

JPA

GUIA DEFINITIVO



JPA - Guia Definitivo

por Thiago Faria e Alexandre Afonso

2ª Edição, 29/03/2020

© 2020 AlgaWorks Softwares, Treinamentos e Serviços Ltda. Todos os direitos reservados.

Nenhuma parte deste livro pode ser reproduzida ou transmitida em qualquer forma, seja por meio eletrônico ou mecânico, sem permissão por escrito da AlgaWorks, exceto para resumos breves em revisões e análises.

AlgaWorks Softwares, Treinamentos e Serviços Ltda
www.algaworks.com
contato@algaworks.com

Siga-nos nas redes sociais e fique por dentro de tudo!



Sobre os autores

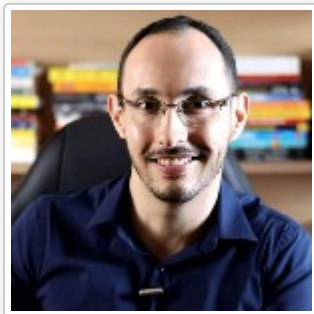


Thiago Faria de Andrade

Fundador da AlgaWorks, uma das principais escolas de desenvolvimento Java e front-end do Brasil. Autor de diversos livros e cursos de Java e front-end. Palestrante no JavaOne San Francisco em 2016, a maior conferência de Java do mundo. Programador desde os 14 anos de idade (1995), quando

desenvolveu o primeiro jogo de truco online e multiplayer (que ficou bem famoso na época).

LinkedIn: <https://www.linkedin.com/in/thiagofa>



Alexandre Afonso

Instrutor na AlgaWorks e programador Java há mais de 10 anos, com grande experiência em desenvolvimento de sistemas corporativos. Autor de diversos livros e cursos de Java.

LinkedIn: <https://www.linkedin.com/in/alexandreafon>

Antes de começar...

Antes que você comece a ler esse livro, nós gostaríamos de combinar algumas coisas com você, para que tenha um excelente aproveitamento do conteúdo. Vamos lá?

O que você precisa saber?

Você só conseguirá absorver o conteúdo desse livro se já conhecer pelo menos o básico de SQL, Java e Orientação a Objetos. Caso você ainda não domine Java e OO, pode ser interessante estudar por [nosso curso online](#).

Como obter ajuda?

Durante os estudos, é muito comum surgir várias dúvidas. Nós gostaríamos muito de te ajudar pessoalmente nesses problemas, mas infelizmente não conseguimos fazer isso com todos os leitores do livro, afinal, ocupamos grande parte do dia ajudando os alunos de cursos online na AlgaWorks.

Então, quando você tiver alguma dúvida e não conseguir encontrar a solução no Google ou com seu próprio conhecimento, nossa recomendação é que você poste na nossa Comunidade do Facebook. É só acessar:

<http://alga.works/comunidade/>

Como sugerir melhorias ou reportar erros no livro?

Se você encontrar algum erro no conteúdo desse livro ou se tiver alguma sugestão para melhorar a próxima edição, vamos ficar muito felizes se você puder nos dizer.

Envie um e-mail para livros@algaworks.com.

Onde encontrar uma cópia atualizada deste livro?

Nós desejamos continuar melhorando e atualizando o conteúdo deste livro. Por isso é importante você acessar o link <http://alga.works/livro-jpa> para baixar a

versão mais atualizada.

Ajude na continuidade desse trabalho

Escrever um livro como esse dá muito trabalho, por isso, esse projeto só faz sentido se muitas pessoas tiverem acesso a ele.

Ajude a divulgar esse livro para seus amigos que também se interessam por programação Java. Compartilhe no Facebook e Twitter!



Sumário

1 Introdução

1.1	O que é persistência de dados?	12
1.2	Mapeamento Objeto Relacional (ORM)	12
1.3	Porque usar ORM?	14
1.4	Java Persistence API e Hibernate	15

2 Primeiros passos com JPA

2.1	Criando e configurando o projeto	16
2.2	Criando o Domain Model	22
2.3	Implementando o equals() e hashCode()	24
2.4	Mapeamento básico	25
2.5	O arquivo persistence.xml	27
2.6	Gerando as tabelas do banco de dados	29
2.7	Definindo detalhes físicos de tabelas	31
2.8	Criando EntityManager	33
2.9	Persistindo objetos	34
2.10	Buscando objetos pelo identificador	37
2.11	Listando objetos	40
2.12	Atualizando objetos	42
2.13	Excluindo objetos	43

3 Gerenciando estados

3.1	Estados e ciclo de vida	45
3.2	Contexto de persistência	47
3.3	Sincronização de dados	49
3.4	Salvando objetos desanexados com merge()	52

4 Mapeamento

4.1	Identificadores	55
4.2	Chaves compostas	56
4.3	Enumerações	59
4.4	Propriedades temporais	61
4.5	Propriedades transientes	62
4.6	Objetos grandes	62
4.7	Objetos embutidos	66
4.8	Associações um-para-um	69
4.9	Associações muitos-para-um	76
4.10	Coleções um-para-muitos	78
4.11	Coleções muitos-para-muitos	80
4.12	Coleções de tipos básicos e objetos embutidos	86
4.13	Herança	91
4.14	Modos de acesso	102

5 Recursos avançados

5.1	Lazy loading e eager loading	106
5.2	Operações em cascata	113
5.3	Exclusão de objetos órfãos	117

5.4	Operações em lote	118
5.5	Concorrência e locking	120
5.6	Métodos de callback e auditores de entidades	124
5.7	Cache de segundo nível	128

6 Java Persistence Query Language

6.1	Introdução à JPQL	133
6.2	Consultas simples e iteração no resultado	137
6.3	Usando parâmetros nomeados	138
6.4	Consultas tipadas	139
6.5	Paginação	140
6.6	Projeções	141
6.7	Resultados complexos e o operador new	143
6.8	Funções de agregação	145
6.9	Queries que retornam um resultado único	146
6.10	Associações e joins	146
6.11	Queries nomeadas	152
6.12	Queries SQL nativas	154

7 Criteria API

7.1	Introdução e estrutura básica	156
7.2	Filtros e queries dinâmicas	157
7.3	Projeções	160
7.4	Funções de agregação	160
7.5	Resultados complexos, tuplas e construtores	161

7.6	Funções	163
7.7	Ordenação de resultado	165
7.8	Join e fetch	165
7.9	Subqueries	167
7.10	Metamodel	169

8 Conclusão

8.1	Próximos passos	173
-----	-----------------------	-----

Introdução

1.1. O que é persistência de dados?

Persistência de dados é a capacidade que sistemas de informação tem em armazenar e preservar o estado (os dados) para além do processo que o criou.

A maioria dos sistemas desenvolvidos em uma empresa precisa de dados persistentes, portanto persistência de dados é um conceito fundamental no desenvolvimento de aplicações.

Se um sistema de informação não preservasse os dados quando ele fosse encerrado, o sistema não seria prático e usual.

Quando falamos de persistência de dados com Java, normalmente falamos do uso de SGBDs (Sistema de Gerenciamento de Banco de Dados), porém existem diversas outras alternativas para persistir dados, como em arquivos XML, arquivos texto e etc.

1.2. Mapeamento Objeto Relacional (ORM)

Mapeamento objeto relacional (*object-relational mapping*, ORM, O/RM ou O/R mapping) é uma técnica de programação para “conversão” do modelo relacional para o modelo orientado a objetos.

Em banco de dados, entidades são representadas por tabelas, que possuem colunas que armazenam propriedades de diversos tipos. Uma tabela pode se associar com outras e criar relacionamentos diversos.

Em uma linguagem orientada a objetos, como Java, entidades são classes, e objetos dessas classes representam elementos que existem no mundo real.

Por exemplo, um sistema de faturamento possui a classe `NotaFiscal`, que no mundo real existe e todo mundo já viu alguma pelo menos uma vez, além de possuir uma classe que pode se chamar `Imposto`. Essas classes são chamadas de classes de domínio do sistema, pois fazem parte do negócio que está sendo desenvolvido.

Em banco de dados, podemos ter as tabelas `nota_fiscal` e também `imposto`, mas a estrutura do banco de dados relacional está longe de ser orientado a objetos, e por isso a ORM foi inventada para suprir a necessidade que os desenvolvedores têm de visualizar tudo como classes e objetos para programarem com mais facilidade.

Podemos comparar o modelo relacional com o modelo orientado a objetos conforme a tabela abaixo:

Modelo relacional	Modelo OO
Tabela	Classe
Linha	Objeto
Coluna	Atributo
-	Método
Chave estrangeira	Associação

Essa comparação é feita em todo o tempo quando estamos desenvolvendo usando algum mecanismo de ORM.

O mapeamento é feito usando metadados que descrevem a relação do modelo relacional do banco de dados e o modelo orientado a objetos.

Uma solução ORM consiste em uma API para executar operações CRUD simples em objetos de classes persistentes, uma linguagem ou API para especificar queries que se referem a classes e propriedades de classes, facilidades para especificar metadados de mapeamento e técnicas para interagir com objetos transacionais para identificarem automaticamente alterações realizadas, carregamento de associações por demanda e outras funções de otimização.

Em um projeto com ORM, a aplicação interage com a API da tecnologia de ORM e o modelo de classes de domínio e os códigos SQL/JDBC são abstraídos. Os comandos SQL são automaticamente gerados a partir dos metadados que relacionam os dois modelos.

1.3. Porque usar ORM?

Apesar de não parecer, uma implementação ORM é mais complexa que outro framework qualquer para desenvolvimento web, porém os benefícios de desenvolver utilizando esta tecnologia são grandes.

Códigos de acesso a banco de dados com queries SQL são chatos de escrever. A implementação ORM elimina muito do trabalho e deixa você se concentrar na lógica de negócio, possibilitando uma produtividade imensa.

A manutenibilidade de sistemas desenvolvidos com ORM costuma ser muito boa, pois o mecanismo faz com que menos linhas de código sejam necessárias. Além de facilitar o entendimento, menos linhas de código deixa o sistema mais fácil de ser alterado.

Existem outras razões que fazem com que um sistema desenvolvido utilizando um mecanismo ORM seja melhor de ser mantido.

Em sistemas com a camada de persistência desenvolvida usando JDBC e SQL puro, existe um trabalho na implementação para representar tabelas como objetos de domínio e alterações no banco de dados ou no modelo de domínio geram um esforço de refatoração que pode custar caro.

ORM abstrai sua aplicação do banco de dados e do dialeto SQL. Você pode

desenvolver um sistema usando um SGBD e migrar para um outro sem muito esforço, se for necessário.

1.4. Java Persistence API e Hibernate

A *Java Persistence API* (JPA) ou *Jakarta Persistence* (novo nome adotado pelo *Jakarta EE*) é uma especificação que oferece uma API de mapeamento objeto-relacional, ou seja, é a tecnologia padrão em Java para trabalhar com ORM.

O JPA até que possui um artefato JAR com algumas classes e interfaces no pacote `javax.persistence`, mas só isso não resolve nada. Não é a implementação ORM de fato.

Isso porque JPA é uma especificação. E o objetivo de uma especificação não é implementar o produto final, mas definir e padronizar a forma como os desenvolvedores resolvem determinada coisa.

Então, no caso de JPA, precisamos que alguém implemente a especificação.

Você mesmo pode fazer isso, se quiser. Mas felizmente já temos boas implementações no mercado e não precisamos nos preocupar com isso.

A implementação mais famosa é o Hibernate.

O legal de usar a especificação JPA é que, se no futuro você quiser migrar do Hibernate pra uma outra implementação, como por exemplo EclipseLink (que é a implementação de referência), o esforço é muito pequeno ou quase nulo.

E para usar JPA, não precisamos mais do que os objetos POJO (*Plain Old Java Objects*), ou seja, não é necessário nada muito especial para tornar os objetos persistentes.

Basta adicionar algumas anotações nas classes que representam as entidades do sistema e começar a persistir ou consultar objetos através da API do JPA.

Capítulo 2

Primeiros passos com JPA

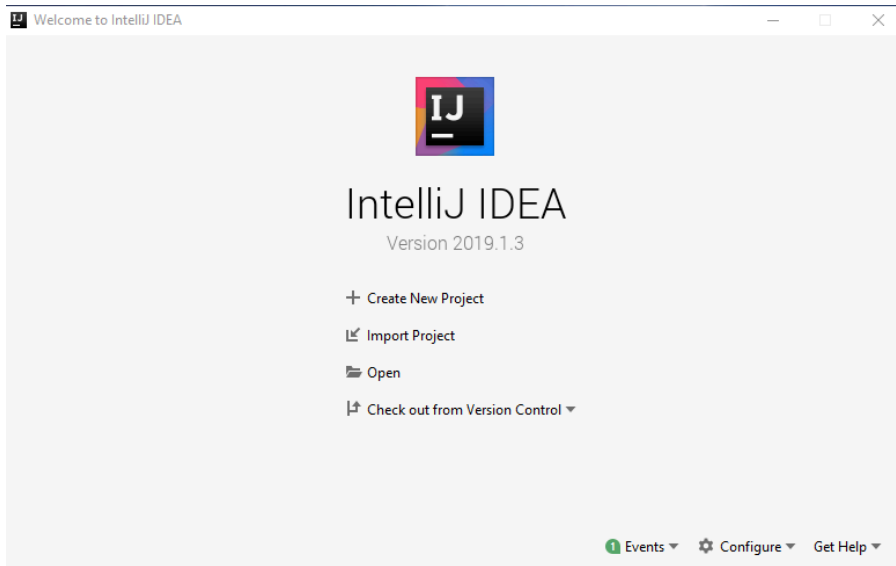
2.1. Criando e configurando o projeto

Usaremos a [IntelliJ IDEA](#) neste livro, que é uma IDE muito popular e excelente para desenvolvimento Java. Você pode usar [Eclipse](#), [NetBeans](#) ou qualquer outra de sua preferência.

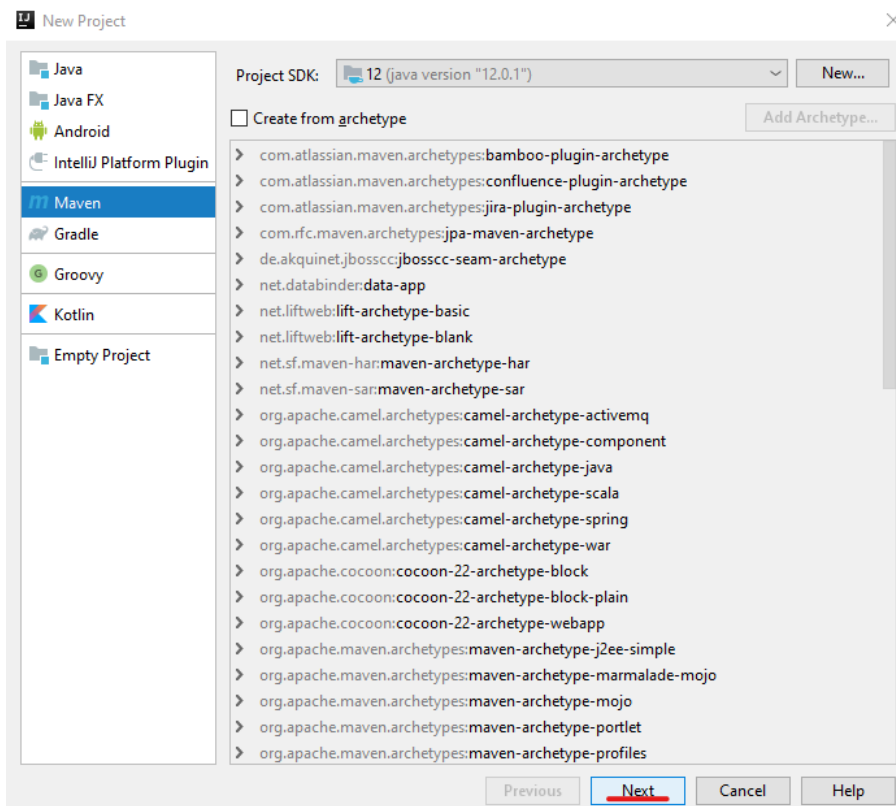
No projeto que vamos criar para os estudos, usaremos [Apache Maven](#), que é uma ferramenta para gerenciamento de dependências e build do projeto. O IntelliJ IDEA já possui suporte nativo ao Maven.

Os passos para criação do projeto com o IntelliJ são bem simples.

Na tela inicial dele, clique na opção *Create New Project*.



Depois, selecione *Maven* no menu lateral esquerdo e clique no botão *Next*.



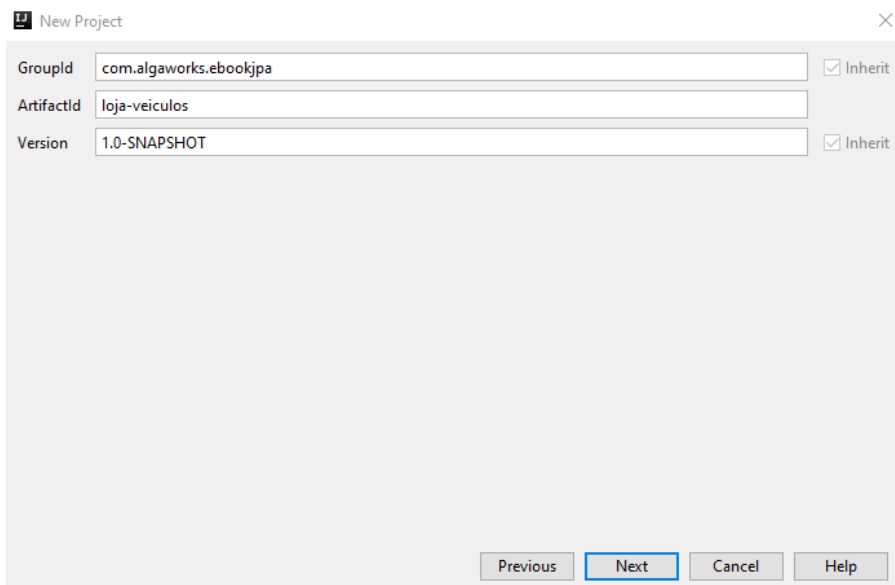
Preencha os campos *GroupId*, *ArtifactId* e *Version*.

O campo *GroupId* identifica o seu projeto entre todos os projetos Maven existentes, por isso você deve escolher um nome que seja único. Para garantir isso, o padrão é seguir as mesmas regras de definição de nomes de pacotes em Java, ou seja, o domínio ao contrário, seguido pelo nome do projeto.

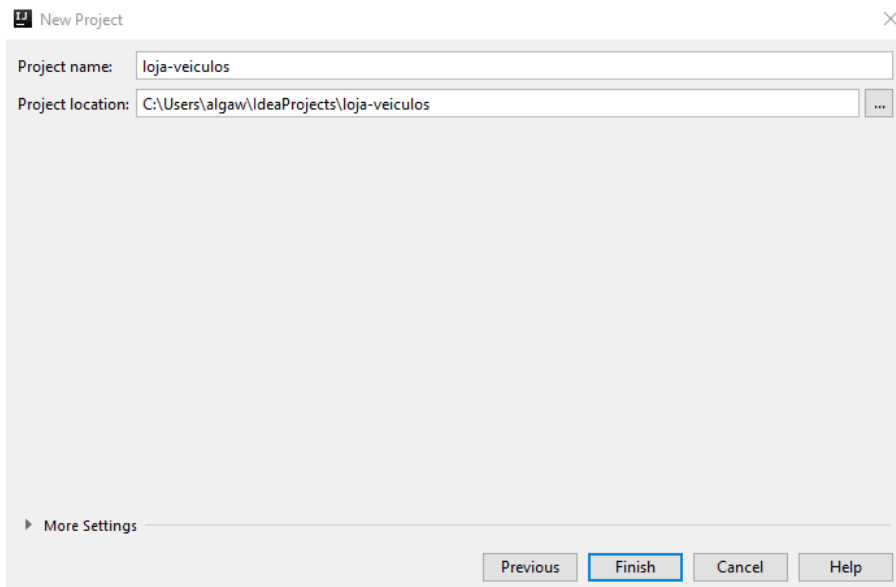
O campo *ArtifactId* identifica o nome do artefato do projeto, ou seja, o nome do JAR (sem extensão). Por padrão, usamos sempre letras minúsculas e separamos as palavras por hífens.

O campo *Version* se refere à versão do artefato. Geralmente, usamos números no formato X.Y ou X.Y.Z. Caso a versão ainda não tenha sido liberada (está em desenvolvimento), usamos o sufixo “-SNAPSHOT”.

Quando os campos estiverem preenchidos, clique no botão *Next* novamente.



Preencha o nome do projeto e escolha o diretório onde os arquivos serão armazenados e clique no botão *Finish*.



Quando o projeto for criado, um arquivo chamado *pom.xml* será criado no diretório raiz do projeto.

Esse arquivo contém informações sobre o projeto e detalhes de configurações usadas para o Maven fazer build do projeto.

Para começar a trabalhar com JPA, precisamos adicionar algumas dependências no *pom.xml*.

Abra o arquivo *pom.xml*, encontre a tag `<dependencies>` e adicione as duas dependências, conforme o código abaixo.

