para cada classe sendo a classe concreta ou abstrata.

Na tabela Dog que contém os dados em comum de todas as classes; na tabela Dog existe um campo para descrever para qual entidade pertence cada linha.

# Capítulo 15: Mapeando Heranças – Table Per Concrete Class

A estratégia Table Per Concrete Class cria uma tabela por entidade concreta. Caso uma entidade abstrata seja encontrada em uma hierarquia a ser persistida, essas informações serão salvas nas classes concretas abaixo dela.

Veja o código abaixo:

```
1  import javax.persistence.*;
2
3  @Entity
4  @Table(name = "DOG")
5  @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
6  public abstract class Dog {
7
8      @Id
9      @GeneratedValue(strategy = GenerationType.AUTO)
10     private int id;
12
13     private String name;
14
15     // get and set
16 }
```

```
1  import javax.persistence.Entity;
2
3  @Entity
4  public class HugeDog extends Dog {
5     private int hugePooWeight;
6
7     public int getHugePooWeight() {
8         return hugePooWeight;
9     }
10
11    public void setHugePooWeight(int hugePooWeight) {
12        this.hugePooWeight = hugePooWeight;
13    }
14 }
```

```
1  import javax.persistence.Entity;
2
3  @Entity
4  public class SmallDog extends Dog {
5     private String littleBark;
```

```
6
7       public String getLittleBark() {
8           return littleBark;
9       }
10
11      public void setLittleBark(String littleBark) {
12          this.littleBark = littleBark;
13      }
14  }
```

Sobre o código acima:

- @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) => É indicado que uma tabela por entidade concreta será utilizado.
- Não existe mais a necessidade da anotação: @DiscriminatorColumn(name = “DOG_CLASS_NAME”). Cada classe concreta terá todos os seus dados, com isso não existirá mais uma tabela com dados que não pertençam a ela.
- Não existe mais a necessidade da anotação: @DiscriminatorValue pelo mesmo motivo explicado acima.
- Abaixo as imagens de como ficaram as estruturas das tabelas:

Tabela HugeDog

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

Tabela SmallDog

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

Note que os dados da classe abstrata Dog foram salvas dentro da tabela HugeDog e SmallDog.

# Capítulo 16: Prós e Contras dos mapeamentos das heranças

Infelizmente não existe o modo mais correto a seguir, cada abordagem tem sua vantagem e desvantagem. É necessário fazer uma análise por qual o melhor modelo a ser utilizado para cada situação:

| Abordagem | Prós | Contras |
|---|---|---|
| SINGLE_TABLE | A estrutura da tabela no banco de dados fica mais fácil de analisar. Apenas uma tabela é necessária | Não pode ter campos "não nulos". Imagine um caso onde a classe SmallDog tivesse o atributo não null "hairColor" no banco de dados. Ao persistir HugeDog que não tem esse atributo, uma mensagem de erro avisando que hairColor estava null seria enviada pelo banco de dados. |
| | Os atributos das entidades poderão ser encontrados em uma única tabela.<br>Todos os dados persistidos serão encontrados em uma única tabela. | |
| | Em geral tem um bom desempenho. | |
| JOINED | Cada entidade uma tabela contendo seus dados. | O insert é mais custoso. Um insert deverá ser feito para cada entidade utilizada na hierarquia. Se existe uma herança C extends B extends A, ao persistir um C, três inserts serão realizados – um em cada entidade mapeada. |

| | | |
|---|---|---|
| | Irá seguir o conceito OO aplicado no modelo de dados. | Quanto maior for a hierarquia, maior será o número de JOINs realizado em cada consulta. |
| TABLE_PER_CLASS | Quando a consulta é realizada em apenas uma entidade seu retorno é mais rápido. Em uma tabela constam apenas os dados da classe. | Colunas serão repetidas. As colunas referentes aos atributos das classes abstratas serão repetidas nas tabelas das classes filhas. |
| | | Quando uma consulta é realizada para trazer mais de uma entidade da herança, essa pesquisa terá um custo maior. Será utilizado UNION ou uma consulta por tabela. |

# Capítulo 17: Embedded Objects

Embedded Objects é um modo de organizar entidades que tem diferentes dados mesma tabela. Para deixar mais simples, imagine uma tabela person (pessoa) onde se encontram os dados referentes à pessoa (nome, idade) e ao seu endereço (rua, número da casa, bairro, cidade).

Veja abaixo a estrutura da tabela:

| 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 |
| | | | | | |

Como é possível ver, existem dados relacionados para as classes pessoa e casa. Veja como ficará a entidade Person (pessoa) e Address (endereço) ao utilizar o conceito de Embedded Objects:

```
1  import javax.persistence.*;
2
3  @Embeddable
4  public class Address {
5     @Column(name = "house_address")
6     private String address;
7
8     @Column(name = "house_color")
9     private String color;
10
11    @Column(name = "house_number")
13    private int number;
14
15    public String getAddress() {
16        return address;
17    }
18
19    public void setAddress(String address) {
20        this.address = address;
21    }
22
23    // get and set
24 }
```

```java
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
}
```

Sobre o código acima:

- A anotação @Embeddable (class Address) indica que essa classe será utilizada dentro de uma entidade, note que Address não é uma entidade. É apenas uma classe java que irá refletir dados do banco de dados de um modo organizado.
- Foi utilizada a anotação @Column dentro da classe ADDRESS para indicar qual o nome da coluna dentro da tabela person.
- A anotação @Embedded (da classe Person) indica que o JPA deve mapear os campos que estão dentro da classe Address como se os dados pertencessem à classe Person.
- A classe Address pode ser utilizada em outras entidades. Existem modos inclusive de sobrescrever o valor definido na anotação @Column em tempo de execução.

# Capítulo 18: ElementCollection – Como mapear uma lista de valores em uma classe

Em diversas modelagens se torna necessário o mapeamento de uma lista de valores que não são entidades para dentro de uma classe; por exemplo: pessoa tem e-mails, cachorro tem apelidos, etc.