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.10.Final</version>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.16</version>
  </dependency>
</dependencies>
```

O Maven baixa automaticamente os JARs das dependências pela internet, do repositório central.

A dependência *hibernate-core* adiciona o Hibernate no nosso projeto, que é a implementação JPA que nós vamos usar.

Ao adicionar a implementação do Hibernate como dependência, estamos automaticamente adicionando também o JPA, que é uma dependência transitiva do Hibernate (o Hibernate depende do JPA, portanto “carrega” junto essa dependência, que é chamada de dependência transitiva).

Nós vamos usar o sistema de gerenciamento de banco de dados [MySQL Community](#), por isso adicionamos a dependência do Driver JDBC do MySQL no *pom.xml* também.

Veja como deve ficar o *pom.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.algaworks.ebookjpa</groupId>
  <artifactId>loja-veiculos</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.4.10.Final</version>
    </dependency>

    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.16</version>
    </dependency>
  </dependencies>

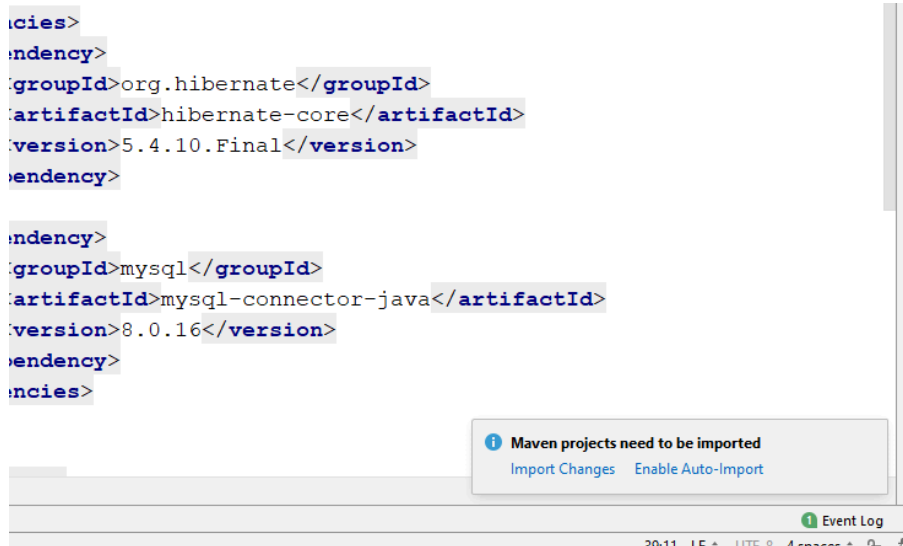
  <build>
```

```

<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
      <source>11</source>
      <target>11</target>
    </configuration>
  </plugin>
</plugins>
</build>
</project>

```

Após fazer a alteração no *pom.xml*, o IntelliJ emite um alerta solicitando que as mudanças sejam importadas. Clique no link *Import Changes*.



Feito isso, nosso projeto já está preparado para usar JPA.

Você quer aprender mais sobre Apache Maven?

Para aprender um pouco mais sobre o Apache Maven, assista esta videoaula gratuita no blog da AlgaWorks.

<https://blog.algaworks.com/comecando-com-apache-maven-em-projetos-java/>

2.2. Criando o Domain Model

Em nosso projeto de exemplo, queremos gravar e consultar informações de veículos de uma loja no banco de dados. Antes de qualquer coisa, precisamos criar nosso modelo de domínio para o negócio em questão.

Inicialmente, nosso sistema possuirá uma entidade chamada `Veiculo`, que é uma classe simples que representa o veículo à venda pela loja.

```
package com.algaworks.lojaveiculos.dominio;
```

```
import java.math.BigDecimal;
```

```
public class Veiculo {
```

```
    private Long codigo;  
    private String fabricante;  
    private String modelo;  
    private Integer anoFabricacao;  
    private Integer anoModelo;  
    private BigDecimal valor;
```

```
    public Long getCodigo() {  
        return codigo;  
    }
```

```
    public void setCodigo(Long codigo) {  
        this.codigo = codigo;  
    }
```

```

public String getFabricante() {
    return fabricante;
}

public void setFabricante(String fabricante) {
    this.fabricante = fabricante;
}

public String getModelo() {
    return modelo;
}

public void setModelo(String modelo) {
    this.modelo = modelo;
}

public Integer getAnoFabricacao() {
    return anoFabricacao;
}

public void setAnoFabricacao(Integer anoFabricacao) {
    this.anoFabricacao = anoFabricacao;
}

public Integer getAnoModelo() {
    return anoModelo;
}

public void setAnoModelo(Integer anoModelo) {
    this.anoModelo = anoModelo;
}

public BigDecimal getValor() {
    return valor;
}

public void setValor(BigDecimal valor) {
    this.valor = valor;
}
}

```

A classe Veiculo possui os seguintes atributos:

- codigo: identificador único do veículo

- fabricante: nome do fabricante do veículo
- modelo: descrição do modelo do veículo
- anoFabricacao: número do ano de fabricação do veículo
- anoModelo: número do ano do modelo do veículo
- valor: valor que está sendo pedido para venda do veículo

O atributo identificador (chamado de *codigo*) é referente à chave primária da tabela de veículos no banco de dados. Se existirem duas instâncias de *Veiculo* com o mesmo identificador, eles representam a mesma linha no banco de dados.

As classes de entidades podem seguir o estilo de JavaBeans, com métodos *getters* e *setters*. É obrigatório que estas classes possuam um construtor sem argumentos.

Como você pode ver, a classe *Veiculo* é um POJO, o que significa que podemos instanciá-la sem necessidade de containeres especiais.

Observação: nas seções e capítulos seguintes, podemos desenvolver novos exemplos para o projeto da loja sem acumular todos os recursos estudados, para não dificultar o entendimento de cada um dos recursos.

2.3. Implementando o `equals()` e `hashCode()`

Para que os objetos das entidades sejam diferenciados uns de outros, precisamos implementar os métodos `equals()` e `hashCode()`.

No banco de dados, as chaves primárias diferenciam registros distintos. Quando mapeamos uma entidade para uma tabela, devemos criar os métodos `equals()` e `hashCode()`, levando em consideração a forma em que os registros são diferenciados no banco de dados.

O IntelliJ IDEA e qualquer outra boa IDE possuem geradores do código desses métodos, que usam uma propriedade (ou várias, informadas por você) para criar o código-fonte. Veja como deve ficar a implementação dos métodos para a entidade *Veiculo*.

```
@Override
public boolean equals(Object o) {
```



```

    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Veiculo veiculo = (Veiculo) o;

    return codigo != null ? codigo.equals(veiculo.codigo)
        : veiculo.codigo == null;
}

@Override
public int hashCode() {
    return codigo != null ? codigo.hashCode() : 0;
}

```

Qual é a real importância dos métodos `equals` e `hashCode`?

Entender os motivos por trás da implementação desses métodos é muito importante. Leia um artigo sobre esse assunto no nosso blog.

<https://blog.algaworks.com/entendendo-o-equals-e-hashcode/>

2.4. Mapeamento básico

Para que o mapeamento objeto-relacional funcione, precisamos informar à implementação do JPA mais detalhes sobre como objetos da classe `Veiculo` devem se tornar persistentes, ou seja, como instâncias dessa classe podem ser gravadas e consultadas no banco de dados. Para isso, devemos anotar os *getters* ou os atributos, além da própria classe, com anotações do JPA.

```

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import java.math.BigDecimal;

@Entity
public class Veiculo {

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long codigo;

@Column
private String fabricante;

private String modelo;

private Integer anoFabricacao;

private Integer anoModelo;

private BigDecimal valor;

// getters, setters e métodos de negócio

// equals e hashCode
}

```

Você deve ter percebido que as anotações foram importadas do pacote `javax.persistence`. Dentro desse pacote estão todas as anotações padronizadas pela JPA.

A anotação `@Entity` diz que a classe é uma entidade JPA, que representa uma tabela do banco de dados.

As anotações nos atributos configuram a relação com as colunas da tabela do banco de dados.

A anotação `@Id` é usada para declarar o identificador da entidade, ou seja, representa a chave primária na tabela do banco de dados.

A anotação `@GeneratedValue` especifica que um valor será gerado automaticamente para este atributo. Definimos a estratégia de geração do identificador através da propriedade `strategy` com o valor `GenerationType.IDENTITY`.

A estratégia `IDENTITY` especifica que o valor será auto-incrementado pela própria coluna do banco de dados. Ou seja, ao inserir um novo registro, esperamos que o próprio banco de dados tome conta do incremento para o código do veículo.

A anotação `@Column` especifica que a propriedade da classe representa uma coluna na tabela do banco de dados.

Propositalmente, anotamos apenas uma propriedade com `@Column`, mas isso não quer dizer que apenas a propriedade anotada está mapeada. Ao omitir a anotação nas outras propriedades, o JPA faz o mapeamento automaticamente, o que significa que todas as propriedades da classe estão mapeadas para colunas.

2.5. O arquivo `persistence.xml`

O *`persistence.xml`* é um arquivo de configuração padrão da JPA. Ele deve ser criado no diretório *META-INF* da aplicação ou do módulo que contém as classes de entidade.

O arquivo *`persistence.xml`* define unidades de persistência, conhecidas como *`persistence units`*.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
  version="2.2">
  <persistence-unit name="AlgaWorks-PU">
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost/ebookjpa?
          createDatabaseIfNotExist=
            true&useTimezone=true&serverTimezone=UTC" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="1234" />
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.cj.jdbc.Driver" />

      <property name="javax.persistence.schema-generation.database.action"
        value="drop-and-create"/>

      <property name="javax.persistence.sql-load-script-source"
        value="META-INF/dados-iniciais.sql"/>

      <property name="hibernate.dialect"
```

```

        value="org.hibernate.dialect.MySQL8Dialect" />

        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
    </properties>
</persistence-unit>
</persistence>

```

O nome da unidade de persistência foi definido como `AlgaWorks-PU`. Precisaremos desse nome daqui a pouco, quando formos colocar tudo para funcionar.

Existem várias opções de configuração que podem ser informadas neste arquivo XML. Vejamos as principais propriedades que usamos em nosso arquivo de configuração:

- `javax.persistence.jdbc.url`: descrição da URL de conexão com o banco de dados
- `javax.persistence.jdbc.driver`: nome completo da classe do driver JDBC
- `javax.persistence.jdbc.user`: nome do usuário do banco de dados
- `javax.persistence.jdbc.password`: senha do usuário do banco de dados
- `javax.persistence.schema-generation.database.action`: Gera automaticamente o schema (as tabelas e relacionamentos) na inicialização da aplicação. Configuramos para *drop-and-create*, ou seja, para recriar as tabelas sempre. Logicamente, isso nos ajuda a ter o banco sempre limpinho para os nossos testes, mas em produção essa opção não deve ser usada.
- `javax.persistence.sql-load-script-source`: Executa um arquivo SQL para uma carga de dados na inicialização da *factory* do JPA, ou seja, a cada vez que criarmos uma instância de `EntityManagerFactory` como ainda veremos aqui no livro.
- `hibernate.dialect`: dialeto a ser usado na construção de comandos SQL
- `hibernate.show_sql`: informa se os comandos SQL devem ser exibidos na console (importante para *debug*, mas deve ser desabilitado em ambiente de produção)
- `hibernate.format_sql`: indica se os comandos SQL exibidos na console devem ser formatados (facilita a compreensão, mas pode gerar textos longos na saída)

Já aproveite para criar o arquivo *dados-iniciais.sql* no diretório *META-INF*. Abaixo você vai ter alguns comandos para incluir nele. Isso vai ajudar em nossos testes.

```
insert into Veiculo (codigo, fabricante, modelo, anoFabricacao,
anoModelo, valor)
values (1, 'Fiat', 'Toro', 2020, 2020, 107000);
insert into Veiculo (codigo, fabricante, modelo, anoFabricacao,
anoModelo, valor)
values (2, 'Ford', 'Fiesta', 2019, 2019, 42000);
insert into Veiculo (codigo, fabricante, modelo, anoFabricacao,
anoModelo, valor)
values (3, 'VW', 'Gol', 2019, 2020, 35000);
```

Cada comando de *insert* está ocupando três linhas, mas na hora de incluir no arquivo, cada comando deverá ocupar somente uma. Isso foi por uma questão de formatação aqui do livro.

Mas no seu arquivo será uma linha para cada comando. Caso contrário, você pode ter problemas na inicialização e seus registros não serão inseridos.

Como você viu acima, estamos usando o *drop-and-create*, ou seja, todas as vezes que iniciarmos o JPA, nossa base de dados será apagada, recriada e os comandos do nosso arquivo serão executados.

Teremos uma base sempre limpa e com os dados que informamos em nosso arquivo. Até o momento serão os três veículos acima.

2.6. Gerando as tabelas do banco de dados

Como ainda não temos a tabela representada pela classe *Veiculo* no banco de dados, precisamos criá-la.

O JPA pode fazer isso pra gente, graças à propriedade `javax.persistence.schema-generation.database.action` com valor `drop-and-create`, que incluímos no arquivo *persistence.xml*.

Precisamos apenas obter uma instância de *EntityManagerFactory* para dar um *start* no mecanismo do JPA. Assim, todas as tabelas mapeadas pelas entidades serão criadas.

```

import javax.persistence.Persistence;

public class Main {

    public static void main(String[] args) {
        EntityManagerFactory entityManagerFactory = Persistence
            .createEntityManagerFactory("AlgaWorks-PU");

        entityManagerFactory.close();
    }
}

```

O parâmetro do método `createEntityManagerFactory` deve ser o mesmo nome que informamos no atributo `name` da *tag* `persistence-unit`, no arquivo *persistence.xml*.

Ao executar o código, a tabela `Veiculo` é criada. Veja a saída na console:

```

...
INFO: HHH10001005: using driver [com.mysql.cj.jdbc.Driver] at URL ...
fev 04, 2020 12:52:44 AM org.hibernate.engine.jdbc.connections.in ...
INFO: HHH10001001: Connection properties: {password=****, user=root}
fev 04, 2020 12:52:44 AM org.hibernate.engine.jdbc.connections.in ...
INFO: HHH10001003: Autocommit mode: false
fev 04, 2020 12:52:44 AM org.hibernate.engine.jdbc.connections.in ...
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
fev 04, 2020 12:52:44 AM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL8Dialect
Hibernate:

    drop table if exists Veiculo
fev 04, 2020 12:52:45 AM org.hibernate.resource.transaction.backe ...
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [...
Hibernate:

    create table Veiculo (
        codigo bigint not null auto_increment,
        anoFabricacao integer,
        anoModelo integer,
        fabricante varchar(255),
        modelo varchar(255),
        valor decimal(19,2),

```

```

        primary key (codigo)
    ) engine=InnoDB

```

2.7. Definindo detalhes físicos de tabelas

A estrutura da tabela que foi criada no banco de dados é bastante simples. Veja que todas as propriedades da classe foram mapeadas automaticamente, mesmo sem incluir qualquer anotação JPA.

Com exceção da chave primária, todas as colunas aceitam valores nulos e as colunas fabricante e modelo aceitam até 255 caracteres.

Name: Veiculo									
Column	Datatype		PK	NN	UQ	BIN	UN	ZF	AI
codigo	BIGINT(20)	↕	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
anoFabricacao	INT(11)	↕	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
anoModelo	INT(11)	↕	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
fabricante	VARCHAR(255)	↕	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
modelo	VARCHAR(255)	↕	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
valor	DECIMAL(19,2)	↕	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Podemos definir melhor estes detalhes físicos no mapeamento de nossa entidade.

Além disso, queremos que nomes de colunas usem *underscore* na separação de palavras, que o nome da tabela seja `tab_veiculo` e a precisão da coluna `valor` seja 10, com 2 casas decimais.

```

import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

```

```

@Entity
@Table(name = "tab_veiculo")
public class Veiculo {

```

```

    @Id

```

```

@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long codigo;

@Column(length = 60, nullable = false)
private String fabricante;

@Column(length = 60, nullable = false)
private String modelo;

@Column(name = "ano_fabricacao", nullable = false)
private Integer anoFabricacao;

@Column(name = "ano_modelo", nullable = false)
private Integer anoModelo;

@Column(precision = 10, scale = 2, nullable = true)
private BigDecimal valor;

// getters e setters

// equals e hashCode

}

```

Vamos analisar as alterações que fizemos individualmente.

Especificamos o nome da tabela como `tab_veiculo`. Se não fizermos isso, o nome da tabela será considerado o mesmo nome da classe.

```

@Table(name = "tab_veiculo")
public class Veiculo {

```

Definimos o tamanho da coluna com 60 e com restrição *not null*.

```

@Column(length = 60, nullable = false)
private String fabricante;

```

Especificamos o nome da coluna como `ano_fabricacao` e com restrição *not null*. Se o nome da coluna não for especificado, por padrão, ela receberá o mesmo nome do atributo mapeado.

```

@Column(name = "ano_fabricacao", nullable = false)
private Integer anoFabricacao;

```


Atribuímos a precisão de 10 com escala de 2 casas na coluna de número decimal, especificando ainda que ela pode receber valores nulos.

```
@Column(precision = 10, scale = 2, nullable = true)
private BigDecimal valor;
```

Esses detalhes físicos são importantes se você estiver gerando as tabelas com o recurso de *schema generation* do JPA, que não é recomendado para produção (use apenas para protótipos e estudos).

Caso o *schema* do seu banco de dados seja gerado externamente, por alguma ferramenta de gerenciamento de migrações (como Flyway, por exemplo) ou até mesmo manualmente, você não precisa especificar todos os detalhes físicos das colunas.

Como tivemos alteração no nome da tabela e de algumas colunas, então, antes de testar, é preciso atualizar o nosso arquivo *dados-iniciais.sql*.

```
insert into tab_veiculo (codigo, fabricante, modelo, ano_fabricacao,
    ano_modelo, valor) values (1, 'Fiat', 'Toro', 2020, 2020, 107000);
insert into tab_veiculo (codigo, fabricante, modelo, ano_fabricacao,
    ano_modelo, valor) values (2, 'Ford', 'Fiesta', 2019, 2019, 42000);
insert into tab_veiculo (codigo, fabricante, modelo, ano_fabricacao,
    ano_modelo, valor) values (3, 'VW', 'Gol', 2019, 2020, 35000);
```

Agora, podemos executar o código `main` anterior, que instancia um `EntityManagerFactory`, e a tabela será recriada.

2.8. Criando EntityManager

Os sistemas que usam JPA precisam de apenas uma instância de `EntityManagerFactory`, que pode ser criada durante a inicialização da aplicação. Esta única instância pode ser usada por qualquer código que queira obter um `EntityManager`.

Um `EntityManager` é responsável por gerenciar entidades no contexto de persistência (que você aprenderá mais a frente). Através dos métodos dessa interface, é possível persistir, pesquisar e excluir objetos do banco de dados.

A inicialização de `EntityManagerFactory` pode demorar alguns segundos, por isso a instância dessa interface deve ser compartilhada na aplicação.

Precisaremos de um lugar para colocar a instância compartilhada de `EntityManagerFactory`, onde qualquer código tenha acesso fácil e rápido. Criaremos a classe `JpaUtil` para armazenar a instância em uma variável estática.

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JpaUtil {

    private static EntityManagerFactory factory;

    static {
        factory = Persistence.createEntityManagerFactory("AlgaWorks-PU");
    }

    public static EntityManager getEntityManager() {
        return factory.createEntityManager();
    }

    public static void close() {
        factory.close();
    }
}
```

Criamos um bloco estático para inicializar a fábrica de *Entity Manager*. Isso ocorrerá apenas uma vez, no carregamento da classe. Agora, sempre que precisarmos de uma `EntityManager`, podemos chamar:

```
EntityManager manager = JpaUtil.getEntityManager();
```

2.9. Persistindo objetos

Chegou a hora de persistir objetos, ou seja, inserir registros no banco de dados.

O código abaixo deve inserir um novo veículo na tabela do banco de dados. Se não funcionar, não se desespere, volte nos passos anteriores e encontre o que você fez de errado.

```

import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;

public class PersistindoVeiculo {

    public static void main(String[] args) {
        EntityManager manager = JpaUtil.getEntityManager();
        EntityTransaction tx = manager.getTransaction();
        tx.begin();

        Veiculo veiculo = new Veiculo();
        veiculo.setFabricante("Honda");
        veiculo.setModelo("Civic");
        veiculo.setAnoFabricacao(2020);
        veiculo.setAnoModelo(2020);
        veiculo.setValor(new BigDecimal(90500));

        manager.persist(veiculo);

        tx.commit();
        manager.close();
        JpaUtil.close();
    }
}

```

Execute o método `main` da classe `PersistindoVeiculo` e veja a saída no console.

```

Hibernate:
    insert
    into
        tab_veiculo
        (ano_fabricacao, ano_modelo, fabricante, modelo, valor)
    values
        (?, ?, ?, ?, ?)

```

O Hibernate gerou o SQL de inserção e nos mostrou na saída, pois configuramos isso no arquivo `persistence.xml`.

Consulte os dados da tabela `tab_veiculo` usando qualquer ferramenta de gerenciamento do seu banco de dados e veja que as informações que definimos nos atributos da instância do veículo, através de métodos *setters*, foram armazenadas.

```
mysql> use ebookjpa;
Database changed
mysql> select * from tab_veiculo;
```

codigo	ano_fabricacao	ano_modelo	fabricante	modelo	valor
1	2020	2020	Fiat	Toro	107000.00
2	2019	2019	Ford	Fiesta	42000.00
3	2019	2020	VW	Gol	35000.00
4	2020	2020	Honda	Civic	90500.00

```
4 rows in set (0.00 sec)
```

Agora vamos entender o que cada linha significa.

O código abaixo obtém um `EntityManager`, que é responsável por gerenciar o ciclo de vida das entidades.

```
EntityManager manager = JpaUtil.getEntityManager();
```

Agora iniciamos uma nova transação.

```
EntityTransaction tx = manager.getTransaction();
tx.begin();
```

Instanciamos um novo veículo e atribuímos alguns valores, chamando os métodos *setters*.

```
Veiculo veiculo = new Veiculo();
veiculo.setFabricante("Honda");
veiculo.setModelo("Civic");
veiculo.setAnoFabricacao(2020);
veiculo.setAnoModelo(2020);
veiculo.setValor(new BigDecimal(90500));
```

Executamos o método `persist`, passando a instância do veículo como parâmetro. Isso fará com que o JPA insira o objeto no banco de dados.

Não informamos o código do veículo, porque ele será obtido automaticamente através do *auto-increment* do MySQL.

```
manager.persist(veiculo);
```

Agora fazemos *commit* da transação, para efetivar a inserção do veículo no banco de dados.

```
tx.commit();
```

Finalmente, fechamos o EntityManager e o EntityManagerFactory.

```
manager.close();  
JpaUtil.close();
```

Viu? Não precisamos escrever nenhum código SQL e JDBC! Trabalhamos apenas com classes, objetos e a API de JPA.

É claro que, no final das contas, SQL e JDBC continuam sendo usados por baixo dos panos, pela implementação do JPA.

2.10. Buscando objetos pelo identificador

Podemos recuperar objetos através do identificador (chave primária) da entidade. O código abaixo busca um veículo com o código igual a 1.

Agora podemos executar o código abaixo, que faz uma consulta por um veículo de identificador 1.

```
public class BuscandoVeiculo1 {  
  
    public static void main(String[] args) {  
        EntityManager manager = JpaUtil.getEntityManager();  
  
        Veiculo veiculo = manager.find(Veiculo.class, 1L);  
  
        System.out.println("Veículo de código " + veiculo.getCodigo()  
            + " é um " + veiculo.getModelo());  
  
        manager.close();  
        JpaUtil.close();  
    }  
}
```

Note que consultamos a instância do veículo usando o método find, de EntityManager, passando como argumento o tipo da entidade e também o código do veículo.

```
Veiculo veiculo = manager.find(Veiculo.class, 1L);
```

O resultado na console é o seguinte:

Hibernate:

```
select
    veiculo0_.codigo as codigol_0_0_,
    veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
    veiculo0_.ano_modelo as ano_mode3_0_0_,
    veiculo0_.fabricante as fabrican4_0_0_,
    veiculo0_.modelo as modelo5_0_0_,
    veiculo0_.valor as valor6_0_0_
from
    tab_veiculo veiculo0_
where
    veiculo0_.codigo=?
```

Veículo de código 1 é um Toro

Veja que o SQL gerado possui a cláusula *where*, para filtrar apenas o veículo de código igual a 1.

Podemos também buscar um objeto pelo identificador usando o método `getReference`.

```
public class BuscandoVeiculo2 {

    public static void main(String[] args) {
        EntityManager manager = JpaUtil.getEntityManager();

        Veiculo veiculo = manager.getReference(Veiculo.class, 1L);
        System.out.println("Veículo de código " + veiculo.getCodigo()
            + " é um " + veiculo.getModelo());

        manager.close();
        JpaUtil.close();
    }
}
```

O resultado na console é o mesmo, deixando a impressão que os métodos `find` e `getReference` fazem a mesma coisa, mas na verdade, esses métodos possuem comportamentos um pouco diferentes.

O método `find` busca o objeto imediatamente no banco de dados, enquanto `getReference` só executa o SQL quando o objeto for usado pela primeira vez, ou

seja, quando invocamos um método *getter* da instância, desde que não seja o `getCodigo`.

Veja esse exemplo:

```
public class BuscandoVeiculo3 {

    public static void main(String[] args) {
        EntityManager manager = JpaUtil.getEntityManager();

        Veiculo veiculo = manager.getReference(Veiculo.class, 1L);
        System.out.println("Buscou veículo. Será que já executou o SELECT?");

        System.out.println("Veículo de código " + veiculo.getCodigo()
            + " é um " + veiculo.getModelo());

        manager.close();
        JpaUtil.close();
    }
}
```

A execução do código exibe na saída:

Buscou veículo. Será que já executou o SELECT?

Hibernate:

```
select
    veiculo0_.codigo as codigol_0_0_,
    veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
    veiculo0_.ano_modelo as ano_mode3_0_0_,
    veiculo0_.fabricante as fabrican4_0_0_,
    veiculo0_.modelo as modelo5_0_0_,
    veiculo0_.valor as valor6_0_0_
from
    tab_veiculo veiculo0_
where
    veiculo0_.codigo=?
```

Veículo de código 1 é um Toro

Note que o SQL foi executado apenas quando um *getter* foi invocado, e não na chamada de `getReference`.

2.11. Listando objetos

Agora você vai aprender como fazer consultas simples de entidades com a linguagem JPQL (*Java Persistence Query Language*).

Vamos aprofundar um pouco mais nessa linguagem em outro capítulo, porém já usaremos o básico para fazer consultas simples no banco de dados.

A JPQL é uma extensão da SQL, porém com características da orientação a objetos. Com essa linguagem, não referenciamos tabelas do banco de dados, mas apenas entidades de nosso modelo.

Quando fazemos pesquisas em objetos, não precisamos selecionar as colunas do banco de dados, como é o caso da SQL. O código em SQL a seguir:

```
select * from veiculo
```

Fica da seguinte forma em JPQL:

```
select v from Veiculo v
```

A sintaxe acima em JPQL significa que queremos buscar os objetos da entidade Veiculo.

Execute agora o código abaixo.

```
import com.algaworks.lojaveiculos.dominio.Veiculo;  
import com.algaworks.lojaveiculos.util.JpaUtil;  
  
import javax.persistence.EntityManager;  
import javax.persistence.Query;  
import java.util.List;  
  
public class ListandoVeiculos {  
  
    public static void main(String[] args) {  
        EntityManager manager = JpaUtil.getEntityManager();  
  
        Query query = manager  
            .createQuery("select v from Veiculo v");  
        List<Veiculo> veiculos = query.getResultList();  
    }  
}
```



```

    for (Veiculo veiculo : veiculos) {
        System.out.println(veiculo.getCodigo() + " - "
            + veiculo.getFabricante() + " "
            + veiculo.getModelo() + ", ano "
            + veiculo.getAnoFabricacao() + "/"
            + veiculo.getAnoModelo() + " por "
            + "R$" + veiculo.getValor());
    }

    manager.close();
    JpaUtil.close();
}
}

```

O resultado na console foi o seguinte:

```

Hibernate:
    select
        veiculo0_.codigo as codigol_0_,
        veiculo0_.ano_fabricacao as ano_fabr2_0_,
        veiculo0_.ano_modelo as ano_mode3_0_,
        veiculo0_.fabricante as fabrican4_0_,
        veiculo0_.modelo as modelo5_0_,
        veiculo0_.valor as valor6_0_
    from
        tab_veiculo veiculo0_
1 - Fiat Toro, ano 2020/2020 por R$107000.00
2 - Ford Fiesta, ano 2019/2019 por R$42000.00
3 - VW Gol, ano 2019/2020 por R$35000.00

```

A consulta SQL foi gerada baseado nas informações do mapeamento, e todos os veículos da tabela foram listados.

A única novidade no código-fonte que usamos são as seguintes linhas:

```

Query query = manager.createQuery("select v from Veiculo v");
List<Veiculo> veiculos = query.getResultList();

```

Veja que criamos uma *query* com a JPQL e atribuímos na variável *query*. Depois executamos o método `getResultList` e obtivemos uma lista de veículos.

As IDEs, como o IntelliJ IDEA ou Eclipse, podem mostrar um alerta de *Type safety* em `query.getResultList()`. Por enquanto você pode ignorar isso, porque

em breve você vai aprender sobre TypedQuery e isso será resolvido.

2.12. Atualizando objetos

Os atributos de entidades podem ser manipulados diretamente ou através dos métodos da classe e todas as alterações serão detectadas e persistidas automaticamente, quando o contexto de persistência for “descarregado” para o banco de dados.

Vamos a um exemplo:

```
public class AtualizandoVeiculo {

    public static void main(String[] args) {
        EntityManager manager = JpaUtil.getEntityManager();
        EntityTransaction tx = manager.getTransaction();
        tx.begin();

        Veiculo veiculo = manager.find(Veiculo.class, 1L);

        System.out.println("Valor atual: " + veiculo.getValor());
        veiculo.setValor(veiculo.getValor().add(new BigDecimal(500)));
        System.out.println("Novo valor: " + veiculo.getValor());

        tx.commit();
        manager.close();
        JpaUtil.close();
    }
}
```

O código acima executa o comando *select* no banco de dados para buscar o veículo de código 1, imprime o valor atual do veículo, atribui um novo valor (soma 500,00 reais) e imprime um novo valor.

Veja que não precisamos chamar nenhum método para a atualização no banco de dados. A alteração foi identificada automaticamente e refletida no banco de dados, através do comando SQL *update*.

A saída abaixo foi apresentada na console:

```

Hibernate:
    select
        veiculo0_.codigo as codigo1_0_0_,
        veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
        veiculo0_.ano_modelo as ano_mode3_0_0_,
        veiculo0_.fabricante as fabrican4_0_0_,
        veiculo0_.modelo as modelo5_0_0_,
        veiculo0_.valor as valor6_0_0_
    from
        tab_veiculo veiculo0_
    where
        veiculo0_.codigo=?
Valor atual: 107000.00
Novo valor: 107500.00
Hibernate:
    update
        tab_veiculo
    set
        ano_fabricacao=?,
        ano_modelo=?,
        fabricante=?,
        modelo=?,
        valor=?
    where
        codigo=?

```

2.13. Excluindo objetos

A exclusão de objetos é feita chamando o método `remove` de `EntityManager`, passando como parâmetro o objeto da entidade.

```

public class ExcluindoVeiculo {

    public static void main(String[] args) {
        EntityManager manager = JpaUtil.getEntityManager();
        EntityTransaction tx = manager.getTransaction();
        tx.begin();

        Veiculo veiculo = manager.find(Veiculo.class, 1L);

        manager.remove(veiculo);
    }
}

```

```

        tx.commit();
        manager.close();
        JpaUtil.close();
    }
}

```

O resultado na console é o seguinte:

Hibernate:

```

select
    veiculo0_.codigo as codigo1_0_0_,
    veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
    veiculo0_.ano_modelo as ano_mode3_0_0_,
    veiculo0_.fabricante as fabrican4_0_0_,
    veiculo0_.modelo as modelo5_0_0_,
    veiculo0_.valor as valor6_0_0_
from
    tab_veiculo veiculo0_
where
    veiculo0_.codigo=?

```

Hibernate:

```

delete
from
    tab_veiculo
where
    codigo=?

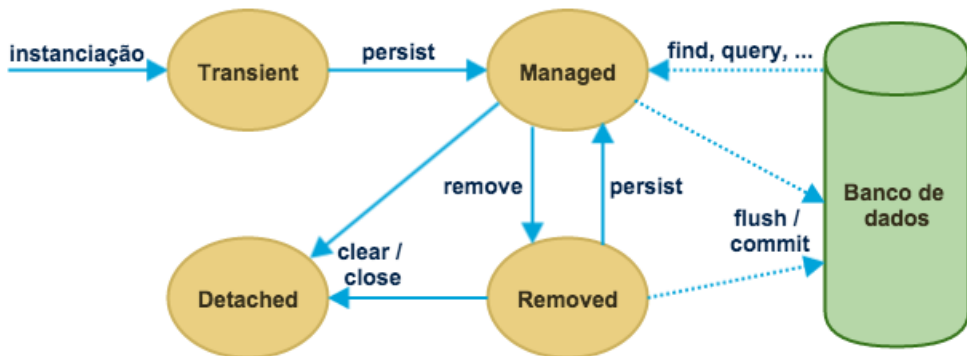
```

Gerenciando estados

3.1. Estados e ciclo de vida

Objetos de entidades são instâncias de classes mapeadas usando JPA, que ficam na memória e representam registros do banco de dados. Essas instâncias possuem um ciclo de vida, que é gerenciado pelo JPA.

Os estados do ciclo de vida das entidades são: *transient* (ou *new*), *managed*, *detached* e *removed*.



As transições entre os estados são feitas através de métodos do EntityManager. Vamos aprender o que significa cada estado do ciclo de vida dos objetos.

Objetos transientes

Objetos transientes (*transient*) são instanciados usando o operador `new`.

Isso significa que eles ainda não estão associados com um registro na tabela do banco de dados e qualquer alteração em seus dados não afeta o estado no banco de dados.

Objetos gerenciados

Objetos gerenciados (*managed*) são instâncias de entidades que possuem um identificador e representam um registro da tabela do banco de dados.

As instâncias gerenciadas podem ser objetos que foram persistidos através da chamada de um método do `EntityManager`, como por exemplo o `persist`.

Eles também podem ter se tornado gerenciados através de métodos de consulta do `EntityManager`, que buscam registros da base de dados e instanciam objetos diretamente no estado *managed*.

Objetos gerenciados estão sempre associados a um contexto de persistência, portanto, quaisquer alterações nesses objetos são sincronizadas com o banco de dados.

Objetos removidos

Uma instância de uma entidade pode ser excluída através do método `remove` do `EntityManager`.

Um objeto entra no estado *removed* quando ele é marcado para ser eliminado, mas é fisicamente excluído durante a sincronização com o banco de dados.

Objetos desanexados

Um objeto sempre inicia no estado transiente e depois pode se tornar gerenciado. Quando o `EntityManager` é fechado, continua existindo uma instância do objeto, mas já no estado *detached*.

Esse estado existe para quando os objetos estão desconectados, não tendo mais sincronia com o banco de dados.

A JPA fornece operações para reconectar esses objetos a um novo EntityManager, que veremos adiante.

3.2. Contexto de persistência

O contexto de persistência é uma coleção de objetos gerenciados por um EntityManager.

Se uma entidade é pesquisada, mas ela já existe no contexto de persistência, o objeto existente é retornado sem acessar o banco de dados. Esse recurso é chamado de **cache de primeiro nível**.

Uma mesma entidade pode ser representada por diferentes objetos na memória, desde que seja em diferentes instâncias de EntityManagers.

Em uma única instância de EntityManager, apenas um objeto que representa determinada entidade (com o mesmo identificador) pode ser gerenciada.

```
EntityManager manager = JpaUtil.getEntityManager();

Veiculo veiculo1 = manager.find(Veiculo.class, 2L);
System.out.println("Buscou veiculo pela primeira vez...");

Veiculo veiculo2 = manager.find(Veiculo.class, 2L);
System.out.println("Buscou veiculo pela segunda vez...");

System.out.println("Mesmo veículo? " + (veiculo1 == veiculo2));

manager.close();
JpaUtil.close();
```

O código acima busca o mesmo veículo duas vezes, dentro do mesmo contexto de persistência. A consulta SQL foi executada apenas na primeira vez, pois na segunda, o objeto já estava no cache de primeiro nível. Veja a saída:

```
Hibernate:
    select
```

```

        veiculo0_.codigo as codigol_0_0_,
        veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
        veiculo0_.ano_modelo as ano_mode3_0_0_,
        veiculo0_.fabricante as fabrican4_0_0_,
        veiculo0_.modelo as modelo5_0_0_,
        veiculo0_.valor as valor6_0_0_
    from
        tab_veiculo veiculo0_
    where
        veiculo0_.codigo=?
Buscou veiculo pela primeira vez...
Buscou veiculo pela segunda vez...
Mesmo veículo? true

```

O método `contains` de `EntityManager` verifica se o objeto está sendo gerenciado pelo contexto de persistência do `EntityManager`.

O método `detach` para de gerenciar a entidade no contexto de persistência, colocando ela no estado *detached*.

```

EntityManager manager = JpaUtil.getEntityManager();

Veiculo veiculo1 = manager.find(Veiculo.class, 2L);
System.out.println("Buscou veiculo pela primeira vez...");

System.out.println("Gerenciado? " + manager.contains(veiculo1));
manager.detach(veiculo1);
System.out.println("E agora? " + manager.contains(veiculo1));

Veiculo veiculo2 = manager.find(Veiculo.class, 2L);
System.out.println("Buscou veiculo pela segunda vez...");

System.out.println("Mesmo veículo? " + (veiculo1 == veiculo2));

manager.close();
JpaUtil.close();

```

A execução do código acima imprime na saída:

```

Hibernate:
    select
        veiculo0_.codigo as codigol_0_0_,
        veiculo0_.ano_fabricacao as ano_fabr2_0_0_,

```



```

        veiculo0_.ano_modelo as ano_mode3_0_0_,
        veiculo0_.fabricante as fabrican4_0_0_,
        veiculo0_.modelo as modelo5_0_0_,
        veiculo0_.valor as valor6_0_0_
    from
        tab_veiculo veiculo0_
    where
        veiculo0_.codigo=?
Buscou veiculo pela primeira vez...
Gerenciado? true
E agora? false
Hibernate:
    select
        veiculo0_.codigo as codigol_0_0_,
        veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
        veiculo0_.ano_modelo as ano_mode3_0_0_,
        veiculo0_.fabricante as fabrican4_0_0_,
        veiculo0_.modelo as modelo5_0_0_,
        veiculo0_.valor as valor6_0_0_
    from
        tab_veiculo veiculo0_
    where
        veiculo0_.codigo=?
Buscou veiculo pela segunda vez...
Mesmo veículo? false

```

Veja que agora a consulta foi executada duas vezes, pois desanexamos o veículo que estava sendo gerenciado pelo contexto de persistência.

3.3. Sincronização de dados

Os estados de entidades são sincronizados com o banco de dados quando ocorre o *commit* da transação associada.

Vamos usar um exemplo anterior, para entender melhor.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

```

```

Veiculo veiculo = manager.find(Veiculo.class, 1L);

System.out.println("Valor atual: " + veiculo.getValor());
veiculo.setValor(veiculo.getValor().add(new BigDecimal(500)));

System.out.println("Novo valor: " + veiculo.getValor());

tx.commit();
manager.close();
JpaUtil.close();

```

Veja na saída abaixo que o comando SQL *update* foi executado apenas no final da execução, exatamente durante o *commit* da transação.

```

Hibernate:
    select
        veiculo0_.codigo as codigo1_0_0_,
        veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
        veiculo0_.ano_modelo as ano_mode3_0_0_,
        veiculo0_.fabricante as fabrican4_0_0_,
        veiculo0_.modelo as modelo5_0_0_,
        veiculo0_.valor as valor6_0_0_
    from
        tab_veiculo veiculo0_
    where
        veiculo0_.codigo=?
Valor atual: 107000.00
Novo valor: 107500.00
Hibernate:
    update
        tab_veiculo
    set
        ano_fabricacao=?,
        ano_modelo=?,
        fabricante=?,
        modelo=?,
        valor=?
    where
        codigo=?

```

Podemos forçar a sincronização antes mesmo do *commit*, chamando o método *flush* de *EntityManager*.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Veiculo veiculo = manager.find(Veiculo.class, 1L);

System.out.println("Valor atual: " + veiculo.getValor());
veiculo.setValor(veiculo.getValor().add(new BigDecimal(500)));

manager.flush();

System.out.println("Novo valor: " + veiculo.getValor());

tx.commit();
manager.close();
JpaUtil.close();

```

Agora o comando SQL *update* foi executado antes do último `println` que fizemos, que exibe o novo valor.

```

Hibernate:
    select
        veiculo0_.codigo as codigo1_0_0_,
        veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
        veiculo0_.ano_modelo as ano_mode3_0_0_,
        veiculo0_.fabricante as fabrican4_0_0_,
        veiculo0_.modelo as modelo5_0_0_,
        veiculo0_.valor as valor6_0_0_
    from
        tab_veiculo veiculo0_
    where
        veiculo0_.codigo=?
Valor atual: 107000.00
Hibernate:
    update
        tab_veiculo
    set
        ano_fabricacao=?,
        ano_modelo=?,
        fabricante=?,
        modelo=?,
        valor=?
    where

```

```
codigo=?  
Novo valor: 107500.00
```

3.4. Salvando objetos desanexados com merge()

Objetos desanexados são objetos em um estado que não é gerenciado pelo EntityManager, mas ainda representam entidades no banco de dados.

As alterações em objetos desanexados não são sincronizadas com o banco de dados.

Quando estamos desenvolvendo com JPA, existem diversos momentos que somos obrigados a trabalhar com objetos desanexados, por exemplo, quando eles são expostos para alteração através de páginas web e apenas em um segundo momento o usuário solicita a gravação das alterações do objeto.