Veja o código abaixo para realizar esse mapeamento:

```
1  import java.util.List;
2  import java.util.Set;
3
4  import javax.persistence.*;
5
6  @Entity
7  @Table(name = "person")
8  public class Person {
9
10    @Id
11    @GeneratedValue
12    private int id;
13    private String name;
14
15    @ElementCollection
16    @CollectionTable(name = "person_has_emails")
17    private Set<String> emails;
18
19    @ElementCollection(targetClass = CarBrands.class)
20    @Enumerated(EnumType.STRING)
21    private List<CarBrands> brands;
22
23    // get and set
24 }
```

```
1 public enum CarBrands {
2    FORD, FIAT, SUZUKI
3 }
```

Sobre o código acima:

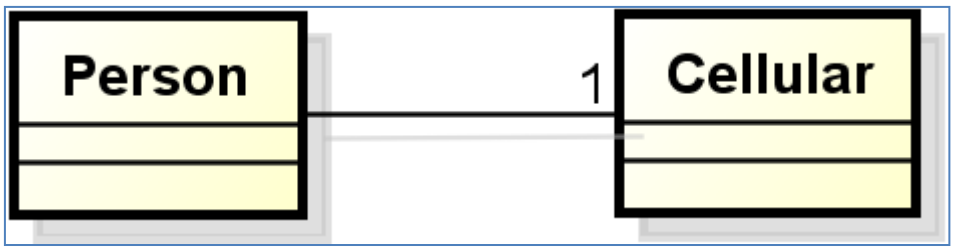
- Note que foi mapeado um Set<String> e um List<CarBrand>. O ElementCollection não está direcionado para uma Entidade, mas para atributos “simples” (um String e um Enum).
- A anotação @ElementCollection é utilizada para marcar um atributo como um item que poderá se repetir diversas vezes.
- A anotação @Enumerated(EnumType.STRING) foi utilizada no ElementCollection de enum. Ela define como o Enum será salvo no banco de dados, se por STRING ou ORDINAL (veja mais informações: http://uaihebert.com/?p=43).
- @CollectionTable(name = “person_has_emails”) => Indica qual será a tabela que terá a informação. Quando essa anotação não é encontrada o JPA irá definir um valor default. No caso do código acima, para o Enum de “List<CarBrand> brands” foi utilizada a tabela: person_brands. O nome padrão se dá juntando o nome da entidade atual mais o nome do atributo.

# Capítulo 19: OneToOne (Um para Um) Unidirecional e Bidirecional

É muito fácil de encontrar entidades que tenham algum relacionamento. Uma pessoa tem cachorros, cachorros têm pulgas, pulgas têm… hum… deixa para lá.

## Unidirecional

O relacionamento um para um é o mais simples. Vamos imagina a situação onde uma pessoa (Person) tem um celular (Cellular), e apenas a entidade Person terá acesso ao Cellular. Veja a imagem abaixo:




Veja como ficará a classe Person:

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Person {
5
6     @Id
7     @GeneratedValue
8     private int id;
9
10    private String name;
11
12    @OneToOne
13    @JoinColumn(name="cellular_id")
14    private Cellular cellular;
15
16    // get and set
17 }
```

```
1  import javax.persistence.*;
2
```

```
3  @Entity
4  public class Cellular {
5
6      @Id
7      @GeneratedValue
8      private int id;
9
10     private int number;
11
12     // get and set
13 }
```

Sobre o código acima:

- Em um relacionamento Unidirecional, apenas um lado conhece o outro. Note que a classe Person conhece Cellular, mas a classe Cellular não conhece Person. É possível fazer person.getCellular() mas não é possível fazer o inverso cellular.getPerson().
- Na entidade Person foi utilizado a anotação @OneToOne. Essa anotação indica ao JPA que existe um relacionamento entre essas entidades.

Todo o relacionamento necessita que umas das entidades seja a "dona desse relacionamento". Ser dona do relacionamento nada mais é do ter a chave estrangeira na tabela do banco de dados. No exemplo acima é possível ver na classe Person a utilização da anotação @JoinColumn. Essa anotação indica que a chave estrangeira ficará dentro da tabela person, fazendo com que a entidade Person seja a dona do relacionamento.

## Bidirecional

Para deixar esse relacionamento bidirecional é necessário alterar apenas a classe Cellular. Veja abaixo com ela ficou:

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Cellular {
5
6      @Id
7      @GeneratedValue
8      private int id;
9
10     private int number;
11
12     @OneToOne(mappedBy="cellular")
```

```
13    private Person person;
14
15    // get and set
16 }
```

Sobre o código acima:

- Veja que é utilizada a mesma anotação @OneToOne da entidade Person.
- Foi utilizado o atributo "mappedBy" na anotação @OneToOne. Esse atributo indica que a entidade Person é a dona do relacionamento; a chave estrangeira deve ficar na tabela Person e não na tabela Cellular.

E necessário que o desenvolvedor tenha sempre em mente que, para o correto funcionamento da aplicação, em um relacionamento é recomendado que só exista um dono do relacionamento. Se a anotação @OneToOne da entidade Cellular estivesse sem o mappedBy, o JPA trataria a classe Cellular como dona do relacionamento também. Não é aconselhável deixar sem o mappedBy nos dois lados do relacionamento ou colocar mappedBy nos dois lados do relacionamento.

O mappedBy deve apontar para o nome do atributo e não para o nome da classe.

## Não existe "auto relacionamento"

Para que um relacionamento OneToOne bidirecional funcione corretamente é necessário fazer como abaixo:

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

O JPA usa o princípio do Java de referência, uma classe precisa fazer referência para outra. O JPA não criará um relacionamento de modo automático para que o relacionamento exista dos dois lados, é necessário fazer o processo acima.

# Capítulo 20: OneToMany (um para muitos) e ManyToOne (muitos para um) Unidirecional e Bidirecional

O relacionamento OneToMany é utilizado quando uma entidade tem uma lista de outra entidade, um Cellular tem muitas Calls (ligações), mas uma Call (ligação) só pode ter um Cellular. O relacionamento OneToMany é representado por uma lista como atibuto, pois está relacionado a mais de um registro.

Vamos começar pelo lado ManyToOne:

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Call {
5
6      @Id
7      @GeneratedValue
8      private int id;
9
10     @ManyToOne
11     @JoinColumn(name = "cellular_id")
12     private Cellular cellular;
13
14     private long duration;
15
16     // get and set
17 }
```

Sobre o código acima:

- Foi utilizada a anotação @ManyToOne.
- Note que já foi utilizada a anotação @JoinColumn para definir quem será o dono do relacionamento.
- O lado do relacionamento que tiver a anotação @ManyToOne será sempre dominante. Não existe a opção de utilizar mappedBy dentro da anotação @ManyToOne

Para realizar um relacionamento bidirecional é necessário alterar a entidade Cellular (que foi criada na página anterior). Veja como ficará a entidade:

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

```
2
3  @Entity
4  public class Cellular {
5
6      @Id
7      @GeneratedValue
8      private int id;
9
10     @OneToOne(mappedBy = "cellular")
11     private Person person;
12
13     @OneToMany(mappedBy = "cellular")
14     private List<Call> calls;
15
16     private int number;
17
18     // get and set
19 }
```

Sobre o código acima:

- Foi utilizada a anotação @OneToMany. Essa anotação deve ser colocada sobre uma coleção.
- Foi utilizado o mappedBy para indicar que esse relacionamento não é o lado dominante.

Todo o relacionamento necessita que umas das entidades seja a "dona desse relacionamento". Ser dona do relacionamento nada mais é do ter a chave estrangeira na tabela do banco de dados. No exemplo acima é possível ver na classe Call a utilização da anotação @JoinColumn. Essa anotação indica que a chave estrangeira ficará dentro da tabela Call. O atributo mappedBy deve apontar para o nome do atributo e não para o nome da classe.

## Não existe "auto relacionamento"

Para que um relacionamento OneToMany/ManyToOne bidirecional funcione corretamente é necessário fazer como abaixo:

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

O JPA usa o princípio do Java de referência. Ele não criará um relacionamento de modo automático. Para que o relacionamento exista dos dois lados é necessário fazer o processo acima.

# Capítulo 21: ManyToMany (muitos para muitos) Unidirecional e Bidirecional

Um exemplo para um relacionamento ManyToMany poderia que uma pessoa pode ter vários cachorros e um cachorro pode ter várias pessoas (imagine um cachorro que more numa casa com 15 pessoas).

Para a abordagem ManyToMany é necessário utilizar uma tabela adicional para realizar a junção desse relacionamento. Teríamos uma tabela person (pessoa), uma tabela dog (cachorro) e uma tabela de relacionamento chamada person_dog. Tabela person_dog teria apenas o id da pessoa, e o id cachorro indicando qual pessoa tem qual cachorro.

Veja nas imagens abaixo de como ficará o banco de dados:

Tabela person

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

Tabela dog

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

Tabela person_dog

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

Note que na tabela person_dog contém apenas “ids”.

Veja como ficará a entidade Person:

```
1  import java.util.List;
2
3  import javax.persistence.*;
4
5  @Entity
6  public class Person {
7
8     @Id
9     @GeneratedValue
10    private int id;
11
12    private String name;
13
14    @ManyToMany
15    @JoinTable(name = "person_dog", joinColumns = @JoinColumn(name = "person_id"),
inverseJoinColumns = @JoinColumn(name = "dog_id"))
16    private List<Dog> dogs;
17
18    @OneToOne
19    @JoinColumn(name = "cellular_id")
20    private Cellular cellular;
21
22    // get and set
23 }
```

Sobre o código acima:

- A anotação @ManyToMany é utilizada para esse tipo de relacionamento.
- A anotação @JoinTable foi utilizada para definir qual tabela seria utilizada para realizar armazenas as entidades relacionadas. “name” define o nome da tabela. “joinColumns” define qual será o nome da coluna na tabela da entidade que é a dona do relacionamento. “inverseJoinColumns” define qual será o nome da coluna na tabela da entidade não dona do relacionamento.

A entidade Person tem o relacionamento com Dog unidirecional. Veja como ficará a classe Dog caso o relacionamento seja bidirecional:

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

```
2
3  import javax.persistence.*;
4
5  @Entity
6  public class Dog {
7
8      @Id
9      @GeneratedValue
10     private int id;
11
12     private String name;
13
14     @ManyToMany(mappedBy="dogs")
15     private List<Person> persons;
16
17     // get and set
18 }
```

É possível encontrar a notação @ManyToMany utilizando a opção mappedBy para determinar que a dona do relacionamento é a entidade Person.

Todo o relacionamento necessita que umas das entidades seja a “dona desse relacionamento”. No caso do relacionamento ManyToMany a classe dona do relacionamento é a classe que não estiver utilizando a opção mappedBy na anotação @ManyToMany. Caso nenhuma classe esteja marcada com mappedBy o JPA irá tratar cada classe marcada com ManyToMany como dona do relacionamento. O mappedBy deve apontar para o nome do atributo e não para o nome da classe.

## Não existe “auto relacionamento”

Para que um relacionamento ManyToMany/ManyToMany bidirecional funcione corretamente é necessário fazer como abaixo:

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

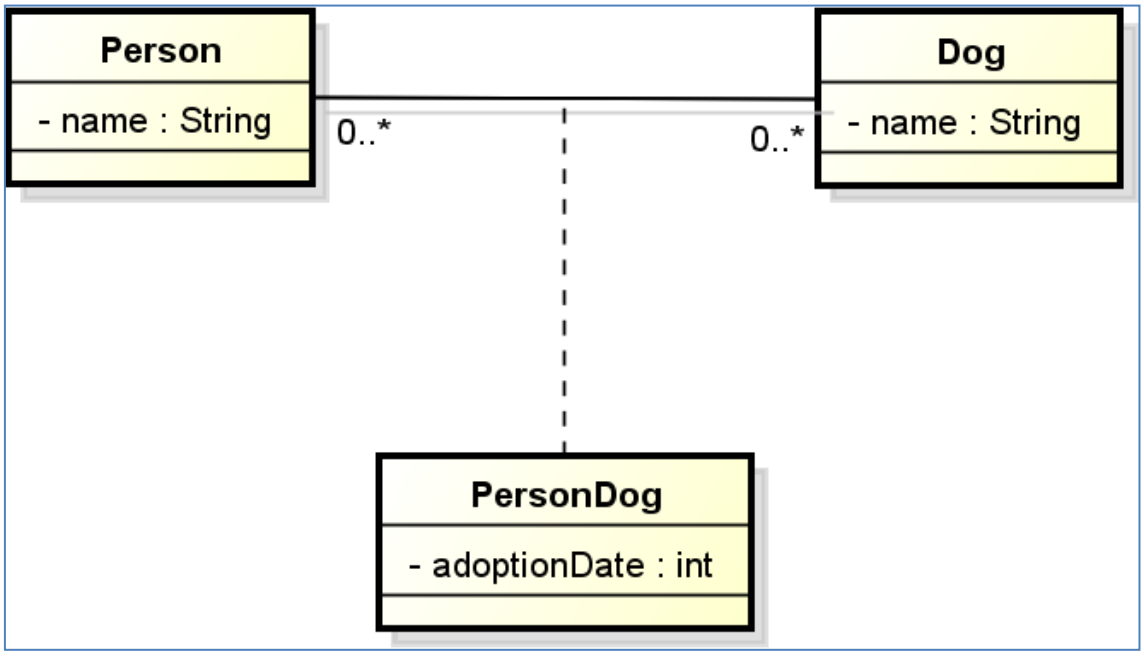
O JPA usa o princípio do Java de referência. Ele não criará um relacionamento de modo automático. Para que o relacionamento exista dos dois lados é necessário fazer o processo acima.

# Capítulo 22: ManyToMany com campos adicionais

Imagine que você tem a entidade Person em um relacionamento ManyToMany com Dog; e que toda vez que uma pessoa adotasse um cachorro a data de adoção fosse salva. Esse valor tem que estar ligado ao relacionamento, e não nas entidades cachorro e pessoa.

É para esse tipo de situação que existe a chamada Classe Associativa ou Entidade Associativa. Com as classes associativas é possível armazenar informações extras ao criar um relacionamento de ManyToMany.

A imagem abaixo explica melhor a necessidade dessa classe:




Para realizar esse mapeamento para entidades, basta fazer como abaixo:

```
1  import java.util.List;
2
3  import javax.persistence.*;
4
5  @Entity
6  public class Person {
7
```

```
8     @Id
9     @GeneratedValue
10    private int id;
11
12    private String name;
13
14    @OneToMany(mappedBy = "person")
15    private List<PersonDog> dogs;
16
17    // get and set
18 }
```

```
1  import java.util.List;
2
3  import javax.persistence.*;
4
5  @Entity
6  public class Dog {
7
8     @Id
9     @GeneratedValue
10    private int id;
11
12    private String name;
13
14    @OneToMany(mappedBy = "dog")
15    private List<PersonDog> persons;
16
17    // get and set
18 }
```

Foi utilizado um código simples com a anotação @OneToMany em conjunto com mappedBy. Note que não existe um relacionamento @ManyToMany diretamente entre as entidades, mas existe uma classe chamada PersonDog que faz a união entre as duas entidades.

Abaixo o código da classe PersonDog:

```
1  import java.util.Date;
2
3  import javax.persistence.*;
4
5  @Entity
6  @IdClass(PersonDogId.class)
7  public class PersonDog {
8
9     @Id
```

```
10    @ManyToOne
11    @JoinColumn(name="person_id")
12    private Person person;
13
14    @Id
15    @ManyToOne
16    @JoinColumn(name="dog_id")
17    private Dog dog;
18
19    @Temporal(TemporalType.DATE)
20    private Date adoptionDate;
21
22    // get and set
23 }
```

Existe nessa classe um relacionamento para cachorro e pessoa, e é possível encontrar um campo para armazenar a data da adoção. Existe uma classe que terá o ID entre o relacionamento PersonDogId:

```
1  import java.io.Serializable;
2
3  public class PersonDogId implements Serializable {
4
5     private static final long serialVersionUID = 1L;
6
7     private int person;
8     private int dog;
9
10    public int getPerson() {
11        return person;
12    }
13
14    public void setPerson(int person) {
15        this.person = person;
16    }
17
18    public int getDog() {
19        return dog;
20    }
21
22    public void setDog(int dog) {
23        this.dog = dog;
24    }
25
26    @Override
27    public int hashCode() {
28        return person + dog;
29    }
30
31    @Override
32    public boolean equals(Object obj) {
33        if(obj instanceof PersonDogId){
34            PersonDogId personDogId = (PersonDogId) obj;
35            return personDogId.dog == dog && personDogId.person == person;
36        }
37
```

```
38         return false;
39     }
40 }
```

Um detalhe interessante é que os atributos presentes na classe PersonDogId tem o mesmo nome dos atributos Person e Dog encontrados na entidade PersonDog. O nome deve ser exatamente igual ao atributo, é como o JPA consegue localizar corretamente as informações. Para mais informações sobre chave composta simples e chave composta complexa ver capítulos 08 e 09.

# Capítulo 23: Como funciona o Cascade? Para que serve o OrphanRemoval? Como tratar a org.hibernate.TransientObjectException

É bastante comum que duas ou mais entidades sofram alterações em uma transação. Ao alterar dados de uma pessoa, pode-se alterar nome, endereço, idade, cor do carro; apenas para as alterações citadas três classes poderiam ser afetadas Person, Car e Address.

Em geral são alterações simples e que poderiam se resumir ao código abaixo

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

Mas se o código abaixo fosse executado a famosa exceção org.hibernate.TransientObjectException apareceria:

```
1  EntityManager entityManager = // get a valid entity manager
2
3  Car car = new Car();
4  car.setName("Black Thunder");
5
6  Address address = new Address();
7  address.setName("Street A");
8
9  entityManager.getTransaction().begin();
10
11 Person person = entityManager.find(Person.class, 33);
12 person.setCar(car);
13 person.setAddress(address);
14
15 entityManager.getTransaction().commit();
16 entityManager.close();
```

Para o EclipseLink gera a seguinte mensagem de erro: "*Caused by: java.lang.IllegalStateException: During synchronization a new object was found through a relationship that was not marked cascade PERSIST*".

Mas o que significa dizer que um objeto é transiente? Ou que esse relacionamento não está marcado com cascade PERSIST?

O JPA funciona como um rastreador de toda entidade que for envolvida em uma transação. Uma entidade envolvido em uma transação é uma entidade que será salva, alterada, apagada. O JPA precisa saber quem é aquela entidade, de onde veio e para onde vai. Ao abrir uma transação toda entidade que é carregada do banco de dados está “attached”. O attached nada mais quer dizer que aquela entidade está dentro da transação, sendo monitorado pelo JPA. Uma entidade estará attached enquanto a transação estiver aberta (regra para aplicações J2SE ou aplicações que não utilizem EJB Stateful com Persistence Scope Extended); para ser considerada attached a entidade precisa ter vindo do banco de dados (por query, ou método find do entity manager, …), ou receber algum comando dentro de transação (merge, refresh).

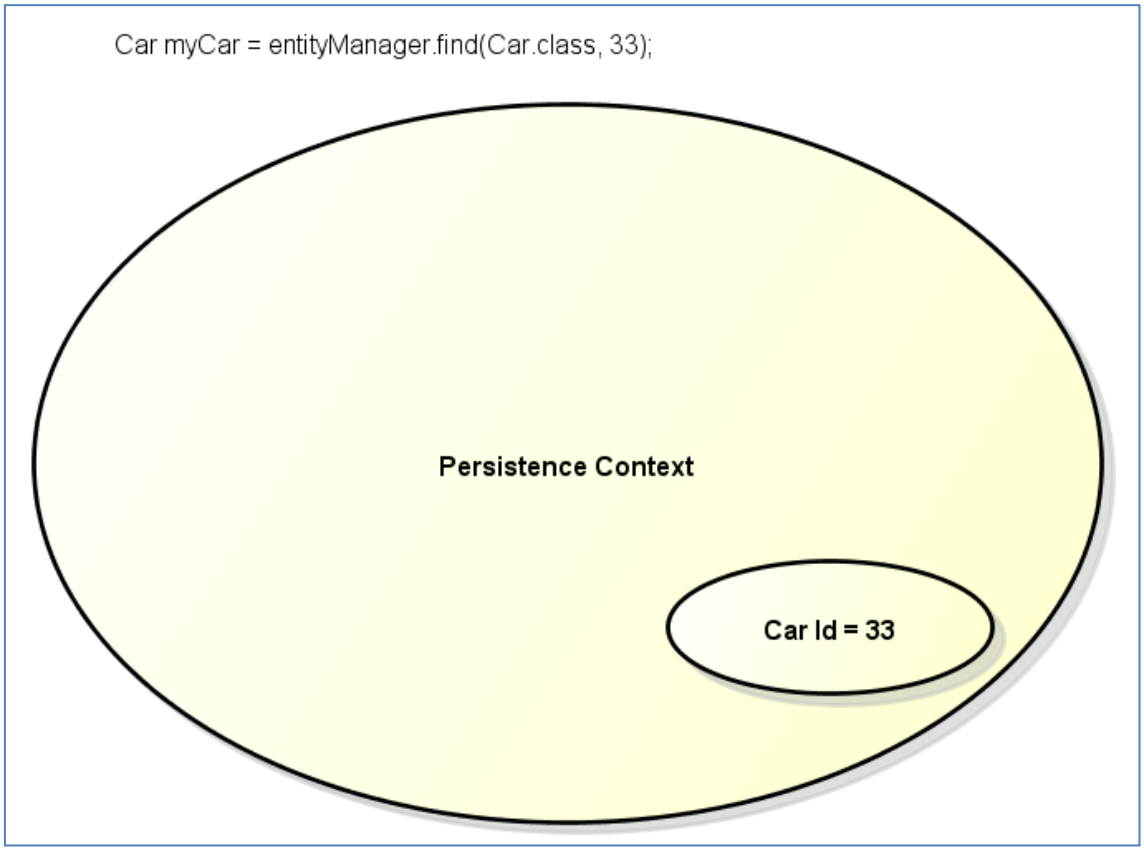
Observe o código abaixo:

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

A transação é aberta, uma alteração é realizada na entidade e que a transação recebe o commit. Não foi necessário realizar um update diretamente na entidade, bastou simplesmente realizar o commit da transação para que a alteração fosse salva no banco de dados. A alteração no carro foi salva no banco de dados por que a entidade está “attached”, qualquer alteração sofrida por uma entidade “attached” será refletida no banco de dados após o commit da transação ou o comando flush().

No código acima a entidade myCar foi localizada no banco de dados dentro de uma transação, por isso ela está attached aquele Persistence Context. Podemos definir Persistence Context como um local em que o JPA toma conta de todos os objetos, uma grande sacola. Veja a imagem abaixo que irá explicar melhor essa situação:

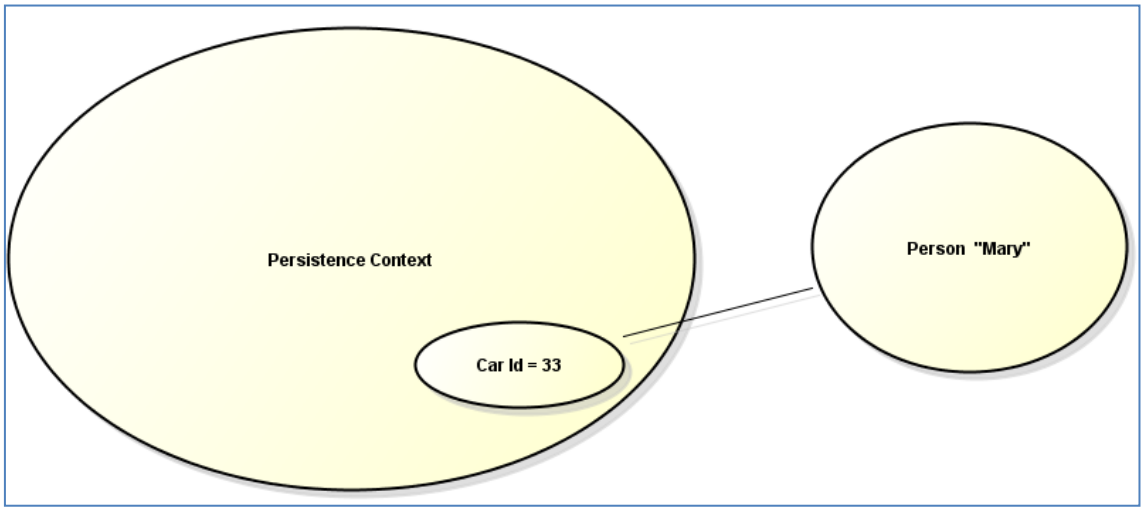

Persistence Context


Car myCar = entityManager.find(Car.class, 33);
Persistence Context
Car Id = 33

Note na imagem acima que após o carro ser localizado no banco de dados ele foi adicionado ao Persistence Context. Toda alteração realizada na entidade será monitorada pelo JPA. Após a transação ser finalizada (ou comando flush), o próprio JPA irá refletir essa alteração no banco de dados.

A exceção acontece quando relacionamos uma entidade com a outra. Veja o código e a imagem abaixo que irá explicar melhor a situação:

```
1  entityManager.getTransaction().begin();
2  Person newPerson = new Person();
3  newPerson.setName("Mary");
4  Car myCar = entityManager.find(Car.class, 33);
5  myCar.setOwner(newPerson);
6  entityManager. getTransaction().commit();
```




A entidade Car agora está relacionada com a entidade Person. O problema é que a pessoa se encontra fora do Persistence Context, note que a entidade Person não foi localizada no banco de dados ou “attached” à transação. Ao realizar o commit o JPA não consegue reconhecer que a pessoa é uma entidade nova, que ainda não existe no banco. Mesmo que o objeto pessoa fosse um objeto já salvo, ao vir de fora da transação (de um ManagedBean do JSF ou de uma Action do Struts por exemplo), ele ainda assim não seria reconhecido nesse Persistence Context. Um objeto fora do Persistence Context é chamado de detached.

Essa situação de uma entidade detached pode ocorrer em qualquer tipo de ação do JPA: INSERT, UPDATE, DELETE…

Para ajudar o desenvolvedor nessas situações o JPA criou a opção Cascade. Essa opção pode ser definidas anotações @OneToOne, @OneToMany e @ManyToMany. Existe um enum que contém todas as opções de cascade possível: javax.persistence.CascadeType.

O cascade tem as opções abaixo:

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

O Cascade tem por conceito de propagar a ação configurada no relacionamento. Veja o código abaixo:

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Car {
5
6     @Id
7     @GeneratedValue
8     private int id;
9
10    private String name;
11
12    @OneToOne(cascade = CascadeType.PERSIST)
13    private Person person;
14
15    // get and set
16 }
```

No código acima foi definido que o relacionamento @OneToOne com Person sofreria a ação Cascade.PERSIST toda vez que o comando entityManager.persist(car) for executado; toda vez que um carro for persistido o JPA realizará o “persist” também no relacionamento com a entidade Person.

A vantagem do Cascade é a propagação automaticamente da ação configurada nos relacionamentos da entidade.

Veja na tabela abaixo todas as opções de Cascade possíveis até hoje:

| Tipo | Ação | Disparado por |
|---|---|---|
| CascadeType.DETACH | Quando uma entidade for retirada do Persistence Context (o que provoca que ela esteja detached) essa ação será refletida nos relacionamentos. | Persistence Context finalizado ou por comando específico: entityManager.detach(), entityManager.clear(). |
| CascadeType.MERGE | Quando uma entidade tiver alguma informação alterada (update) essa ação será refletida nos relacionamentos. | Quando a entidade for alterada e a transação finalizada ou por comando específico: entityManager.merge(). |
| CascadeType.PERSIST | Quando uma entidade for nova e inserida no banco de dados essa ação será refletida nos relacionamentos. | Quando uma transação finalizada ou por comando específico: entityManager.persist(). |
| CascadeType.REFRESH | Quando uma entidade tiver seus dados sincronizados com o banco de dados essa ação será refletida nos relacionamentos. | Por comando específico: entityManager.refresh(). |
| CascadeType.REMOVE | Quando uma entidade for removida (apagada) do banco de dados essa ação será refletida nos relacionamentos. | Por comando específico: entityManager.remove(). |
| CascadeType.ALL | Quando qualquer ação citada acima for invocada pelo JPA ou por comando, essa ação será refletida no objeto. | Por qualquer ação ou comando listado acima. |

Uma vez que o cascade fosse definido, o código abaixo seria executado sem mensagem de erro:

```
1  entityManager.getTransaction().begin();
2  Person newPerson = new Person();
3  newPerson.setName("Mary");
4  Car myCar = entityManager.find(Car.class, 33);
5  myCar.setOwner(newPerson);
6  entityManager. getTransaction().commit();
```

Observações sobre o Cascade:

- É preciso ter bastante cuidado ao anotar um relacionamento com CascadeType.ALL. Uma vez que se a entidade fosse removida o seu relacionamento também seria removido do banco de dados. No exemplo acima, caso a anotação CascadeType.ALL fosse encontrado na entidade Car, ao seu excluir um carro a pessoa também seria excluída do banco de dados.
- CascadeType.ALL (ou opções únicas) pode causar lentidão em todas as ações realizadas na entidade. Caso a entidade tenha muitas listas um merge() poderia causar todas as listas a receberem merge().
- car.setOwner(personFromDB) => se a pessoa atribuída ao relacionamento for uma pessoa que já exista no banco mas está detached, o Cascade não irá ajudar nesse caso. Ao executar o comando entityManager.persist(car) o JPA tentará executar o persist nos objeto do relacionamento marcado com CascadeType.PERSIST (entityManager.persist(person)). Uma vez que a pessoa já existe no banco o JPA tentará inserir novamente e uma mensagem de erro será exibida. Seria necessário um objeto atualizado do banco de dados e o melhor modo de se fazer é utilizando o método getReferenceOnly(): http://uaihebert.com/?p=1137.