No código abaixo, alteramos o valor de um veículo em um momento que o objeto está no estado *detached*, por isso, a modificação não é sincronizada.

```
EntityManager manager = JpaUtil.getEntityManager();  
EntityTransaction tx = manager.getTransaction();  
tx.begin();  
  
Veiculo veiculo = manager.find(Veiculo.class, 1L);  
  
tx.commit();  
manager.close();  
  
// essa alteração não será sincronizada  
veiculo.setValor(new BigDecimal(112_000));  
  
JpaUtil.close();
```

Podemos reanexar objetos em qualquer EntityManager usando o método merge.

```
EntityManager manager = JpaUtil.getEntityManager();  
EntityTransaction tx = manager.getTransaction();  
tx.begin();  
  
Veiculo veiculo = manager.find(Veiculo.class, 1L);
```

```

tx.commit();
manager.close();

veiculo.setValor(new BigDecimal(112_000));

manager = JpaUtil.getEntityManager();
tx = manager.getTransaction();
tx.begin();

// reanexamos o objeto ao novo EntityManager
veiculo = manager.merge(veiculo);

tx.commit();
manager.close();
JpaUtil.close();

```

O conteúdo do objeto desanexado é copiado para um objeto gerenciado com a mesma identidade.

Se o EntityManager ainda não estiver gerenciando um objeto com a mesma identidade, será realizada uma consulta para encontrá-lo, ou ainda, será persistida uma nova entidade.

O retorno do método merge é uma instância de um objeto gerenciado. O objeto desanexado não muda de estado, ou seja, continua *detached*.

A execução do código acima exibe na saída:

```

Hibernate:
    select
        veiculo0_.codigo as codigo1_0_0_,
        veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
        veiculo0_.ano_modelo as ano_mode3_0_0_,
        veiculo0_.fabricante as fabrican4_0_0_,
        veiculo0_.modelo as modelo5_0_0_,
        veiculo0_.valor as valor6_0_0_
    from
        tab_veiculo veiculo0_
    where
        veiculo0_.codigo=?
Hibernate:
    select
        veiculo0_.codigo as codigo1_0_0_,

```

```

        veiculo0_.ano_fabricacao as ano_fabr2_0_0_,
        veiculo0_.ano_modelo as ano_mode3_0_0_,
        veiculo0_.fabricante as fabrican4_0_0_,
        veiculo0_.modelo as modelo5_0_0_,
        veiculo0_.valor as valor6_0_0_
    from
        tab_veiculo veiculo0_
    where
        veiculo0_.codigo=?
Hibernate:
    update
        tab_veiculo
    set
        ano_fabricacao=?,
        ano_modelo=?,
        fabricante=?,
        modelo=?,
        valor=?
    where
        codigo=?

```

Mapeamento

4.1. Identificadores

A propriedade de identificação mapeia a chave primária de uma tabela do banco de dados. Nos exemplos anteriores, a propriedade `codigo` da classe `Veiculo` representava o código do veículo (chave primária) no banco de dados.

Um exemplo típico de mapeamento de identificadores é o seguinte:

```
@Id
@GeneratedValue
private Long codigo;
```

Assim, mapeamos o atributo `codigo` como chave primária. Podemos alterar o nome da coluna usando `@Column`.

```
@Id
@GeneratedValue
@Column(name = "cod_veiculo")
private Long codigo;
```

A anotação `@Id` marca o atributo como um identificador.

Já a anotação `@GeneratedValue` sem uma estratégia especificada, permite a implementação, que no caso é o Hibernate, escolher a forma como a chave será gerada.

Este último mapeamento poderia ser modificado para a seguinte forma (e o significado seria o mesmo):

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "cod_veiculo")
private Long codigo;
```

A única mudança realizada foi a inclusão da propriedade `strategy` na anotação `@GeneratedValue`. Quando essa propriedade não é informada, é considerada a estratégia *AUTO* como padrão.

Na classe `Veiculo`, estamos usando a estratégia *IDENTITY*.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long codigo;
```

Essa estratégia pode ser escolhida quando o banco de dados tem a capacidade de autoincrementar o valor da chave primaria. No caso do MySQL, estamos falando em ter a coluna da chave primaria com `auto_increment`.

Para SGBDs com suporte a autoincremento, a estratégia *IDENTITY* é a mais usada. Consulte a documentação caso precise de outro tipo de estratégia.

4.2. Chaves compostas

Para exemplificar o uso de chaves compostas, incluiremos os atributos `cidade` e `placa` como identificador de `Veiculo`.

O atributo `codigo` não será mais o identificador, por isso precisaremos eliminá-lo.

Vamos criar uma classe chamada `VeiculoId` para representar o identificador (a chave composta) da entidade.

```
import java.io.Serializable;
import javax.persistence.Embeddable;

@Embeddable
public class VeiculoId implements Serializable {
```



```

    private String placa;
    private String cidade;

    public VeiculoId() {
    }

    public VeiculoId(String placa, String cidade) {
        super();
        this.placa = placa;
        this.cidade = cidade;
    }

    // getters e setters

    // hashCode e equals
}

```

Veja que anotamos a classe `VeiculoId` com `@Embeddable`, pois ela será sempre utilizada de forma “embutida” em outra classe.

Na classe `Veiculo`, criamos um atributo `id` do tipo `VeiculoId` e anotamos apenas com `@EmbeddedId`.

```

import javax.persistence.EmbeddedId;

@Entity
@Table(name = "tab_veiculo")
public class Veiculo {

    @EmbeddedId
    private VeiculoId codigo;

    // outros atributos

    // getters e setters

    // hashCode e equals
}

```

Essa alteração vai impactar também na estrutura do nosso banco de dados. Por isso, para fazer esse teste é preciso alterar nosso arquivo `dados-iniciais.sql`.

```

insert into tab_veiculo (placa, cidade, fabricante, modelo, ano_fabricacao,
ano_modelo, valor) values ('AAA', '1111', 'Fiat',
'Toro', 2020, 2020, 107000);
insert into tab_veiculo (placa, cidade, fabricante, modelo, ano_fabricacao,
ano_modelo, valor) values ('BBB', '2222', 'Ford',
'Fiesta', 2019, 2019, 42000);
insert into tab_veiculo (placa, cidade, fabricante, modelo, ano_fabricacao,
ano_modelo, valor) values ('CCC', '3333', 'VW',
'Gol', 2019, 2020, 35000);

```

Para persistir um novo veículo, é preciso informar a cidade e placa, que faz parte do identificador. Instanciaremos um `VeiculoId` e atribuiremos ao id do veículo.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Veiculo veiculo = new Veiculo();
veiculo.setCodigo(new VeiculoId("ABC-1234", "Uberlândia"));
veiculo.setFabricante("Honda");
veiculo.setModelo("Civic");
veiculo.setAnoFabricacao(2020);
veiculo.setAnoModelo(2020);
veiculo.setValor(new BigDecimal(71_300));

manager.persist(veiculo);

tx.commit();
manager.close();
JpaUtil.close();

```

Para buscar um veículo pela chave, precisamos também instanciar um `VeiculoId` e chamar o método `find` de `EntityManager`, passando como parâmetro a instância de `VeiculoId`.

```

EntityManager manager = JpaUtil.getEntityManager();

VeiculoId codigo = new VeiculoId("AAA-1111", "Uberlândia");
Veiculo veiculo = manager.find(Veiculo.class, codigo);

System.out.println("Placa: " + veiculo.getCodigo().getPlaca());
System.out.println("Cidade: " + veiculo.getCodigo().getCidade());
System.out.println("Fabricante: " + veiculo.getFabricante());
System.out.println("Modelo: " + veiculo.getModelo());

```

```
manager.close();
JpaUtil.close();
```

Nos próximos exemplos deste livro, voltaremos a usar chaves simples.

4.3. Enumerações

Enumerações em Java é um tipo que define um número finito de valores (instâncias), como se fossem constantes.

Na classe `Veiculo`, incluiremos uma enumeração `TipoCombustivel`, para representar os possíveis tipos de combustíveis que os veículos podem suportar.

Primeiro, definimos a *enum*:

```
public enum TipoCombustivel {

    ALCOOL,
    GASOLINA,
    DIESEL,
    BICOMBUSTIVEL

}
```

Depois criamos a propriedade `tipoCombustivel` do tipo `TipoCombustivel` na classe `Veiculo` e configuramos seu mapeamento:

```
public class Veiculo {

    // outros atributos

    @Column(name = "tipo_combustivel", nullable = false)
    @Enumerated(EnumType.STRING)
    private TipoCombustivel tipoCombustivel;

    // getters e setters

    // equals e hashCode

}
```

O novo atributo foi mapeado como uma coluna normal, porém incluímos a

anotação `@Enumerated`, para configurar o tipo da enumeração como *string*. Fizemos isso para que a coluna do banco de dados armazene o nome da constante, e não o número que representa a opção na enumeração.

Para inserir um novo veículo no banco de dados, atribuímos também o tipo do combustível, como você pode ver no exemplo abaixo:

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Veiculo veiculo = new Veiculo();
veiculo.setFabricante("Ford");
veiculo.setModelo("Focus");
veiculo.setAnoFabricacao(2019);
veiculo.setAnoModelo(2020);
veiculo.setValor(new BigDecimal(41_500));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);

manager.persist(veiculo);

tx.commit();
manager.close();
JpaUtil.close();
```

Se o parâmetro da anotação `@Enumerated` for alterado para `EnumType.ORDINAL` (padrão), será inserido o número que representa a opção na enumeração. No caso da gasolina, esse valor seria igual a 1.

```
@Column(name = "tipo_combustivel", nullable = false)
@Enumerated(EnumType.ORDINAL)
private TipoCombustivel tipoCombustivel;
```

Como a propriedade será obrigatória, então precisamos alterar os comandos *insert* do nosso arquivo *dados-iniciais.sql*. Para cada um dos três veículos, adicione a coluna *tipo_combustivel*.

```
insert into tab_veiculo (codigo, fabricante, modelo,
    ano_fabricacao, ano_modelo, valor, tipo_combustivel)
values (1, 'Fiat', 'Toro', 2020, 2020, 107000, 'DIESEL');
insert into tab_veiculo (codigo, fabricante, modelo,
    ano_fabricacao, ano_modelo, valor, tipo_combustivel)
values (2, 'Ford', 'Fiesta', 2019, 2019, 42000, 'GASOLINA');
```

```
insert into tab_veiculo (codigo, fabricante, modelo,
    ano_fabricacao, ano_modelo, valor, tipo_combustivel)
values (3, 'VW', 'Gol', 2019, 2020, 35000, 'BICOMBUSTIVEL');
```

4.4. Propriedades temporais

Para utilizar a “nova” API de datas do Java com `LocalDate`, `LocalDateTime` e `LocalTime`, não é preciso nada de especial.

Basta criar a propriedade com o tipo da precisão desejada.

```
@Column(name = "data_cadastro", nullable = false)
private LocalDate dataCadastro;
```

Hoje em dia, essas são as classes mais utilizadas e as que recomendamos que você use para trabalhar com datas.

Agora, podemos ter atributos de data dos tipos `Date` ou `Calendar`, mas nesse caso precisaríamos também da anotação `@Temporal` para informar a precisão.

Veja abaixo um atributo com o tipo `Date` equivalente ao exemplo anterior.

```
@Temporal(TemporalType.DATE)
@Column(name = "data_cadastro", nullable = false)
private Date dataCadastro;
```

A JPA não define a precisão que deve ser usada se `@Temporal` não for especificada, mas quando usamos Hibernate, as propriedades de datas usam a definição `TemporalType.TIMESTAMP` por padrão. Outras opções são `TemporalType.TIME` e `TemporalType.DATE`.

Para ter uma ideia melhor, `LocalDate` é equivalente a `TemporalType.DATE`, `LocalDateTime` equivale a `TemporalType.TIMESTAMP` e `LocalTime` é `TemporalType.TIME`.

Como a propriedade será obrigatória, então precisamos alterar os comandos *insert* do nosso arquivo *dados-iniciais.sql*. Para cada um dos três veículos, adicione a coluna `data_cadastro`.

Veja um exemplo de como vai ficar:

```
insert into tab_veiculo (codigo, fabricante, modelo,
ano_fabricacao, ano_modelo, valor, tipo_combustivel,
data_cadastro) values (1, 'Fiat', 'Toro', 2020, 2020,
107000, 'DIESEL', sysdate());
insert into tab_veiculo (codigo, fabricante, modelo,
ano_fabricacao, ano_modelo, valor, tipo_combustivel,
data_cadastro) values (2, 'Ford', 'Fiesta', 2019, 2019,
42000, 'GASOLINA', sysdate());
insert into tab_veiculo (codigo, fabricante, modelo,
ano_fabricacao, ano_modelo, valor, tipo_combustivel,
data_cadastro) values (3, 'VW', 'Gol', 2019, 2020,
35000, 'BICOMBUSTIVEL', sysdate());
```

4.5. Propriedades transientes

As propriedades de uma entidade são automaticamente mapeadas se não especificarmos nenhuma anotação.

Por diversas vezes, podemos precisar criar atributos que não representam uma coluna no banco de dados. Nestes casos, devemos anotar com `@Transient`.

```
@Transient
private String descricaoCompleta;
```

A propriedade será ignorada totalmente pelo mecanismo de persistência.

4.6. Objetos grandes

Quando precisamos armazenar muitos dados em uma coluna, por exemplo um texto longo, um arquivo qualquer ou uma imagem, mapeamos a propriedade com a anotação `@Lob`.

Objeto grande em caracteres (CLOB)

Um CLOB (*Character Large Object*) é um tipo de dado em bancos de dados que pode armazenar objetos grandes em caracteres (textos muito longos).

Para mapear uma coluna CLOB em JPA, definimos uma propriedade com o tipo String, char[] ou Character[] e anotamos com @Lob.

```
public class Veiculo {  
  
    // outros atributos  
  
    @Lob  
    private String especificacoes;  
  
    // getters e setters  
  
    // equals e hashCode  
}
```

A coluna criada na tabela é do tipo LONGTEXT, que é um tipo de CLOB do MySQL.

<code>especificacoes</code>	<code>LONGTEXT</code>
-----------------------------	-----------------------

No teste que vamos fazer, por motivos óbvios, não vamos usar a máxima capacidade dessa coluna do banco de dados.

Poderíamos incluir um texto **muito maior** para a especificação do veículo.

Quanto a busca, ela é feita normalmente. Veja a inserção e a busca abaixo.

```
EntityManager manager = JpaUtil.getEntityManager();  
  
EntityTransaction tx = manager.getTransaction();  
tx.begin();  
  
StringBuilder especificacoes = new StringBuilder();  
especificacoes.append("Carro em excelente estado.\n");  
especificacoes.append("Completo, menos ar.\n");  
especificacoes.append("Primeiro dono, com manual de instrução ");  
especificacoes.append("e todas as revisões feitas.\n");  
especificacoes.append("IPVA pago, aceita financiamento.");  
  
Veiculo veiculo = new Veiculo();  
veiculo.setFabricante("VW");  
veiculo.setModelo("Gol");  
veiculo.setAnoFabricacao(2018);  
veiculo.setAnoModelo(2019);  
veiculo.setValor(new BigDecimal(17_200));
```

```

veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(LocalDate.now());
veiculo.setEspecificacoes(especificacoes.toString());

manager.persist(veiculo);
tx.commit();
manager.detach(veiculo);

Veiculo veiculo2 = manager.find(Veiculo.class, veiculo.getCodigo());
System.out.println("Veículo: " + veiculo2.getModelo());
System.out.println("-----");
System.out.println(veiculo2.getEspecificacoes());

manager.close();
JpaUtil.close();

```

Veja a saída da execução:

```

Veículo: Gol
-----
Carro em excelente estado.
Completo, menos ar.
Primeiro dono, com manual de instrução e todas as revisões feitas.
IPVA pago, aceita financiamento

```

Objeto grande binário (BLOB)

Um BLOB (*Binary Large Object*) é um tipo de dado em bancos de dados que pode armazenar objetos grandes em binário (arquivos diversos, incluindo executáveis, músicas, imagens, etc).

Para mapear uma coluna BLOB em JPA, definimos uma propriedade com o tipo `byte[]` ou `Byte[]` e anotamos com `@Lob`.

```

public class Veiculo {

    // outros atributos

    @Lob
    private byte[] foto;

    // getters e setters

```



```

        // equals e hashCode
    }
}

```

A coluna criada na tabela é do tipo `LONGBLOB`, que é um tipo de `BLOB` do MySQL.

Vamos persistir um novo veículo, incluindo uma foto. Podemos buscar um veículo normalmente, obter os bytes da imagem e exibi-la usando a classe `javax.swing.JOptionPane`.

```

import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.persistence.EntityManager;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JOptionPane;

public class ExibindoImagem {

    public static void main(String[] args) throws IOException {
        // Lê bytes do arquivo da imagem.
        Path path = FileSystems.getDefault()
            .getPath("arquivos-extras/Hyundai-ix35.jpg");
        byte[] foto = Files.readAllBytes(path);

        EntityManager manager = JpaUtil.getEntityManager();
        EntityTransaction tx = manager.getTransaction();
        tx.begin();

        Veiculo veiculo = new Veiculo();
        veiculo.setFabricante("Hyundai");
        veiculo.setModelo("ix35");
        veiculo.setAnoFabricacao(2019);
        veiculo.setAnoModelo(2020);
        veiculo.setValor(new BigDecimal(100_000));
        veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
        veiculo.setDataCadastro(LocalDate.now());
        veiculo.setFoto(foto);

        manager.persist(veiculo);
        tx.commit();
        manager.detach(veiculo);

        Veiculo veiculo2 = manager.find(Veiculo.class, veiculo.getCodigo());
    }
}

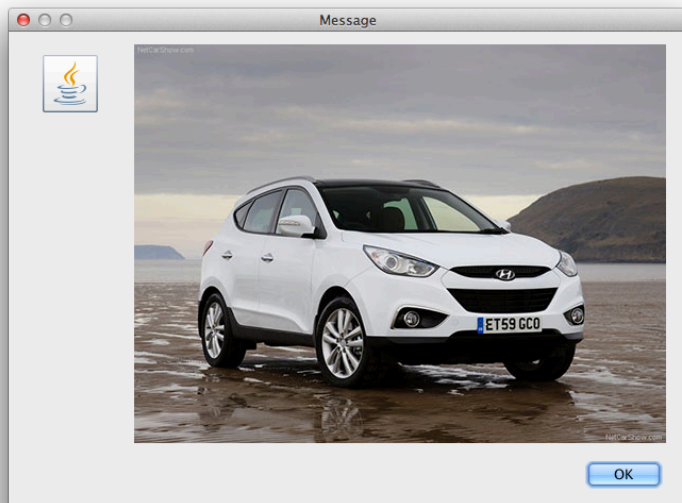
```

```

    if (veiculo2.getFoto() != null) {
        BufferedImage img = ImageIO.read(new ByteArrayInputStream(
            veiculo2.getFoto()));
        JOptionPane.showMessageDialog(null, new JLabel(
            new ImageIcon(img)));
    } else {
        System.out.println("Veículo não possui foto.");
    }

    manager.close();
    JpaUtil.close();
}
}

```



4.7. Objetos embutidos

Objetos embutidos são componentes de uma entidade, cujas propriedades são mapeadas para a mesma tabela da entidade.

Em algumas situações, podemos precisar usar a orientação a objetos para componentizar nossas entidades, mas manter os dados em uma única tabela.

Isso pode ser interessante para evitar muitas associações entre tabelas, e por consequência, diversos *joins* durante as consultas.

Outra situação comum é o mapeamento de tabelas de bancos de dados legados, onde não é permitido alterar a estrutura das tabelas.

Vamos incluir novas colunas na tabela de veículos para armazenar alguns dados do proprietário, mas para não ficar tudo na entidade *Veiculo*, criaremos uma classe *Proprietario*, que representa um proprietário.

```
@Embeddable
public class Proprietario {

    @Column(name = "nome_proprietario", nullable = false)
    private String nome;

    @Column(name = "telefone_proprietario", nullable = false)
    private String telefone;

    @Column(name = "email_proprietario")
    private String email;

    // getters e setters

}
```

Como as colunas serão criadas na tabela de veículos, mapeamos os atributos de *Proprietario* e definimos outros nomes de colunas.

Note que a classe *Proprietario* foi anotada com *@Embeddable*.

Na classe *Veiculo*, incluímos um novo atributo do tipo *Proprietario* anotado com *@Embedded*.

```
public class Veiculo {

    // outros atributos

    @Embedded
    private Proprietario proprietario;

    // getters e setters

}
```

```
// equals e hashCode
```

```
}
```

Já vamos persistir um novo veículo, mas precisamos primeiro ajustar os comandos para inserção de veículos que já existem em nosso arquivo *dados-iniciais.sql*.