Para que o Cascade seja disparado pelo JPA é preciso sempre executar a ação na entidade que contém a opção do cascade configurada. Veja o código abaixo:

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Car {
5
6     @Id
7     @GeneratedValue
8     private int id;
9
10    private String name;
11
12    @OneToOne(cascade = CascadeType.PERSIST)
13    private Person person;
14
15    // get and set
16 }
```

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Person {
5
6     @Id
7     private int id;
8
9     private String name;
10
11    @OneToOne(mappedBy="person")
11    private Car car;
12
13    // get and set
14 }
```

Analisando o código acima o modo correto de persistir a entidade e acionar o cascade é:

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

Desse modo o JPA irá analisar a classe carro e procurar por relacionamentos onde o cascade deve ser aplicado. Caso o comando abaixo fosse executado o erro de objeto transiente seria exibido:

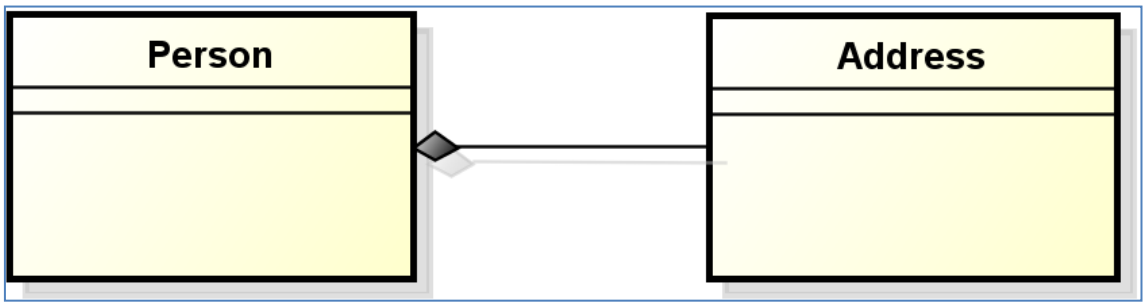
```
1  entityManager.persist(person);
```

*Lembre-se: o cascade só é acionado pelo JPA quando a ação for executada pela entidade em que o Cascade foi definido. No caso acima, apenas a classe Car definiu Cascade para Person, apenas a classe Car irá disparar a função de Cascade.*

## OrphanRemoval

Outra opção parecida ao CascadeType.REMOVE é o OrphanRemoval. A opção OrphanRemoval é conceitualmente aplicada em casos onde uma entidade é usada apenas em outra.

Imagine a situação onde a entidade Address só vai existir dentro de um Person:




Caso uma pessoa seja salva no banco de dados uma entidade endereço também será criada. Conceitualmente a entidade Address só existe quando uma Person é criada; ao excluir uma pessoa seu endereço deve ser removido também. É possível perceber que bastaria adicionar o CascadeType.REMOVE no relacionamento que o endereço seria removido juntamente com a pessoa, mas ainda existe outra diferença.

Para casos como Person e Address é que foi criado o a opção OrphanRemoval. O OrphanRemoval tem quase a mesma função do CascadeType.REMOVE, mas conceitualmente deve ser aplicado nos casos de Composição.

Veja o exemplo abaixo:

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Address {
5
6     @Id
7     @GeneratedValue
8     private int id;
9
10    private String name;
11
12    // get and set
13 }
```

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Person {
5
6     @Id
7     private int id;
8
9     private String name;
10
11    @OneToMany(orphanRemoval=true)
12    private Address address;
13
14    // get and set
15 }
```

Imagine que a entidade Address será utilizada com Person apenas e nenhuma outra entidade utiliza essa classe. Conceitualmente essa é o local ideal para se aplicar o OrphanRemoval.

Outro fator que difere o OrphanRemoval do CascadeType.REMOVE é que ao colocar o relacionamento para null a entidade será removida do banco de dados. Veja o código abaixo:

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

Quando a entidade Person fosse atualizada no banco de dados a entidade Address seria excluída do banco de dados.

Essa opção está disponível apenas os relacionamentos: @OneToOne e @OneToMany.

# Capítulo 24: Como excluir corretamente uma entidade com relacionamentos. Capturar entidades relacionadas no momento da exclusão de um registro no banco de dados

Ao excluir uma entidade do banco de dados o um erro de constraint pode aparecer, algo parecido com "java.sql.SQLIntegrityConstraintViolationException". Esse erro pode acontecer caso a entidade Person esteja relacionada a uma entidade Car, por exemplo, e o CascadeType.REMOVE não estiver configurado (ver página anterior sobre cascade).

Imagine que a pessoa de id 33 está relacionada com o carro de id 44 no banco de dados. O código abaixo causaria um erro de violação de chave:

```
1  entityManager.remove(person);
```

É possível resolver esse erro de algumas maneiras:

- CascadeType.REMOVE => Uma vez que a entidade for apagada o JPA ficará responsável de eliminar essa dependência. Na página anterior desse post mostra como fazer isso.
- OrphanRemoval => Uma solução que pode ser aplicada, mas com algumas restrições de relacionamentos. Na página anterior desse post mostra como fazer isso.
- Colocar o relacionamento para "*null*" antes de realizar a exclusão. Veja abaixo como fazer:

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

Existe um grande problema em localizar qual o relacionamento está pendente. O que pode ser feito é capturar, na mensagem de erro qual a classe com padrão. Essa solução não é

uma solução precisa pois será necessário capturar a mensagem de erro (em String) e localizar o nome da tabela. Infelizmente essa mensagem pode variar de implementação do JPA, JDBC driver e a linguagem em que foi desenvolvido.

# Capítulo 25: Criando um EntityManagerFactory por aplicação

Geralmente é utilizado um EntityManagerFactory por aplicação quando se controla a transação programaticamente, e não quando o servidor controla a transação. Essa solução é muito utilizada pois carregar um EntityManagerFactory na memória tem um custo relativamente alto, pois o JPA irá analisar todo o banco, validar Entities e demais informações. Realizar a opção de criar um novo EntityManagerFactory a cada transação fica inviável.