Vamos precisar incluir as duas colunas obrigatórias `nome_proprietario` e `telefone_proprietario`.

```
insert into tab_veiculo (codigo, fabricante, modelo,
    ano_fabricacao, ano_modelo, valor, tipo_combustivel, data_cadastro,
    nome_proprietario, telefone_proprietario) values (1, 'Fiat', 'Toro',
    2020, 2020, 107000, 'DIESEL', sysdate(), 'Fernando Martins',
    '34 9 1111 1111');
insert into tab_veiculo (codigo, fabricante, modelo,
    ano_fabricacao, ano_modelo, valor, tipo_combustivel, data_cadastro,
    nome_proprietario, telefone_proprietario) values (2, 'Ford', 'Fiesta',
    2019, 2019, 42000, 'GASOLINA', sysdate(), 'Isabela Santos',
    '34 9 2222 2222');
insert into tab_veiculo (codigo, fabricante, modelo,
    ano_fabricacao, ano_modelo, valor, tipo_combustivel, data_cadastro,
    nome_proprietario, telefone_proprietario) values (3, 'VW', 'Gol',
    2019, 2020, 35000, 'BICOMBUSTIVEL', sysdate(), 'Ulisses Silva',
    '34 9 3333 3333');
```

Ajustando o comando como acima, você evita erros nos comandos de inserção.

Agora, para persistir um novo veículo, precisamos instanciar um `Proprietario` e atribuir ao veículo.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();
```

```
Proprietario proprietario = new Proprietario();
proprietario.setNome("João das Couves");
proprietario.setTelefone("(34) 1234-5678");
```

```
Veiculo veiculo = new Veiculo();
veiculo.setFabricante("VW");
veiculo.setModelo("Gol");
veiculo.setAnoFabricacao(2019);
```

```

veiculo.setAnoModelo(2019);
veiculo.setValor(new BigDecimal(17_200));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(LocalDate.now());
veiculo.setProprietario(proprietario);

manager.persist(veiculo);

tx.commit();
manager.close();
JpaUtil.close();

```

Veja as novas colunas que foram criadas na tabela.

Field	Type	Length	Unsigned	Zerofill	Binary	Allow Null	Key
codigo	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	PRI
ano_fabricacao	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
ano_modelo	INT	11	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
data_cadastro	DATE		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
fabricante	VARCHAR	60	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
modelo	VARCHAR	60	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
email_proprietario	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
nome_proprietario	VARCHAR	60	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
telefone_proprietario	VARCHAR	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
tipo_combustivel	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
valor	DECIMAL	10,2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	

4.8. Associações um-para-um

O relacionamento um-para-um, também conhecido como *one-to-one*, pode ser usado para dividir uma entidade em duas (criando duas tabelas), para ficar mais normalizado e organizado.

Esse tipo de associação poderia ser usado entre Veiculo e Proprietario.



Precisamos apenas anotar a classe Proprietario com @Entity e, opcionalmente, @Table.

Além disso, incluímos também um atributo `codigo`, para armazenar o identificador da entidade (chave primária) e implementamos os métodos `hashCode` e `equals`.

```
@Entity
@Table(name = "proprietario")
public class Proprietario {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column(length = 60, nullable = false)
    private String nome;

    @Column(length = 20, nullable = false)
    private String telefone;

    @Column(length = 255)
    private String email;

    // getters e setters

    // hashCode e equals

}
```

Na classe `Veiculo`, adicionamos a propriedade `proprietario` e mapeamos com `@OneToOne`.

```
public class Veiculo {

    // outros atributos

    @OneToOne
    private Proprietario proprietario;

    // getters e setters

    // equals e hashCode

}
```

Novamente, antes de tentar persistir, vamos ajustar nosso arquivo *dados-iniciais.sql*.

Primeiro, é preciso remover as colunas `nome_proprietario` e `telefone_proprietario` do comando de inserção da tabela `tab_veiculo`.

Depois você deve adicionar, logo antes das inserções de veículos, os comandos para inclusão de proprietários, como abaixo.

```
insert into proprietario (codigo, nome, telefone)
  values (1, 'Fernando Martins', '34 9 1111 1111');
insert into proprietario (codigo, nome, telefone)
  values (2, 'Isabela Santos', '34 9 2222 2222');
insert into proprietario (codigo, nome, telefone)
  values (3, 'Ulisses Silva', '34 9 3333 3333');
```

Por fim, é preciso ajustar os comandos para inserção de veículo adicionando a coluna `proprietario_codigo`. Veja abaixo.

```
insert into tab_veiculo (codigo, fabricante, modelo, ano_fabricacao,
  ano_modelo, valor, tipo_combustivel, data_cadastro, proprietario_codigo)
  values (1, 'Fiat', 'Toro', 2020, 2020, 107000, 'DIESEL', sysdate(), 1);
insert into tab_veiculo (codigo, fabricante, modelo, ano_fabricacao,
  ano_modelo, valor, tipo_combustivel, data_cadastro, proprietario_codigo)
  values (2, 'Ford', 'Fiesta', 2019, 2019, 42000, 'GASOLINA', sysdate(), 2);
insert into tab_veiculo (codigo, fabricante, modelo, ano_fabricacao,
  ano_modelo, valor, tipo_combustivel, data_cadastro, proprietario_codigo)
  values (3, 'VW', 'Gol', 2019, 2020, 35000, 'BICOMBUSTIVEL', sysdate(), 3);
```

Dessa forma, não teremos problemas com os nossos comandos de inserção.

Agora podemos tentar persistir um novo veículo associado a um proprietário.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();
```

```
Proprietario proprietario = new Proprietario();
proprietario.setNome("João das Couves");
proprietario.setTelefone("(34) 1234-5678");
```

```
Veiculo veiculo = new Veiculo();
veiculo.setFabricante("VW");
veiculo.setModelo("Gol");
veiculo.setAnoFabricacao(2018);
veiculo.setAnoModelo(2018);
veiculo.setValor(new BigDecimal(17_200));
```

```

veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(LocalDate.now());
veiculo.setProprietario(proprietario);

manager.persist(veiculo);

tx.commit();
manager.close();
JpaUtil.close();

```

Quando executamos o código acima, recebemos uma exceção, dizendo que o objeto do tipo Veiculo possui uma propriedade que faz referência a um objeto transiente do tipo Proprietario.

```

Caused by: org.hibernate.TransientPropertyValueException: object
references an unsaved transient instance - save the transient
instance before flushing : com.algaworks.lojaveiculos.dominio
.Veiculo.proprietario -> com.algaworks.lojaveiculos.dominio.Proprietario

```

Precisamos de uma instância persistida de proprietário para atribuir ao veículo.

```

Proprietario proprietario = new Proprietario();
proprietario.setNome("João das Couves");
proprietario.setTelefone("(34) 1234-5678");

```

```

// Basta incluir essa linha abaixo para persistir o proprietário.
manager.persist(proprietario);

```

```

Veiculo veiculo = new Veiculo();
veiculo.setFabricante("VW");
veiculo.setModelo("Gol");
veiculo.setAnoFabricacao(2018);
veiculo.setAnoModelo(2018);
veiculo.setValor(new BigDecimal(17_200));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(LocalDate.now());
veiculo.setProprietario(proprietario);

manager.persist(veiculo);

```

Veja a saída da execução:

```

Hibernate:
    insert

```



```

        into
            proprietario
            (email, nome, telefone)
        values
            (?, ?, ?)
Hibernate:
        insert
        into
            tab_veiculo
            (ano_fabricacao, ano_modelo, data_cadastro, fabricante, modelo,
            proprietario_codigo, tipo_combustivel, valor)
        values
            (?, ?, ?, ?, ?, ?, ?, ?)

```

Note que foi criada uma coluna `proprietario_codigo` na tabela `tab_veiculo`.

Por padrão, o nome da coluna é definido com o nome do atributo da associação, mais *underscore*, mais o nome do atributo do identificador da entidade destino. Podemos mudar isso com a anotação `@JoinColumn`.

```

@OneToOne
@JoinColumn(name = "cod_proprietario")
private Proprietario proprietario;

```

O relacionamento *one-to-one* aceita referências nulas, por padrão. Podemos obrigar a atribuição de proprietário durante a persistência de `Veiculo`, incluindo o atributo `optional` com valor `false` na anotação `@OneToOne`.

```

@OneToOne(optional = false)
private Proprietario proprietario;

```

Dessa forma, se tentarmos persistir um veículo sem proprietário, uma exceção é lançada.

```

Caused by: org.hibernate.PropertyValueException: not-null property
references a null or transient value: Veiculo.proprietario

```

Consultando veículos

Quando consultamos veículos, o provedor JPA executa uma consulta do proprietário para cada veículo encontrado:

```

EntityManager manager = JpaUtil.getEntityManager();

List<Veiculo> veiculos = manager
    .createQuery("select v from Veiculo v", Veiculo.class)
    .getResultList();

for (Veiculo veiculo : veiculos) {
    System.out.println(veiculo.getModelo() + " - "
        + veiculo.getProprietario().getNome());
}

manager.close();
JpaUtil.close();

```

Veja as *queries* executadas:

Hibernate:

```

select
    veiculo0_.codigo as codigol_1_,
    veiculo0_.ano_fabricacao as ano_fabr2_1_,
    veiculo0_.ano_modelo as ano_mode3_1_,
    veiculo0_.data_cadastro as data_cad4_1_,
    veiculo0_.fabricante as fabrican5_1_,
    veiculo0_.modelo as modelo6_1_,
    veiculo0_.proprietario_codigo as propriet9_1_,
    veiculo0_.tipo_combustivel as tipo_com7_1_,
    veiculo0_.valor as valor8_1_
from
    tab_veiculo veiculo0_

```

Hibernate:

```

select
    proprietar0_.codigo as codigol_0_0_,
    proprietar0_.email as email2_0_0_,
    proprietar0_.nome as nome3_0_0_,
    proprietar0_.telefone as telefone4_0_0_
from
    proprietario proprietar0_
where
    proprietar0_.codigo=?

```

Podemos mudar esse comportamento padrão e executar apenas uma *query* usando recursos da JPQL, mas ainda não falaremos sobre isso.

Se consultarmos um veículo pelo identificador, a *query* inclui um *left join* ou *inner join* com a tabela de proprietários, dependendo do que foi definido no atributo `optional` do mapeamento `@OneToOne`.

```
EntityManager manager = JpaUtil.getEntityManager();

Veiculo veiculo = manager.find(Veiculo.class, 1L);

System.out.println(veiculo.getModelo() + " - "
    + veiculo.getProprietario().getNome());

manager.close();
JpaUtil.close();
```

Veja a *query* executada com `@OneToOne(optional = false)`:

```
select
    veiculo0_.codigo as codigol_1_1_,
    veiculo0_.ano_fabricacao as ano_fabr2_1_1_,
    veiculo0_.ano_modelo as ano_mode3_1_1_,
    veiculo0_.data_cadastro as data_cad4_1_1_,
    veiculo0_.fabricante as fabrican5_1_1_,
    veiculo0_.modelo as modelo6_1_1_,
    veiculo0_.proprietario_codigo as propriet9_1_1_,
    veiculo0_.tipo_combustivel as tipo_com7_1_1_,
    veiculo0_.valor as valor8_1_1_,
    proprietar1_.codigo as codigol_0_0_,
    proprietar1_.email as email2_0_0_,
    proprietar1_.nome as nome3_0_0_,
    proprietar1_.telefone as telefone4_0_0_
from
    tab_veiculo veiculo0_
inner join
    proprietario proprietar1_
        on veiculo0_.proprietario_codigo=proprietar1_.codigo
where
    veiculo0_.codigo=?
```

Associação bidirecional

A associação que fizemos entre `Veiculo` e `Proprietario` é unidirecional, ou seja, podemos obter o proprietário a partir de um veículo, mas não conseguimos obter

o veículo a partir de um proprietário.

Para tornar a associação um-para-um bidirecional e então conseguirmos obter o veículo a partir de um proprietário, precisamos apenas incluir uma nova propriedade na classe `Proprietario` e mapear com `@OneToOne` usando o atributo `mappedBy`.

```
@Entity
@Table(name = "proprietario")
public class Proprietario {

    // outros atributos

    @OneToOne(mappedBy = "proprietario")
    private Veiculo veiculo;

    // getters e setters

    // equals e hashCode

}
```

O valor de `mappedBy` deve ser igual ao nome da propriedade na classe `Veiculo` que associa com `Proprietario`.

Agora podemos consultar um proprietário e obter o veículo dele.

```
EntityManager manager = JpaUtil.getEntityManager();

Proprietario proprietario = manager.find(Proprietario.class, 1L);

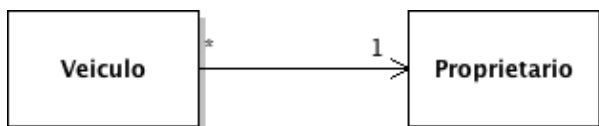
System.out.println(proprietario.getVeiculo().getModelo() + " - "
    + proprietario.getNome());

manager.close();
JpaUtil.close();
```

4.9. Associações muitos-para-um

Na última seção, mapeamos o atributo `proprietario` na entidade `Veiculo` com um-para-um. Mudaremos o relacionamento agora para *many-to-one*. Dessa

forma, um veículo poderá possuir apenas um proprietário, mas um proprietário poderá estar associado a muitos veículos.



Vamos remover o atributo `veiculo` da entidade `Proprietario` e alterar o mapeamento de `proprietario` na classe `Veiculo`.

```
public class Veiculo {

    // outros atributos

    @ManyToOne
    @JoinColumn(name = "proprietario_codigo")
    private Proprietario proprietario;

    // getters e setters

    // equals e hashCode

}
```

A anotação `@ManyToOne` indica a multiplicidade do relacionamento entre veículo e proprietário.

Vamos persistir um proprietário e 2 veículos, que pertencem ao mesmo dono.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();
```

```
Proprietario proprietario = new Proprietario();
proprietario.setNome("João das Couves");
proprietario.setTelefone("(34) 1234-5678");
```

```
manager.persist(proprietario);
```

```
Veiculo veiculo1 = new Veiculo();
veiculo1.setFabricante("GM");
veiculo1.setModelo("Celta");
veiculo1.setAnoFabricacao(2015);
```

```

veiculo1.setAnoModelo(2015);
veiculo1.setValor(new BigDecimal(11_000));
veiculo1.setTipoCombustivel(TipoCombustivel.GASOLINA);
veiculo1.setDataCadastro(LocalDate.now());
veiculo1.setProprietario(proprietario);

manager.persist(veiculo1);

Veiculo veiculo2 = new Veiculo();
veiculo2.setFabricante("VW");
veiculo2.setModelo("Gol");
veiculo2.setAnoFabricacao(2018);
veiculo2.setAnoModelo(2018);
veiculo2.setValor(new BigDecimal(17_200));
veiculo2.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo2.setDataCadastro(LocalDate.now());
veiculo2.setProprietario(proprietario);

manager.persist(veiculo2);

tx.commit();
manager.close();
JpaUtil.close();

```

Para deixar a associação bidirecional, precisamos mapear um atributo na entidade Proprietario, usando a anotação @OneToMany. Veremos isso na próxima seção.

4.10. Coleções um-para-muitos

A anotação @OneToMany deve ser utilizada para mapear coleções.

Mapearemos o inverso da associação *many-to-one*, que fizemos na última seção, indicando que um proprietário pode ter muitos veículos.



Incluiremos o atributo `veiculos` na entidade `Proprietario`, do tipo `List<Veiculo>`.

```

public class Proprietario {

    // outros atributos

    @OneToMany(mappedBy = "proprietario")
    private List<Veiculo> veiculos;

    // getters e setters

    // equals e hashCode

}

```

Para a listagem abaixo ficar mais interessante, inclua mais esse veículo no seu arquivo *dados-iniciais.sql*.

```

insert into tab_veiculo (codigo, fabricante, modelo,
    ano_fabricacao, ano_modelo, valor, tipo_combustivel,
    data_cadastro, proprietario_codigo) values (4, 'Ford', 'Ka',
    2018, 2019, 27000, 'BICOMBUSTIVEL', sysdate(), 1);

```

Assim o proprietário de código 1 vai ficar, segundo nosso arquivo, com dois veículos.

Agora execute o código abaixo que busca um proprietário e lista todos os veículos dele.

```

EntityManager manager = JpaUtil.getEntityManager();

Proprietario proprietario = manager.find(Proprietario.class, 1L);

System.out.println("Proprietário: " + proprietario.getNome());

for (Veiculo veiculo : proprietario.getVeiculos()) {
    System.out.println("Veículo: " + veiculo.getModelo());
}

manager.close();
JpaUtil.close();

```

Veja a saída da execução do código acima:

```

Hibernate:
    select

```

```

        proprietario0_.codigo as codigo1_0_0_,
        proprietario0_.email as email2_0_0_,
        proprietario0_.nome as nome3_0_0_,
        proprietario0_.telefone as telefone4_0_0_
    from
        proprietario proprietario0_
    where
        proprietario0_.codigo=?
Proprietário: Fernando Martins
Hibernate:
    select
        veiculos0_.cod_proprietario as cod_prop9_0_1_,
        veiculos0_.codigo as codigo1_1_1_,
        veiculos0_.codigo as codigo1_1_0_,
        veiculos0_.ano_fabricacao as ano_fabr2_1_0_,
        veiculos0_.ano_modelo as ano_mode3_1_0_,
        veiculos0_.data_cadastro as data_cad4_1_0_,
        veiculos0_.fabricante as fabrican5_1_0_,
        veiculos0_.modelo as modelo6_1_0_,
        veiculos0_.cod_proprietario as cod_prop9_1_0_,
        veiculos0_.tipo_combustivel as tipo_com7_1_0_,
        veiculos0_.valor as valor8_1_0_
    from
        tab_veiculo veiculos0_
    where
        veiculos0_.cod_proprietario=?
Veículo: Toro
Veículo: Ka

```

A primeira *query* foi executada para buscar o proprietário. A segunda, para pesquisar a lista de veículos do proprietário.

Note que a segunda consulta só foi executada quando chamamos o método `getVeiculos`. Esse comportamento é chamado de *lazy loading* e será estudado mais adiante.

4.11. Coleções muitos-para-muitos

Para praticar o relacionamento *many-to-many*, criaremos uma entidade `Acessorio`,

que representa os acessórios que um carro pode ter. Dessa forma, um veículo poderá possuir vários acessórios e um acessório poderá ser associado a vários veículos.



A classe `Acessorio` é uma entidade sem nenhuma novidade no mapeamento.

```
@Entity
@Table(name = "acessorio")
public class Acessorio {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column(length = 60, nullable = false)
    private String descricao;

    // getters e setters

    // hashCode e equals
}
```

Na entidade `Veiculo`, criamos um atributo `acessorios` do tipo `Set<Acessorio>`.

Definimos como um conjunto porque um veículo não poderá possuir o mesmo acessório repetido.

Usamos a anotação `@ManyToMany` para mapear a propriedade de coleção.

```
public class Veiculo {

    // outros atributos

    @ManyToMany
    private Set<Acessorio> acessorios = new HashSet<>();

    // getters e setters
}
```

```
// equals e hashCode
```

```
}
```

Esse tipo de relacionamento precisará de uma tabela de associação para que a multiplicidade muitos-para-muitos funcione. O recurso de *schema generation* do JPA poderá recriar as tabelas automaticamente.



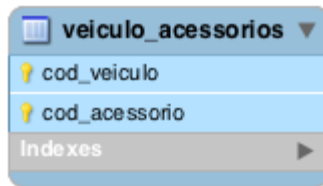
Por padrão, um mapeamento com `@ManyToMany` cria a tabela de associação com os nomes das entidades relacionadas, separados por *underscore*, com as colunas com nomes também gerados automaticamente.

Podemos customizar o nome da tabela de associação e das colunas com a anotação `@JoinTable`.

```
@ManyToMany
```

```
@JoinTable(name = "veiculo_acessorio",
    joinColumns = @JoinColumn(name = "veiculo_codigo"),
    inverseJoinColumns = @JoinColumn(name = "acessorio_codigo"))
private Set<Acessorio> acessorios = new HashSet<>();
```

Neste exemplo, definimos o nome da tabela de associação como `veiculo_acessorio`, o nome da coluna que faz referência para a tabela de veículos como `veiculo_codigo` e da coluna que referencia a tabela de acessórios como `acessorio_codigo` (lado inverso).



Agora vamos inserir e relacionar alguns acessórios e veículos.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

// instancia acessórios
Acessorio roda = new Acessorio();
roda.setDescricao("Rodas de liga leve");

Acessorio sensor = new Acessorio();
sensor.setDescricao("Sensores de estacionamento");

Acessorio mp3 = new Acessorio();
mp3.setDescricao("MP3 Player");

Acessorio pintura = new Acessorio();
pintura.setDescricao("Pintura metalizada");

// persiste acessórios
manager.persist(roda);
manager.persist(sensor);
manager.persist(mp3);
manager.persist(pintura);

// instancia veículos
Veiculo veiculo1 = new Veiculo();
veiculo1.setFabricante("VW");
veiculo1.setModelo("Gol");
veiculo1.setAnoFabricacao(2018);
veiculo1.setAnoModelo(2018);
veiculo1.setValor(new BigDecimal(17_200));
veiculo1.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo1.setDataCadastro(LocalDate.now());
veiculo1.getAcessorios().add(roda);
veiculo1.getAcessorios().add(mp3);

Veiculo veiculo2 = new Veiculo();
```

```

veiculo2.setFabricante("Hyundai");
veiculo2.setModelo("i30");
veiculo2.setAnoFabricacao(2019);
veiculo2.setAnoModelo(2019);
veiculo2.setValor(new BigDecimal(53_500));
veiculo2.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo2.setDataCadastro(LocalDate.now());
veiculo2.getAcessorios().add(roda);
veiculo2.getAcessorios().add(sensor);
veiculo2.getAcessorios().add(mp3);
veiculo2.getAcessorios().add(pintura);

// persiste veículos
manager.persist(veiculo1);
manager.persist(veiculo2);

tx.commit();
manager.close();
JpaUtil.close();

```

O provedor de persistência incluirá os registros nas 3 tabelas.