Abaixo segue um código que poderia ser utilizado para centralizar a criação de EntityManager na aplicação:

```
1  import javax.persistence.*;
2
3  public abstract class ConnectionFactory {
4      private ConnectionFactory() {
5      }
6
7     private static EntityManagerFactory entityManagerFactory;
8
9     public static EntityManager getEntityManager(){
10        if (entityManagerFactory == null){
11            entityManagerFactory =
Persistence.createEntityManagerFactory("MyPersistenceUnit");
12        }
13
14        return entityManagerFactory.createEntityManager();
15    }
16 }
```

# Capítulo 26: Entendendo o Lazy/Eager Load

Uma entidade pode ter um atributo que ocupe muitos bytes (um arquivo grande por exemplo) ou então ter uma lista enorme de atributos. Uma pessoa pode ter diversos carros, ou uma imagem atribuída ao seu perfil de 150MB.

Veja a classe abaixo:

```
1  import javax.persistence.*;
2
3  @Entity
4  public class Car {
5
6      @Id
7      @GeneratedValue
8      private int id;
9
10     private String name;
11
12     @ManyToOne
13     private Person person;
14
15     // get and set
16 }
```

```
1  import java.util.List;
2
3  import javax.persistence.*;
4
5  @Entity
6  public class Person {
7
8      @Id
9      private int id;
10
11     private String name;
12
13     @OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
14     private List<Car> cars;
15
16     @Lob
17     @Basic(fetch = FetchType.LAZY)
18     private Byte[] hugePicture;
19
```

```
20    // get and set
21 }
```

Sobre o código acima:

- Note que a coleção cars foi marcada com a opção FetchType.LAZY.
- Note que a imagem Byte[] hugePicture foi também marcada como Lazy.

O Lazy indica que o relacionamento/atributo não será carregado do banco de dados de modo natural. Ao fazer entityManager.find(Person.class, 33) a lista cars e a imagem hugePicture não serão carregados. Desse modo a quantidade de retorno de dados vinda do banco de dados será menor. A vantagem dessa abordagem é que a query fica mais leve, e o tráfego de dados pela rede fica menor.

Esses dados podem ser acessados após um get ser realizado na propriedade. Ao fazer person.getHugePicture() uma nova consulta será realizada no banco de dados para trazer essa informação.

Por padrão um atributo/campo simples será sempre EAGER. Para colocar um atributo/campo simples como Lazy basta utilizar a anotação @Basic como mostrado acima.

Os relacionamentos entre entidades tem um comportamento padrão:

- Relacionamentos que terminam em One serão EAGER por padrão: @OneToOne e @ManyToOne.
- Relacionamentos que terminam em Many serão LAZY por padrão: @OneToMany e @ManyToMany.

Para alterar algum relacionamento padrão, basta fazer como exibido no código acima.

Ao utilizar o Lazy por default o erro chamado “Lazy Initialization Exception” poderá acontecer. Esse erro acontece quando o atributo Lazy é acessado sem nenhuma conexão ativa. Veja esse post para mais detalhes sobre esse problema e como resolvê-lo : http://uaihebert.com/?p=1367

O problema de utilizar EAGER em todas as listas/atributos da entidade é que o custo da consulta no banco de dados pode aumentar muito. Caso todas as entidades Person tenham uma lista de 40.000 ações no log, imagina o custo para trazer 100.000 pessoas do banco de dados? É necessário ter bastante cautela ao escolher qual o tipo de “fetch” que será utilizado no atributo/relacionamento.

# Capítulo 27: Tratando o erro: “cannot simultaneously fetch multiple bags”

Esse erro acontece quando o JPA busca uma entidade com mais de uma coleção no banco de dados de modo EAGER. É comum encontrar códigos nas entidades onde um desenvolvedor para evitar o erro de LazyInitializationException (http://uaihebert.com/?p=1367) atribui a propriedade EAGER para todas as listas (Collection, List, …); ao consultar uma entidade no banco de dados, todos as listas já viriam carregadas.

Veja o código abaixo:

```
1  import java.util.List;
2
3  import javax.persistence.*;
4
5  @Entity
6  public class Person {
7
8     @Id
9     private int id;
10
11    private String name;
12
13    @OneToMany(mappedBy = "person", fetch = FetchType.EAGER, cascade =
CascadeType.ALL)
14    private List<Car> cars;
15
16    @OneToMany(fetch = FetchType.EAGER)
17    private List<Dog> dogs;
18
19    // get and set
20 }
```

Ao executar uma consulta para buscar uma pessoa da entidade acima a seguinte mensagem de erro aparecerá : “javax.persistence.PersistenceException: org.hibernate.HibernateException: cannot simultaneously fetch multiple bags”. Uma lista (List, Collection) também pode ser conhecida por “bag”.

Esse erro acontece pois o Hibernate tenta igualar o número de resultados vindo em uma consulta. Caso o SQL gerado retorne como resultado 2 linhas para a lista de dogs, e uma para car, o Hibernate irá repetir esse resultado para igualar as linhas da consulta no resultado. Veja a imagem abaixo para entender melhor o que acontece caso o seguinte código fosse executado entityManager.find(Person.class, 33):

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 |

O problema é que ao fazer isso, resultados repetidos apareceriam e quebraria o resultado correto da query. O resultado destacado em vermelho mostra o código que o Hibernate acaba por entender como repetido.

Existem 4 soluções (até o dia de hoje) que poderiam resolver essa situação:

- Utilizar java.util.Set ao invés das outras coleções => Com essa simples alteração esse erro poderá ser evitado.
- Utilizar EclipseLink = > é uma solução bastante radical, mas para os usuários de JPA que utilizam apenas as anotações do JPA a mudança será de baixo impacto.
- Utilizar FetchType.LAZY ao invés de EAGER => Essa solução é parcial, pois caso uma consulta seja feita e seja utiliza join fetch nas coleções, o problema volta a acontecer. Uma consulta que poderia gerar esse erro é: "*select p from Person p join fetch p.dogs d join fetch p.cars c*". Ao optar por essa abordagem o erro de LazyInitializationException (http://uaihebert.com/?p=1367) poderá acontecer.
- Utilizar @LazyCollection ou @IndexColumn do próprio Hibernate na coleção => É necessário conhecer bem a anotação @IndexColumn e suas implicações quando utilizada, pois seu comportamento varia se colocado em uma ponta do relacionamento ou na outra (não vem ao caso desse post explicar essa anotação).