Consultando acessórios de um veículo

Podemos iterar nos acessórios de um veículo usando o método `getAcessorios` da classe `Veiculo`.

Para que tenhamos registros em nossa base para analisar o resultado da consulta, inclua os comandos abaixo no final do seu arquivo *dados-iniciais.sql*, após a inserção dos veículos.

```

insert into acessorio (codigo, descricao) values (1, 'Direção hidráulica');
insert into acessorio (codigo, descricao) values (2, 'Alarme');
insert into acessorio (codigo, descricao) values (3, 'Ar condicionado');
insert into acessorio (codigo, descricao) values (4, 'Bancos de couro');

insert into veiculo_acessorio (veiculo_codigo, acessorio_codigo)
values (1, 1);
insert into veiculo_acessorio (veiculo_codigo, acessorio_codigo)
values (1, 2);
insert into veiculo_acessorio (veiculo_codigo, acessorio_codigo)
values (1, 3);
insert into veiculo_acessorio (veiculo_codigo, acessorio_codigo)
values (1, 4);

```

Agora sim, podemos testar.

```
EntityManager manager = JpaUtil.getEntityManager();

Veiculo veiculo = manager.find(Veiculo.class, 1L);
System.out.println("Veículo: " + veiculo.getModelo());

for (Acessorio acessorio : veiculo.getAcessorios()) {
    System.out.println("Acessório: " + acessorio.getDescricao());
}

manager.close();
JpaUtil.close();
```

Veja a saída da execução do código acima:

Hibernate:

```
select
    veiculo0_.codigo as codigo1_1_0_,
    veiculo0_.ano_fabricacao as ano_fabr2_1_0_,
    veiculo0_.ano_modelo as ano_mode3_1_0_,
    veiculo0_.data_cadastro as data_cad4_1_0_,
    veiculo0_.fabricante as fabrican5_1_0_,
    veiculo0_.modelo as modelo6_1_0_,
    veiculo0_.tipo_combustivel as tipo_com7_1_0_,
    veiculo0_.valor as valor8_1_0_
from
    veiculo veiculo0_
where
    veiculo0_.codigo=?
```

Veículo: Toro

Hibernate:

```
select
    acessorios0_.cod_veiculo as cod_veic1_1_1_,
    acessorios0_.cod_acessorio as cod_aces2_2_1_,
    acessorio1_.codigo as codigo1_0_0_,
    acessorio1_.descricao as descrica2_0_0_
from
    veiculo_acessorios acessorios0_
inner join
    acessorio acessorio1_
        on acessorios0_.cod_acessorio=acessorio1_.codigo
where
```

```
        acessorios0_.cod_veiculo=?
Acessório: Bancos de couro
Acessório: Direção hidráulica
Acessório: Alarme
Acessório: Ar condicionado
```

Mapeamento bidirecional

Para fazer o mapeamento bidirecional, o lado inverso deve apenas fazer referência ao nome da propriedade que mapeou a coleção na entidade dona da relação, usando o atributo `mappedBy`.

```
public class Acessorio {

    // outros atributos

    @ManyToMany(mappedBy = "acessorios")
    private Set<Veiculo> veiculos = new HashSet<>();

    // getters e setters

    // equals e hashCode

}
```

4.12. Coleções de tipos básicos e objetos embutidos

Em algumas situações, não precisamos criar e relacionar duas entidades, pois uma coleção de tipos básicos ou embutíveis seria suficiente. Para esses casos, usamos `@ElementCollection`.

Para nosso exemplo, voltaremos a usar a entidade `Proprietario`. Um proprietário pode ter vários números de telefones, que são do tipo `String`. Tudo que precisamos é de um `List<String>`.

```
@Entity
@Table(name = "proprietario")
public class Proprietario {

    @Id
```

```

@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long codigo;

@Column(name = "nome", length = 60, nullable = false)
private String nome;

@Column(length = 255)
private String email;

@ElementCollection
@CollectionTable(name = "proprietario_telefone",
    joinColumns = @JoinColumn(name = "proprietario_codigo"))
@Column(name = "telefone_numero", length = 20, nullable = false)
private List<String> telefones = new ArrayList<>();

// getters e setters

// hashCode e equals
}

```

A tabela que armazena os dados da coleção foi customizada através da anotação `@CollectionTable`.

Personalizamos também o nome da coluna que faz referência à tabela de proprietário usando a propriedade `joinColumns`.

A anotação `@Column` foi usada para personalizar o nome da coluna que armazena o número do telefone na tabela da coleção.

Já vamos persistir um proprietário com diversos telefones.

Antes precisamos ajustar as inserções de um proprietário em nosso arquivo *dados-iniciais.sql*.

O ajuste é porque a propriedade `telefone` foi removida da entidade, portanto não faz mais sentido ter o comando de inserção.

```

insert into proprietario (codigo, nome) values (1, 'Fernando Martins');
insert into proprietario (codigo, nome) values (2, 'Isabela Santos');
insert into proprietario (codigo, nome) values (3, 'Ulisses Silva');

```

Agora já podemos persistir sem se preocupar com erros em nosso arquivo.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Proprietario proprietario = new Proprietario();
proprietario.setNome("Sebastião");
proprietario.getTelefones().add("(34) 1234-5678");
proprietario.getTelefones().add("(11) 9876-5432");

manager.persist(proprietario);

tx.commit();
manager.close();
JpaUtil.close();

```

E também podemos listar todos os telefones de um proprietário específico.

Para listagem ficar interessante, logo depois dos comandos da tabela proprietario, inclua as inserções dos telefones.

```

insert into proprietario_telefone (proprietario_codigo, telefone_numero)
  values (1, '34 9 1111 1111');
insert into proprietario_telefone (proprietario_codigo, telefone_numero)
  values (2, '34 9 2222 2222');
insert into proprietario_telefone (proprietario_codigo, telefone_numero)
  values (3, '34 9 3333 3333');

```

Vamos a listagem.

```

EntityManager manager = JpaUtil.getEntityManager();

Proprietario proprietario = manager.find(Proprietario.class, 1L);
System.out.println("Proprietário: " + proprietario.getNome());

for (String telefone : proprietario.getTelefones()) {
    System.out.println("Telefone: " + telefone);
}

manager.close();
JpaUtil.close();

```

Objetos embutidos

Podemos usar @ElementCollection para mapear também tipos de objetos

embutíveis. Por exemplo, suponha que queremos separar o número do telefone em DDD + número e armazenar também o ramal (opcional). Para isso, criamos uma classe Telefone e anotamos com @Embeddable.

@Embeddable

```
public class Telefone {

    @Column(length = 3, nullable = false)
    private String prefixo;

    @Column(length = 20, nullable = false)
    private String numero;

    @Column(length = 5)
    private String ramal;

    public Telefone() {
    }

    public Telefone(String prefixo, String numero, String ramal) {
        this.prefixo = prefixo;
        this.numero = numero;
        this.ramal = ramal;
    }

    // getters e setters

}
```

Na entidade Proprietario, criamos um atributo do tipo List<Telefone> e mapeamos com @ElementCollection.

```
public class Proprietario {

    // outros atributos

    @ElementCollection
    @CollectionTable(name = "proprietario_telefone",
        joinColumns = @JoinColumn(name = "proprietario_codigo"))
    @AttributeOverrides({
        @AttributeOverride(name = "numero",
            column = @Column(name = "telefone_numero", length = 20,
                nullable = false))
    })
    private List<Telefone> telefones = new ArrayList<>();

}
```

```

// getters e setters

// equals e hashCode

}

```

A anotação `@AttributeOverrides` foi usada para substituir o mapeamento da propriedade `numero`, alterando o nome da coluna para `telefone_numero`.

Como sempre, vamos corrigir nosso arquivo *dados-iniciais.sql*. Agora a tabela `proprietario_telefone` tem mais colunas.

```

insert into proprietario_telefone (proprietario_codigo, prefixo,
    telefone_numero, ramal) values (1, '34', '9 1111 1111', '1');
insert into proprietario_telefone (proprietario_codigo, prefixo,
    telefone_numero, ramal) values (2, '34', '9 2222 2222', null);
insert into proprietario_telefone (proprietario_codigo, prefixo,
    telefone_numero, ramal) values (3, '34', '9 3333 3333', null);

```

Agora podemos persistir um proprietário com telefones sem se preocupar com erros em nosso arquivo.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Proprietario proprietario = new Proprietario();
proprietario.setNome("Sebastião");
proprietario.getTelefones().add(new Telefone("34", "1234-5678", "104"));
proprietario.getTelefones().add(new Telefone("11", "9876-5432", null));

manager.persist(proprietario);

tx.commit();
manager.close();
JpaUtil.close();

```

Para listar todos os telefones de um veículo, obtemos a coleção de telefones e chamamos os *getters*.

```

EntityManager manager = JpaUtil.getEntityManager();

Proprietario proprietario = manager.find(Proprietario.class, 1L);

```

```

System.out.println("Proprietário: " + proprietario.getNome());

for (Telefone telefone : proprietario.getTelefones()) {
    System.out.println("Telefone: (" + telefone.getPrefixo() + ") "
        + telefone.getNumero()
        + (telefone.getRamal() != null ? " x" + telefone.getRamal() : ""));
}

manager.close();
JpaUtil.close();

```

4.13. Herança

Mapear herança de classes que representam tabelas no banco de dados pode ser uma tarefa complexa e nem sempre pode ser a melhor solução. Use este recurso com moderação. Muitas vezes é melhor você mapear usando associações do que herança.

A JPA define 3 formas de se fazer o mapeamento de herança:

- Tabela única para todas as classes (*single table*)
- Uma tabela para cada classe da hierarquia (*joined*)
- Uma tabela para cada classe concreta (*table per class*)

Tabela única para todas as classes

Essa estratégia de mapeamento de herança é a melhor em termos de performance e simplicidade, porém seu maior problema é que as colunas das propriedades declaradas nas classes filhas precisam aceitar valores nulos.

A falta da *constraint NOT NULL* pode ser um problema sério no ponto de vista de integridade de dados.

Para implementar essa estratégia, criaremos uma classe abstrata Pessoa.

```

@Entity
@Table(name = "pessoa")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo")

```

```

public abstract class Pessoa {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column(length = 100, nullable = false)
    private String nome;

    // getters e setters

    // hashCode e equals

}

```

Definimos a estratégia SINGLE_TABLE com a anotação @Inheritance. Esse tipo de herança é o padrão, ou seja, não precisaríamos anotar a classe com @Inheritance, embora seja melhor deixar explícito para facilitar o entendimento.

A anotação @DiscriminatorColumn foi usada para informar o nome de coluna de controle para discriminar de qual classe é o registro.

Agora, criaremos as subclasses Cliente e Funcionario.

```

@Entity
@DiscriminatorValue("C")
public class Cliente extends Pessoa {

    @Column(name = "limite_credito", nullable = true)
    private BigDecimal limiteCredito;

    @Column(name = "renda_mensal", nullable = true)
    private BigDecimal rendaMensal;

    @Column(nullable = true)
    private boolean bloqueado;

    // getters e setters

}

@Entity
@DiscriminatorValue("F")
public class Funcionario extends Pessoa {

```

```

@Column(nullable = true)
private BigDecimal salario;

@Column(length = 60, nullable = true)
private String cargo;

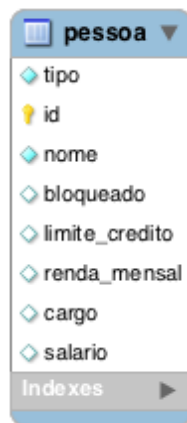
// getters e setters

}

```

As subclasses foram anotadas com `@DiscriminatorValue` para definir o valor discriminador de cada tipo.

Veja a única tabela criada, que armazena os dados de todas as subclasses.



Podemos persistir um cliente e um funcionário normalmente.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

```

```

Funcionario funcionario = new Funcionario();
funcionario.setNome("Fernando");
funcionario.setCargo("Gerente");
funcionario.setSalario(new BigDecimal(12_000));

```

```

Cliente cliente = new Cliente();
cliente.setNome("Mariana");
cliente.setRendaMensal(new BigDecimal(8_500));
cliente.setLimiteCredito(new BigDecimal(2_000));

```

```

cliente.setBloqueado(true);

manager.persist(funcionario);
manager.persist(cliente);

tx.commit();
manager.close();
JpaUtil.close();

```

Veja na saída da execução do código acima que 2 *inserts* foram feitos.

```

Hibernate:
    insert
    into
        pessoa
        (nome, cargo, salario, tipo)
    values
        (?, ?, ?, 'F')
Hibernate:
    insert
    into
        pessoa
        (nome, bloqueado, limite_credito, renda_mensal, tipo)
    values
        (?, ?, ?, ?, 'C')

```

Repare que os valores discriminatórios **F** e **C** foram incluídos no comando de inserção.

Agora vamos incluir alguns comandos no final do nosso arquivo *dados-iniciais.sql* para podermos fazer uma consulta.

```

insert into pessoa (codigo, nome, bloqueado, limite_credito, renda_mensal,
    tipo) values (1, 'Mariana Aguilar', false, 10000, 5000, 'C');
insert into pessoa (codigo, nome, bloqueado, limite_credito, renda_mensal,
    tipo) values (2, 'Douglas Montes', false, 8000, 4500, 'C');

```

Podemos consultar apenas clientes ou funcionários, sem nenhuma novidade no código.

```

EntityManager manager = JpaUtil.getEntityManager();

List<Cliente> clientes = manager

```

```

        .createQuery("select c from Cliente c", Cliente.class)
        .getResultList();

for (Cliente cliente : clientes) {
    System.out.println(cliente.getNome()
        + " - " + cliente.getRendaMensal());
}

manager.close();
JpaUtil.close();

```

Essa consulta gera um SQL com uma condição na cláusula *where*:

```

Hibernate:
select
    cliente0_.codigo as codigo2_0_,
    cliente0_.nome as nome3_0_,
    cliente0_.bloqueado as bloquead4_0_,
    cliente0_.limite_credito as limite_c5_0_,
    cliente0_.renda_mensal as renda_me6_0_
from
    pessoa cliente0_
where
    cliente0_.tipo='C'

```

Inclua agora um funcionário para podemos fazer um outro tipo de consulta.

```

insert into pessoa (codigo, nome, cargo, salario, tipo)
values (3, 'Maria das Dores', 'Gerente', 8000, 'F');

```

Vamos fazer uma consulta polimórfica de pessoas.

```

List<Pessoa> pessoas = manager
    .createQuery("select p from Pessoa p", Pessoa.class)
    .getResultList();

for (Pessoa pessoa : pessoas) {
    System.out.print(pessoa.getNome());

    if (pessoa instanceof Cliente) {
        System.out.println(" - é um cliente");
    } else {
        System.out.println(" - é um funcionário");
    }
}

```

Essa consulta busca todas as pessoas (clientes e funcionários), executando a consulta abaixo:

Hibernate:

```
select
    pessoa0_.codigo as codigo2_0_,
    pessoa0_.nome as nome3_0_,
    pessoa0_.bloqueado as bloquead4_0_,
    pessoa0_.limite_credito as limite_c5_0_,
    pessoa0_.renda_mensal as renda_me6_0_,
    pessoa0_.cargo as cargo7_0_,
    pessoa0_.salario as salario8_0_,
    pessoa0_.tipo as tipol_0_
from
    pessoa pessoa0_
```

Uma tabela para cada classe da hierarquia

Outra forma de fazer mapeamento de herança é usar uma tabela para cada classe da hierarquia (subclasses e superclasse).

Alteramos a estratégia de herança para JOINED na entidade Pessoa.

```
@Entity
@Table(name = "pessoa")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Pessoa {

    // atributos

    // getters e setters

    // equals e hashCode

}
```

Nas classes Cliente e Funcionario, podemos adicionar a anotação @PrimaryKeyJoinColumn para informar o nome da coluna que faz referência à tabela pai, ou seja, o identificador de Pessoa. Se o nome dessa coluna for igual ao nome da coluna da tabela pai, essa anotação não precisa ser utilizada.

```
@Entity
```



```

@Table(name = "funcionario")
@PrimaryKeyJoinColumn(name = "pessoa_codigo")
public class Funcionario extends Pessoa {

    // atributos

    // getters e setters

}

@Entity
@Table(name = "cliente")
@PrimaryKeyJoinColumn(name = "pessoa_codigo")
public class Cliente extends Pessoa {

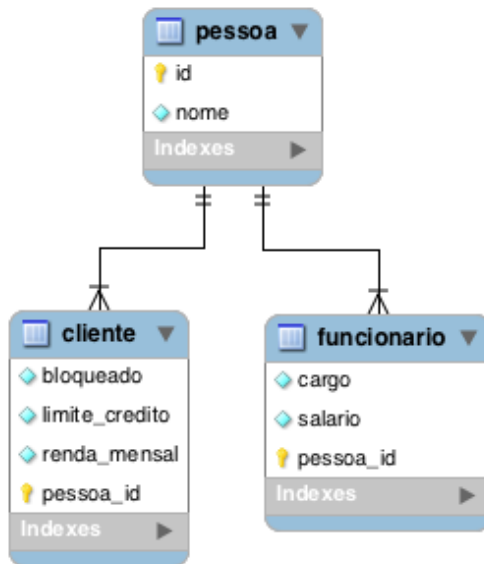
    // atributos

    // getters e setters

}

```

Este tipo de mapeamento criará 3 tabelas.



A parte de persistência das duas entidades pode ficar da mesma forma que você já viu anteriormente. Não muda.

O que vai mudar é a estrutura das tabelas no banco de dados. Inclusive, precisamos ajustar os comandos de inserção de funcionários e clientes. Agora eles ficaram como abaixo.

```
insert into pessoa (codigo, nome) values (1, 'Mariana Aguilar');
insert into pessoa (codigo, nome) values (2, 'Douglas Montes');
insert into pessoa (codigo, nome) values (3, 'Maria das Dores');
```

```
insert into cliente (pessoa_codigo, bloqueado, limite_credito,
renda_mensal) values (1, false, 10000, 5000);
insert into cliente (pessoa_codigo, bloqueado, limite_credito,
renda_mensal) values (2, false, 8000, 4500);
```

```
insert into funcionario (pessoa_codigo, cargo, salario)
values (3, 'Gerente', 8000);
```

Agora podemos executar a pesquisa polimórfica, para listar todas as pessoas. Veja a consulta SQL usada:

```
Hibernate:
select
    pessoa0_.codigo as codigo1_2_,
    pessoa0_.nome as nome2_2_,
    pessoa0_1_.bloqueado as bloquead1_0_,
    pessoa0_1_.limite_credito as limite_c2_0_,
    pessoa0_1_.renda_mensal as renda_me3_0_,
    pessoa0_2_.cargo as cargo1_1_,
    pessoa0_2_.salario as salario2_1_,
    case
        when pessoa0_1_.pessoa_codigo is not null then 1
        when pessoa0_2_.pessoa_codigo is not null then 2
        when pessoa0_.codigo is not null then 0
    end as clazz_
from
    pessoa pessoa0_
left outer join
    cliente pessoa0_1_
        on pessoa0_.codigo=pessoa0_1_.pessoa_codigo
left outer join
    funcionario pessoa0_2_
        on pessoa0_.codigo=pessoa0_2_.pessoa_codigo
```

O Hibernate usou *left outer join* para relacionar a tabela pai às tabelas filhas e

também fez um *case* para controlar de qual tabela/classe pertence o registro.

Uma tabela para cada classe concreta

Uma outra opção de mapeamento de herança é ter tabelas apenas para classes concretas (subclasses). Cada tabela deve possuir todas as colunas, incluindo as da superclasse.

Para utilizar essa forma de mapeamento, devemos anotar a classe Pessoa da maneira apresentada abaixo:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Pessoa {

    @Id
    @GeneratedValue(generator = "inc")
    @GenericGenerator(name = "inc", strategy = "increment")
    private Long codigo;

    // outros atributos

    // getters e setters

    // equals e hashCode

}
```

Tivemos que mudar a estratégia de geração de identificadores “increment”, que a implementação do Hibernate disponibiliza (não é padronizada pelo JPA). Não podemos usar a geração automática de chaves nativa do banco de dados.

Também não precisamos mais da anotação @PrimaryKeyJoinColumn. Pode removê-la das entidades Cliente e Funcionario.

```
@Entity
@Table(name = "cliente")
public class Cliente extends Pessoa {

    // atributos

    // getters e setters

}
```

```

}

@Entity
@Table(name = "funcionario")
public class Funcionario extends Pessoa {

    // atributos

    // getters e setters
}

```

Veja a estrutura das tabelas criadas:



Antes de você executar a consulta, como a estrutura de tabelas mudou novamente, precisamos ajustar nosso arquivo *dados-iniciais.sql*. Agora não tem mais o comando para a tabela pessoa e as outras duas tabelas ficaram como abaixo.

```

insert into cliente (codigo, nome, bloqueado, limite_credito, renda_mensal)
values (1, 'Mariana Aguilar', false, 10000, 5000);
insert into cliente (codigo, nome, bloqueado, limite_credito, renda_mensal)
values (2, 'Douglas Montes', false, 8000, 4500);

insert into funcionario (codigo, nome, cargo, salario)
values (3, 'Maria das Dores', 'Gerente', 8000);

```

Agora a consulta polimórfica por objetos do tipo Pessoa gera o seguinte SQL:

```

Hibernate:
select
    pessoa0_.codigo as codigo1_6_,
    pessoa0_.nome as nome2_6_,

```

```

        pessoa0_.bloqueado as bloquead1_4_,
        pessoa0_.limite_credito as limite_c2_4_,
        pessoa0_.renda_mensal as renda_me3_4_,
        pessoa0_.cargo as cargo1_5_,
        pessoa0_.salario as salario2_5_,
        pessoa0_.clazz_ as clazz_
from
    ( select
        codigo,
        nome,
        bloqueado,
        limite_credito,
        renda_mensal,
        null as cargo,
        null as salario,
        1 as clazz_
    from
        cliente
    union
    all select
        codigo,
        nome,
        null as bloqueado,
        null as limite_credito,
        null as renda_mensal,
        cargo,
        salario,
        2 as clazz_
    from
        funcionario
    ) pessoa0_

```

Herança de propriedades da superclasse

Pode ser útil em algumas situações compartilhar propriedades através de uma superclasse, sem considerá-la como uma entidade mapeada. Para isso, podemos usar a anotação `@MappedSuperclass`.

```

@MappedSuperclass
public abstract class Pessoa {

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long codigo;

// outros atributos

// getters e setters

// equals e hashCode
}

```

As subclasses são mapeadas normalmente, sem nada especial. Continuam somente com `@Entity` e `@Table`.

Apenas as tabelas `cliente` e `funcionario` serão criadas. Como esse tipo de mapeamento não é uma estratégia de herança da JPA, não conseguimos fazer uma consulta polimórfica. Veja a mensagem de erro se tentarmos isso:

```

Caused by: org.hibernate.hql.internal.ast.QuerySyntaxException:
Pessoa is not mapped [select p from Pessoa p]

```

4.14. Modos de acesso

O estado das entidades precisam ser acessíveis em tempo de execução pelo provedor JPA, para poder ler e alterar os valores e sincronizar com o banco de dados.

Existem duas formas de acesso ao estado de uma entidade: *field access* e *property access*.

Field access

Se anotarmos os atributos de uma entidade, o provedor JPA irá usar o modo *field access* para obter e atribuir o estado da entidade, como fizemos nos exemplos das seções anteriores.

Os métodos *getters* e *setters* não são obrigatórios neste caso, mas caso eles existam, serão ignorados pelo provedor JPA.

É recomendado que os atributos tenham o modificador de acesso protegido, privado ou padrão. O modificador público não é recomendado, pois expõe demais os atributos para qualquer outra classe.

No exemplo abaixo, mapeamos a entidade Tarefa usando *field access*.

```
@Entity
@Table(name = "tarefa")
public class Tarefa {

    @Id
    @GeneratedValue
    private Long codigo;

    @Column(length = 100, nullable = false)
    private String descricao;

    @Column(nullable = false)
    private LocalDateTime dataLimite;

    public Long getCodigo() {
        return codigo;
    }

    public void setCodigo(Long codigo) {
        this.codigo = codigo;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public LocalDateTime getDataLimite() {
        return dataLimite;
    }

    public void setDataLimite(LocalDateTime dataLimite) {
        this.dataLimite = dataLimite;
    }

    // hashCode e equals
```

```
}
```

Quando a anotação `@Id` é colocada no atributo, fica automaticamente definido que o modo de acesso é pelos atributos (*field access*).

Property access

Quando queremos usar o acesso pelas propriedades da entidade (*property access*), devemos fazer o mapeamento nos métodos *getters*.

Neste caso, é obrigatório que existam os métodos *getters* e *setters*, pois o acesso aos atributos é feito por eles.

Veja a mesma entidade Tarefa mapeada com *property access*.

```
@Entity
@Table(name = "tarefa")
public class Tarefa {

    private Long codigo;
    private String descricao;
    private LocalDateTime dataLimite;

    @Id
    @GeneratedValue
    public Long getCodigo() {
        return codigo;
    }

    public void setCodigo(Long codigo) {
        this.codigo = codigo;
    }

    @Column(length = 100, nullable = false)
    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
}
```



```
@Column(nullable = false)
public LocalDateTime getDataLimite() {
    return dataLimite;
}

public void setDataLimite(LocalDateTime dataLimite) {
    this.dataLimite = dataLimite;
}

// hashCode e equals
}
```

Recursos avançados

5.1. Lazy loading e eager loading

Podemos definir a estratégia de carregamento de relacionamentos de entidades, podendo ser *lazy* (tardia) ou *eager* (ansiosa).

Para exemplificar, criaremos as entidades *Produto* e *Categoria*, conforme os códigos abaixo.

```
@Entity
@Table(name = "produto")
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column(length = 60, nullable = false)
    private String nome;

    @ManyToOne(optional = false)
    private Categoria categoria;

    // getters e setters

    // hashCode e equals
}
```

```

@Entity
@Table(name = "categoria")
public class Categoria {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column(length = 60, nullable = false)
    private String nome;

    @OneToMany(mappedBy = "categoria")
    private List<Produto> produtos;

    // getters e setters

    // hashCode e equals

}

```

Como você pode perceber, um produto tem uma categoria e uma categoria pode ter muitos produtos.

Vamos inserir, no final do arquivo *dados-iniciais.sql*, uma categoria e alguns produtos para nossos futuros testes.

```

insert into categoria (codigo, nome) values (1, 'Bebidas');

insert into produto (codigo, categoria_codigo, nome)
values (1, 1, 'Água');
insert into produto (codigo, categoria_codigo, nome)
values (2, 1, 'Refrigerante');
insert into produto (codigo, categoria_codigo, nome)
values (3, 1, 'Cerveja');

```

Lazy loading

O código-fonte abaixo obtém um produto com o identificador igual a 3 e exibe o nome dele.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();

Produto produto = manager.find(Produto.class, 3L);

```

```
System.out.println("Nome: " + produto.getNome());
```

```
manager.close();
```

```
JpaUtil.close();
```

A saída da execução do código acima foi:

Hibernate:

```
select
    produto0_.codigo as codigo1_4_0_,
    produto0_.categoria_codigo as categori3_4_0_,
    produto0_.nome as nome2_4_0_,
    categorial_.codigo as codigol_1_1_,
    categorial_.nome as nome2_1_1_
from
    produto produto0_
inner join
    categoria categorial_
        on produto0_.categoria_codigo=categorial_.codigo
where
    produto0_.codigo=?
```

Nome: Cerveja

Note que a *query* SQL fez um *join* na tabela categoria, pois um produto possui uma categoria, mas nós não usamos informações dessa entidade em momento algum. Neste caso, as informações da categoria do produto foram carregadas ansiosamente, mas não foram usadas.

Podemos mudar a estratégia de carregamento para *lazy* no mapeamento da associação de produto com categoria.

```
@ManyToOne(optional = false, fetch = FetchType.LAZY)
```

```
private Categoria categoria;
```

Quando executamos a consulta anterior novamente, veja a saída:

Hibernate:

```
select
    produto0_.codigo as codigo1_4_0_,
    produto0_.categoria_codigo as categori3_4_0_,
    produto0_.nome as nome2_4_0_
```

```

from
    produto produto0_
where
    produto0_.codigo=?
Nome: Cerveja

```

O provedor JPA não buscou as informações de categoria, pois o carregamento passou a ser tardio, ou seja, apenas se for necessário que uma consulta SQL separada será executada. Vamos ver um exemplo:

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();

Produto produto = manager.find(Produto.class, 3L);

System.out.println("Nome: " + produto.getNome());
System.out.println("Categoria: " + produto.getCategoria().getNome());

manager.close();
JpaUtil.close();

```

Agora, duas consultas SQL foram executadas, sendo que a segunda (que busca a categoria do produto) foi executada apenas no momento que o provedor notou a necessidade de informações da categoria.

```

Hibernate:
    select
        produto0_.codigo as codigol_4_0_,
        produto0_.categoria_codigo as categori3_4_0_,
        produto0_.nome as nome2_4_0_
    from
        produto produto0_
    where
        produto0_.codigo=?
Nome: Cerveja

```

```

Hibernate:
    select
        categoria0_.codigo as codigol_1_0_,
        categoria0_.nome as nome2_1_0_
    from
        categoria categoria0_
    where
        categoria0_.codigo=?

```

Categoria: Bebidas

Esse comportamento de carregamento tardio é chamado de *lazy loading* e é útil para evitar consultas desnecessárias, na maioria dos casos.

Quando usamos *lazy loading* precisamos tomar cuidado com o estado da entidade no ciclo de vida. Se tivermos uma instância *detached*, não conseguiremos obter um relacionamento *lazy* ainda não carregado.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();

Produto produto = manager.find(Produto.class, 3L);

System.out.println("Nome: " + produto.getNome());

// quando fechamos o EntityManager,
// todas as instâncias se tornam detached
manager.close();

System.out.println("Categoria: " + produto.getCategoria().getNome());
JpaUtil.close();
```

O código acima gera um erro na saída:

```
Exception in thread "main" org.hibernate.LazyInitializationException:
could not initialize proxy [com.algaworks.lojaveiculos.dominio
.Categoria#1] - no Session
```

Lazy loading em mapeamento OneToOne

Antes de configurar o mapeamento one-to-one para usar lazy loading, você precisa ler este artigo em nosso blog.

<https://blog.algaworks.com/lazy-loading-com-mapeamento-onetoone/>

Eager loading

Todos os relacionamentos *qualquer-coisa-para-muitos*, ou seja, *one-to-many* e *many-to-many*, possuem o *lazy loading* como estratégia padrão. Os demais relacionamentos (que são *qualquer-coisa-para-um*) possuem a estratégia *eager loading* como padrão.

A propriedade `categoria` da entidade `Produto` possuía *eager loading* como estratégia padrão, mas depois alteramos para *lazy loading*.

Eager loading carrega o relacionamento ansiosamente, mesmo se a informação não for usada. Isso pode ser bom se, na maioria das vezes, precisarmos de alguma informação do relacionamento, mas também pode ser ruim, se na maioria das vezes as informações desse relacionamento não forem necessárias e mesmo assim consultadas por causa do *eager loading*.

No exemplo abaixo, consultamos uma categoria pelo identificador e, através do objeto da entidade retornado, iteramos nos produtos.

```
EntityManager manager = JpaUtil.getEntityManager();

Categoria categoria = manager.find(Categoria.class, 1L);

System.out.println("Categoria: " + categoria.getNome());

for (Produto produto : categoria.getProdutos()) {
    System.out.println("Produto: " + produto.getNome());
}

manager.close();
JpaUtil.close();
```

Como a propriedade `produtos` na entidade `Categoria` é um tipo de coleção (mapeado com `@OneToMany`), duas consultas SQL foram executadas, sendo que a segunda, apenas no momento em que o relacionamento de produtos foi necessário.

```
Hibernate:
  select
    categoria0_.codigo as codigo1_1_0_,
    categoria0_.nome as nome2_1_0_
```

```

from
    categoria categoria0_
where
    categoria0_.codigo=?
Categoria: Bebidas
Hibernate:
    select
        produtos0_.categoria_codigo as categori3_4_0_,
        produtos0_.codigo as codigo1_4_0_,
        produtos0_.codigo as codigo1_4_1_,
        produtos0_.categoria_codigo as categori3_4_1_,
        produtos0_.nome as nome2_4_1_
    from
        produto produtos0_
    where
        produtos0_.categoria_codigo=?
Produto: Água
Produto: Refrigerante
Produto: Cerveja

```

Embora não recomendado em muitos casos, podemos mudar a estratégia de carregamento de produtos na entidade *Categoria* para *eager*.

```

@OneToMany(mappedBy = "categoria", fetch = FetchType.EAGER)
private List<Produto> produtos;

```

Agora, quando consultamos um produto e iteramos nas categorias, apenas uma *query* é executada, incluindo um *join* com a tabela de produtos:

```

Hibernate:
    select
        categoria0_.codigo as codigo1_1_0_,
        categoria0_.nome as nome2_1_0_,
        produtos1_.categoria_codigo as categori3_4_1_,
        produtos1_.codigo as codigo1_4_1_,
        produtos1_.codigo as codigo1_4_2_,
        produtos1_.categoria_codigo as categori3_4_2_,
        produtos1_.nome as nome2_4_2_
    from
        categoria categoria0_
    left outer join
        produto produtos1_

```



```

        on categoria0_.codigo=produtos1_.categoria_codigo
    where
        categoria0_.codigo=?
Categoria: Bebidas
Produto: Água
Produto: Refrigerante
Produto: Cerveja

```

Saber escolher a melhor estratégia é muito importante para uma boa performance do software que está sendo desenvolvido.

5.2. Operações em cascata

Nesta seção usaremos como base o mesmo mapeamento dos exemplos sobre *lazy loading* e *eager loading*, ou seja, as entidades Produto e Categoria.

As operações chamadas nos EntityManagers são aplicadas apenas na entidade informada como parâmetro, por padrão. Por exemplo, vamos tentar persistir um produto relacionado a uma categoria transiente no código abaixo:

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Categoria categoria = new Categoria();
categoria.setNome("Roupas");

Produto produto = new Produto();
produto.setNome("Camisa Social");
produto.setCategoria(categoria);

manager.persist(produto);

tx.commit();
manager.close();
JpaUtil.close();

```

Como a associação entre produto e categoria não pode ser nula e a categoria não é persistida automaticamente, a exceção abaixo será lançada:

```

Caused by: org.hibernate.TransientPropertyValueException:

```

```
Not-null property references a transient value - transient instance
must be saved before current operation : com.algaworks.lojaveiculos
.dominio.Produto.categoria ->
com.algaworks.lojaveiculos.dominio.Categoria
```

Persistência em cascata

Em diversas situações, quando persistimos uma entidade, queremos também que seus relacionamentos sejam persistidos. Podemos chamar o método `persist` para cada entidade relacionada, mas essa é uma tarefa um pouco chata.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();
```

```
Categoria categoria = new Categoria();
categoria.setNome("Roupas");
```

```
manager.persist(categoria);
```

```
Produto produto = new Produto();
produto.setNome("Camisa Social");
produto.setCategoria(categoria);
```

```
manager.persist(produto);
```

```
tx.commit();
manager.close();
JpaUtil.close();
```

Felizmente, a JPA fornece um mecanismo para facilitar a persistência de entidades e seus relacionamentos transientes, sempre que o método `persist` for chamado. Esse recurso se chama *cascade*. Para configurá-lo, basta adicionar uma propriedade `cascade` na anotação de relacionamento e definir o valor `CascadeType.PERSIST`.

```
@ManyToOne(optional = false, cascade = CascadeType.PERSIST)
private Categoria categoria;
```

Agora, quando persistirmos um produto, a categoria será persistida também automaticamente.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Categoria categoria = new Categoria();
categoria.setNome("Roupas");

Produto produto = new Produto();
produto.setNome("Camisa Social");
produto.setCategoria(categoria);

manager.persist(produto);

tx.commit();
manager.close();
JpaUtil.close();

```

As operações do EntityManager são identificadas pela enumeração CascadeType com as constantes PERSIST, REFRESH, REMOVE, MERGE e DETACH.

A constante ALL é um atalho para declarar que todas as operações devem ser em cascata.

Legal, agora queremos adicionar produtos em uma categoria e persistir a categoria, através do método persist de EntityManager.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Categoria categoria = new Categoria();
categoria.setNome("Carnes");

Produto produto = new Produto();
produto.setNome("Picanha");
produto.setCategoria(categoria);

categoria.setProdutos(Arrays.asList(produto));

manager.persist(categoria);

tx.commit();
manager.close();
JpaUtil.close();

```

Sabe qual é o resultado disso? Apenas a categoria é inserida no banco de dados. Os produtos foram ignorados.

Para o *cascading* funcionar nessa operação, precisamos configurá-lo no lado inverso do relacionamento, ou seja, no atributo produtos da classe Categoria.

```
@OneToMany(mappedBy = "categoria", cascade = CascadeType.PERSIST)
private List<Produto> produtos;
```

Agora sim, o último exemplo irá inserir a categoria e o produto associado a ela.

Exclusão em cascata

Quando excluimos uma entidade, por padrão, apenas a entidade passada por parâmetro para o método remove é removida.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Categoria categoria = manager.find(Categoria.class, 1L);
manager.remove(categoria);

tx.commit();
manager.close();
JpaUtil.close();
```

O provedor JPA tentará remover apenas a categoria, mas isso não será possível, pois o banco de dados checa violações de integridade.

```
Caused by: java.sql.SQLIntegrityConstraintViolationException:
Cannot delete or update a parent row: a foreign key constraint
fails (`ebookjpa`.`produto`, CONSTRAINT `FKtfuf17yvliycysg3vt5h0sp2v`
FOREIGN KEY (`categoria_codigo`) REFERENCES `categoria` (`codigo`))
```

Vamos configurar a operação de exclusão em cascata no relacionamento produtos da entidade Categoria. Para isso, basta adicionar a constante CascadeType.REMOVE na propriedade cascade do mapeamento.

```
@OneToMany(mappedBy = "categoria",
            cascade = { CascadeType.PERSIST, CascadeType.REMOVE })
private List<Produto> produtos;
```

Agora o último exemplo, que tenta excluir apenas uma categoria, excluirá todos os produtos relacionados antes de remover a categoria.

5.3. Exclusão de objetos órfãos

Usaremos como base o mesmo mapeamento dos exemplos sobre *lazy loading* e *eager loading*, ou seja, as entidades Produto e Categoria.

Para não influenciar nosso exemplo nesta seção, deixe o mapeamento de categoria na entidade Produto da seguinte forma:

```
@ManyToOne(optional = false)
private Categoria categoria;
```

A operação de exclusão em cascata, que você estudou na última seção, remove a entidade passada como parâmetro para o método `remove` de `EntityManager`, mas e se o vínculo entre duas entidades for desfeito sem que haja uma exclusão de entidade pelo `EntityManager`?

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Categoria categoria = manager.find(Categoria.class, 1L);
Produto produto = manager.find(Produto.class, 1L);

categoria.getProdutos().remove(produto);

tx.commit();
manager.close();
JpaUtil.close();
```

No exemplo acima, consideramos que temos uma categoria de identificador 1 e um produto também com identificador 1 e que ambos estão relacionados. Ao remover o produto da coleção de produtos de uma categoria, nada acontece!

Deixamos o produto órfão de um pai (categoria), por isso, o produto poderia ser excluído automaticamente.

Podemos configurar a remoção de órfãos incluindo a propriedade `orphanRemoval`

no mapeamento.

```
@OneToMany(mappedBy = "categoria", cascade = CascadeType.PERSIST,  
    orphanRemoval = true)  
private List<Produto> produtos;
```

A partir de agora, quando deixarmos um produto órfão de categoria, ele será excluído automaticamente do banco de dados.

5.4. Operações em lote

Quando precisamos atualizar ou remover centenas ou milhares de registros do banco de dados, pode se tornar inviável fazer isso objeto por objeto.

Neste caso, podemos usar *bulk operations*, ou operações em lote. JPA suporta esse tipo de operação através da JPQL.

Para este exemplo, criamos uma entidade `Usuario`.

```
@Entity  
@Table(name = "usuario")  
public class Usuario {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long codigo;  
  
    @Column(length = 255, nullable = false)  
    private String email;  
  
    private boolean ativo;  
  
    // getters e setters  
  
    // hashCode e equals  
  
}
```

Incluimos alguns usuários em nosso arquivo *dados-iniciais.sql* para nosso teste.

```
insert into usuario (codigo, email, ativo)  
values (1, 'joao@algaworks.com', true);
```

```

insert into usuario (codigo, email, ativo)
    values (2, 'manoel@algaworks.com', true);
insert into usuario (codigo, email, ativo)
    values (3, 'sebastiao123@gmail.com', true);
insert into usuario (codigo, email, ativo)
    values (4, 'marquim321@gmail.com', false);

```

Queremos inativar todos os usuários que possuem e-mails do domínio “@gmail.com”. Para isso, fazemos uma operação em lote, criando um objeto do tipo Query e chamando o método executeUpdate.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Query query = manager.createQuery(
    "update Usuario set ativo = false where email like :email");
query.setParameter("email", "%@gmail.com");

int linhasAfetadas = query.executeUpdate();

System.out.println(linhasAfetadas + " registro(s) atualizado(s).");

tx.commit();
manager.close();
JpaUtil.close();

```

O texto **:email** no conteúdo da *query* JPQL é um *placeholder* para o parâmetro de e-mail. Definimos o seu valor usando o método `setParameter`.

Todos os usuários com e-mails do domínio “@gmail.com” serão inativados de uma única vez, sem carregar os objetos em memória, o que é muito mais rápido.

Temos que tomar cuidado quando fazemos operações em lote, pois o contexto de persistência não é atualizado de acordo com as operações executadas, portanto, é sempre bom fazer esse tipo de operação em uma transação isolada, onde nenhuma outra alteração nas entidades é feita.

Para fazer uma exclusão em lote, basta usarmos o comando *DELETE* da JPQL.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

```

```

Query query = manager.createQuery("delete from Usuario where ativo = false");

int linhasExcluidas = query.executeUpdate();

System.out.println(linhasExcluidas + " registros removidos.");

tx.commit();
manager.close();
JpaUtil.close();

```

5.5. Concorrência e locking

Em sistemas reais, é comum existir concorrência em entidades por dois ou mais EntityManagers. Simularemos o problema com a entidade Usuario, que usamos na seção anterior.

No código abaixo, abrimos dois EntityManagers, buscamos usuários representados pelo identificador 1 em cada EntityManager e depois alteramos o e-mail em cada objeto.

```

// obtém primeiro EntityManager e inicia transação
EntityManager manager1 = JpaUtil.getEntityManager();
EntityTransaction tx1 = manager1.getTransaction();
tx1.begin();

// obtém segundo EntityManager e inicia transação
EntityManager manager2 = JpaUtil.getEntityManager();
EntityTransaction tx2 = manager2.getTransaction();
tx2.begin();

// altera objeto associado ao primeiro EntityManager
Usuario u1 = manager1.find(Usuario.class, 1L);
u1.setEmail("maria@algaworks.com");

// altera objeto associado ao segundo EntityManager
Usuario u2 = manager2.find(Usuario.class, 1L);
u2.setEmail("jose@algaworks.com");

// faz commit na primeira transação
tx1.commit();
manager1.close();

```



```
// faz commit na segunda transação
tx2.commit();
manager2.close();

JpaUtil.close();
```

Como buscamos os usuários usando EntityManagers diferentes, os objetos não são os mesmos na memória da JVM.

Veja a saída da execução do código acima:

```
Hibernate:
  select
    usuario0_.codigo as codigo1_0_0_,
    usuario0_.ativo as ativo2_0_0_,
    usuario0_.email as email3_0_0_
  from
    usuario usuario0_
  where
    usuario0_.codigo=?
```

```
Hibernate:
  select
    usuario0_.codigo as codigo1_0_0_,
    usuario0_.ativo as ativo2_0_0_,
    usuario0_.email as email3_0_0_
  from
    usuario usuario0_
  where
    usuario0_.codigo=?
```

```
Hibernate:
  update
    usuario
  set
    ativo=?,
    email=?
  where
    codigo=?
```

```
Hibernate:
  update
    usuario
  set
    ativo=,
```

```
email=?  
where  
codigo=?
```

Como você pode imaginar, quando fizemos *commit* no primeiro *EntityManager*, a alteração foi sincronizada com o banco de dados. O outro *commit*, do segundo *EntityManager*, também sincronizou a alteração feita no usuário associado a este contexto de persistência, substituindo a modificação anterior.

Se uma situação como essa ocorrer em valores monetários, por exemplo, você pode ter sérios problemas. Imagine fazer sua empresa perder dinheiro por problemas com concorrência?

Locking otimista

Uma das formas de resolver o problema de concorrência é usando locking otimista.

Este tipo de locking tem como filosofia que, dificilmente, outro *EntityManager* estará fazendo uma alteração no mesmo objeto ao mesmo tempo, ou seja, é uma estratégia otimista que entende que o problema de concorrência é uma exceção.

No momento que uma alteração for sincronizada com o banco de dados, o provedor JPA verifica se o objeto foi alterado por outra transação e lança uma exceção *OptimisticLockException*, caso exista concorrência.

Mas como o provedor JPA sabe se houve uma alteração no objeto?

A resposta é que o provedor mantém um controle de versão em um atributo da entidade. Precisamos mapear uma propriedade para armazenar a versão da entidade, usando a anotação *@Version*.

```
public class Usuario {  
  
    // outros atributos  
  
    @Version  
    private Long versao;  
  
    // getters e setters
```

```
// equals e hashCode  
  
}
```

Note que o atributo versão também é uma coluna no banco de dados, portanto precisamos ajustar os comandos *insert* de usuário em nosso arquivo.

```
insert into usuario (codigo, email, ativo, versao)  
  values (1, 'joao@algaworks.com', true, 0);  
insert into usuario (codigo, email, ativo, versao)  
  values (2, 'manoel@algaworks.com', true, 0);  
insert into usuario (codigo, email, ativo, versao)  
  values (3, 'sebastiao123@gmail.com', true, 0);  
insert into usuario (codigo, email, ativo, versao)  
  values (4, 'marquim321@gmail.com', false, 0);
```

Agora, se houver concorrência na alteração do mesmo registro, uma exceção será lançada:

```
Caused by: org.hibernate.StaleObjectStateException:  
Row was updated or deleted by another transaction (or unsaved-value  
mapping was incorrect) : [com.algaworks.lojaveiculos.dominio.Usuario#1]
```

Aprenda mais sobre Locking Otimista

Para não ficar qualquer dúvida, leia este artigo do blog da AlgaWorks que explica como funciona o locking otimista.

<https://blog.algaworks.com/entendendo-o-lock-otimista-do-jpa/>

Locking pessimista

O locking pessimista “trava” o objeto imediatamente, ao invés de aguardar o *commit* com otimismo, esperando que nada dê errado.

Um lock pessimista garante que o objeto travado não será modificado por outra transação até que a transação atual seja finalizada e libere a trava.

Usar essa abordagem limita a escalabilidade de sua aplicação, pois deixa as operações ocorrerem apenas em série, por isso, deve ser usada com cuidado. Na maioria dos casos, as operações poderiam ocorrer em paralelo, usando locking otimista.

Uma das formas de usar locking pessimista é passando um parâmetro para o método `find` de `EntityManager`.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Usuario usuario = manager.find(Usuario.class, 1L,
    LockModeType.PESSIMISTIC_WRITE);

tx.commit();
manager.close();
JpaUtil.close();
```

O *lock* é feito adicionando `for update` na consulta SQL executada, veja:

```
Hibernate:
    select
        usuario0_.codigo as codigo1_8_0_,
        usuario0_.ativo as ativo2_8_0_,
        usuario0_.email as email3_8_0_,
        usuario0_.versao as versao4_8_0_
    from
        usuario usuario0_
    where
        usuario0_.codigo=? for update
```

5.6. Métodos de callback e auditores de entidades

Em algumas situações específicas, precisamos escutar e reagir a alguns eventos que acontecem no mecanismo de persistência, como por exemplo durante o carregamento de uma entidade, antes de persistir, depois de persistir, etc.

A especificação JPA fornece duas formas para fazer isso: métodos de callback e auditores de entidades, também conhecidos como *callback methods* e *entity*

listeners.

Métodos de callback

Podemos criar um método na entidade, que será chamado quando algum evento ocorrer. Tudo que precisamos fazer é anotar o método com uma ou mais anotações de callback.

Neste exemplo, usaremos a entidade `Animal` do sistema de uma fazenda, que controla a idade dos animais.

```
@Entity
@Table(name = "animal")
public class Animal {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column(length = 60, nullable = false)
    private String nome;

    @Column(name = "data_nascimento", nullable = false)
    private LocalDate dataNascimento;

    @Column(name = "data_ultima_atualizacao", nullable = false)
    private LocalDateTime dataUltimaAtualizacao;

    @Transient
    private Integer idade;

    @PostLoad
    @PostPersist
    @PostUpdate
    public void calcularIdade() {
        long anos = ChronoUnit.YEARS
            .between(this.dataNascimento, LocalDate.now());

        this.idade = (int) anos;
    }

    // getters e setters
}
```

```
// hashCode e equals

}
```

Veja que temos uma propriedade *transient* (não persistida) chamada idade. Essa propriedade é calculada automaticamente nos eventos @PostLoad, @PostPersist e @PostUpdate, através do método calcularIdade.

Esses eventos representam, respectivamente: após o carregamento de uma entidade no contexto de persistência, após a persistência de uma entidade e após a atualização de uma entidade.

Vamos testar o evento @PostPersist com o código abaixo:

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
tx.begin();

Animal animal = new Animal();
animal.setNome("Mimosa");
animal.setDataNascimento(LocalDate.now().minusYears(5));
animal.setDataUltimaAtualizacao(LocalDateTime.now());

System.out.println("Idade antes de persistir: " + animal.getIdade());

manager.persist(animal);

System.out.println("Idade depois de persistir: " + animal.getIdade());

tx.commit();
manager.close();
JpaUtil.close();
```

Analisando a saída da execução, notamos que o cálculo da idade do animal foi feito apenas após a persistência da entidade.

```
Idade antes de persistir: null
Hibernate:
    insert
    into
        animal
        (data_nascimento, data_ultima_atualizacao, nome)
    values
```

```
(?, ?, ?)
Idade depois de persistir: 2
```

As anotações de callback possíveis são: @PrePersist, @PreRemove, @PostPersist, @PostRemove, @PreUpdate, @PostUpdate e @PostLoad.

Audidores de entidades

Você pode também definir uma classe auditora de entidade, ao invés de criar métodos de callback diretamente dentro da entidade.

Criaremos um evento para alterar a data de última atualização do animal automaticamente, antes da persistência e atualização, para que não haja necessidade de informar isso no código toda vez.

```
public class AuditorAnimal {

    @PreUpdate
    @PrePersist
    public void alterarDataUltimaAtualizacao(Animal animal) {
        animal.setDataUltimaAtualizacao(LocalDateTime.now());
    }

}
```

Precisamos anotar a entidade com @EntityListeners, informando as classes que escutarão os eventos da entidade.

```
@Entity
@Table(name = "animal")
@EntityListeners(AuditorAnimal.class)
public class Animal {

    // atributos, getters e setters, equals e hashCode

}
```

Agora, não existe mais a necessidade de informar a data de última atualização de um animal. Ao persistir um animal, a data será calculada automaticamente.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction tx = manager.getTransaction();
```

```

tx.begin();

Animal animal = new Animal();
animal.setNome("Campeão");
animal.setDataNascimento(LocalDate.now().minusYears(7));

manager.persist(animal);

tx.commit();
manager.close();
JpaUtil.close();

```

5.7. Cache de segundo nível

Caching é um termo usado quando armazenamos objetos em memória para acesso mais rápido no futuro. O cache de segundo nível (L2) é compartilhado entre todos os EntityManagers da aplicação.

Para ativar o cache de segundo nível, precisamos selecionar uma implementação de cache e adicionar algumas configurações no arquivo *persistence.xml*.

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
    version="2.2">
    <persistence-unit name="AlgaWorks-PU">

        <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>

    <properties>
        <property name="javax.persistence.jdbc.url"
            value="jdbc:mysql://localhost/ebookjpa?
                createDatabaseIfNotExist=
                true&useTimezone=true&serverTimezone=UTC" />
        <property name="javax.persistence.jdbc.user" value="root" />
        <property name="javax.persistence.jdbc.password" value="1234" />
        <property name="javax.persistence.jdbc.driver"
            value="com.mysql.cj.jdbc.Driver" />

        <property
            name="javax.persistence.schema-generation.database.action"

```



```

        value="drop-and-create"/>

        <property name="javax.persistence.sql-load-script-source"
            value="META-INF/dados-iniciais.sql"/>

        <property name="hibernate.dialect"
            value="org.hibernate.dialect.MySQL8Dialect" />

        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />

        <property name="hibernate.cache.region.factory_class"
            value="org.hibernate.cache.jcache.internal.JCacheRegionFactory"/>
        <property name="hibernate.javax.cache.provider"
            value="org.ehcache.jsr107.EhcacheCachingProvider"/>
        <property name="hibernate.javax.cache.uri"
            value="META-INF/ehcache.xml"/>
    </properties>
</persistence-unit>
</persistence>

```

Usaremos o EhCache, por isso precisamos adicionar essas duas dependências no arquivo *pom.xml*:

```

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jcache</artifactId>
    <version>5.4.10.Final</version>
</dependency>

<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.8.1</version>
</dependency>

```

Definimos a implementação do cache no elemento `hibernate.cache.region.factory_class` do arquivo *persistence.xml*. Em nosso exemplo, especificamos uma classe que não será a provedora do cache em si. Ela irá procurar por algum provedor que implemente a especificação de cache do Java chamada JCache.

Assim como temos a especificação JPA que é implementada pelo Hibernate, temos também a especificação JCache que é implementada pelo EhCache.

Através do elemento `hibernate.javax.cache.provider` nós deixamos explícita a classe do EhCache que contém a implementação.

Por fim, utilizando o elemento `hibernate.javax.cache.uri`, podemos informar o caminho para o arquivo de configuração do EhCache. O conteúdo dele pode ficar como o abaixo.

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns='http://www.ehcache.org/v3'
  xmlns:jsr107='http://www.ehcache.org/v3/jsr107'
  xsi:schemaLocation="
    http://www.ehcache.org/v3
    http://www.ehcache.org/schema/ehcache-core-3.0.xsd
    http://www.ehcache.org/v3/jsr107
    http://www.ehcache.org/schema/ehcache-107-ext-3.0.xsd">

  <service>
    <jsr107:defaults default-template="padrao" />
  </service>

  <cache-template name="padrao">
    <key-type>java.lang.Object</key-type>
    <value-type>java.lang.Object</value-type>

    <expiry>
      <ttl unit="seconds">20</ttl>
    </expiry>

    <heap unit="entries">1000</heap>
  </cache-template>
</config>
```

Com a configuração acima, estamos dizendo que será permitido armazenar até 1000 objetos na memória e cada objeto tem um tempo de vida no cache de 20 segundos.

O elemento `shared-cache-mode` no *persistence.xml* aceita alguns valores que configuram o cache de segundo nível:

- `ENABLE_SELECTIVE` (padrão): as entidades não são armazenadas no cache, a menos que sejam explicitamente anotadas com `@Cacheable(true)`.

- **DISABLE_SELECTIVE**: as entidades são armazenadas no cache, a menos que sejam explicitamente anotadas com `@Cacheable(false)`.
- **ALL**: todas as entidades são sempre armazenadas no cache.
- **NONE**: nenhuma entidade é armazenada no cache (desabilita o cache de segundo nível).

Para exemplificar o uso de cache de segundo nível, criaremos uma entidade `CentroCusto`. Considerando que o centro de custo de um sistema de gestão empresarial é algo que não se altera com frequência, podemos definir que as entidades podem ser adicionadas ao cache de segundo nível usando a anotação `@Cacheable(true)`.

```
@Entity
@Table(name = "centro_custo")
@Cacheable(true)
public class CentroCusto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column(length = 60, nullable = false)
    private String nome;

    // getters e setters

    // hashCode e equals

}
```

Para testar, vamos inserir alguns centros de custo em nosso arquivo *dados-iniciais.sql*.

```
insert into centro_custo (codigo, nome) values (1, 'Tecnologia');
insert into centro_custo (codigo, nome) values (2, 'Comercial');
```

Agora, fazemos duas consultas da mesma entidade em `EntityManagers` diferentes.

```
EntityManager manager1 = JpaUtil.getEntityManager();
CentroCusto centro1 = manager1.find(CentroCusto.class, 1L);
System.out.println("Centro de custo: " + centro1.getNome());
```

```

manager1.close();

System.out.println("-----");

EntityManager manager2 = JpaUtil.getEntityManager();
CentroCusto centro2 = manager2.find(CentroCusto.class, 1L);
System.out.println("Centro de custo: " + centro2.getNome());
manager2.close();

JpaUtil.close();

```

Apenas uma consulta SQL foi gerada, graças ao cache de segundo nível.

```

Hibernate:
    select
        centrocust0_.codigo as codigo1_3_0_,
        centrocust0_.nome as nome2_3_0_
    from
        centro_custo centrocust0_
    where
        centrocust0_.codigo=?
Centro de custo: Tecnologia
-----
Centro de custo: Tecnologia

```

Aprenda mais sobre cache de segundo nível

No blog da AlgaWorks tem um artigo que explica um pouco mais sobre cache de segundo nível.

<https://blog.algaworks.com/introducao-ao-cache-de-segundo-nivel-do-jpa/>

Java Persistence Query Language

6.1. Introdução à JPQL

Se você quer consultar um objeto e já sabe o identificador dele, pode usar os métodos `find` ou `getReference` de `EntityManager`, como já vimos anteriormente.

Agora, caso o identificador seja desconhecido ou você quer consultar uma coleção de objetos, você precisará de uma *query*.

A JPQL (*Java Persistence Query Language*) é a linguagem de consulta padrão da JPA, que permite escrever consultas portáteis, que funcionam independente do SGBD.

Esta linguagem de *query* usa uma sintaxe parecida com a SQL para selecionar objetos e valores de entidades e os relacionamentos entre elas.

Os exemplos desse capítulo usarão as entidades que foram geradas até aqui, mais especificamente, as que serão colocadas a seguir:

```
@Entity
@Table(name = "tab_veiculo")
public class Veiculo {

    @Id
```

```

@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long codigo;

@Column(length = 60, nullable = false)
private String fabricante;

@Column(length = 60, nullable = false)
private String modelo;

@Column(name = "ano_fabricacao", nullable = false)
private Integer anoFabricacao;

@Column(name = "ano_modelo", nullable = false)
private Integer anoModelo;

@Column(precision = 10, scale = 2, nullable = true)
private BigDecimal valor;

@Column(name = "tipo_combustivel", nullable = false)
@Enumerated(EnumType.STRING)
private TipoCombustivel tipoCombustivel;

@Column(name = "data_cadastro", nullable = false)
private LocalDate dataCadastro;

@Transient
private String descricaoCompleta;

@Lob
private String especificacoes;

@Lob
private byte[] foto;

@ManyToOne
@JoinColumn(name = "proprietario_codigo")
private Proprietario proprietario;

@ManyToMany
@JoinTable(name = "veiculo_acessorio",
    joinColumns = @JoinColumn(name = "veiculo_codigo"),
    inverseJoinColumns = @JoinColumn(name = "acessorio_codigo"))
private Set<Acessorio> acessorios = new HashSet<>();

// getters e setters

```

```

        // hashCode e equals
    }

    public enum TipoCombustivel {

        ALCOOL, GASOLINA, DIESEL, BICOMBUSTIVEL

    }

    @Entity
    @Table(name = "proprietario")
    public class Proprietario {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long codigo;

        @Column(length = 60, nullable = false)
        private String nome;

        @Column(length = 255)
        private String email;

        @OneToMany(mappedBy = "proprietario")
        private List<Veiculo> veiculos;

        @ElementCollection
        @CollectionTable(name = "proprietario_telefone",
            joinColumns = @JoinColumn(name = "proprietario_codigo"))
        @AttributeOverrides({
            @AttributeOverride(name = "numero",
                column = @Column(name = "telefone_numero",
                    length = 20, nullable = false))
        })
        private List<Telefone> telefones = new ArrayList<>();

        // getters e setters

        // hashCode e equals
    }

    @Entity
    @Table(name = "acessorio")
    public class Acessorio {

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long codigo;

@Column(length = 60, nullable = false)
private String descricao;

@ManyToMany(mappedBy = "acessorios")
private Set<Veiculo> veiculos = new HashSet<>();

// getters e setters

// hashCode e equals
}

```

Usaremos também esses dois DTOs:

```

public class PrecoVeiculo {

    private String modelo;
    private BigDecimal valor;

    public PrecoVeiculo(String modelo, BigDecimal valor) {
        super();
        this.modelo = modelo;
        this.valor = valor;
    }

    // getters

}

public class TotalCarroPorAno {

    private Integer anoFabricacao;
    private Double mediaPreco;
    private Long quantidadeCarros;

    public TotalCarroPorAno(Integer anoFabricacao, Double mediaPreco,
        Long quantidadeCarros) {
        super();
        this.anoFabricacao = anoFabricacao;
        this.mediaPreco = mediaPreco;
        this.quantidadeCarros = quantidadeCarros;
    }

}

```



```
// getters  
  
}
```

6.2. Consultas simples e iteração no resultado

O método `EntityManager.createQuery` é usado para consultar entidades e valores usando JPQL. Já usamos esse método para fazer consultas simples em outros capítulos, mas não exploramos os detalhes.

As consultas criadas através do método `createQuery` são chamadas de **consultas dinâmicas**, pois elas são definidas diretamente no código da aplicação.

```
EntityManager manager = JpaUtil.getEntityManager();  
  
Query query = manager.createQuery(  
    "select v from Veiculo v where anoFabricacao = 2019");  
List veiculos = query.getResultList();  
  
for (Object obj : veiculos) {  
    Veiculo veiculo = (Veiculo) obj;  
  
    System.out.println(veiculo.getModelo() + " " + veiculo.getFabricante()  
        + ": " + veiculo.getAnoFabricacao());  
}  
  
manager.close();  
JpaUtil.close();
```

O método `createQuery` retorna um objeto do tipo `Query`, que pode ser consultado através de `getResultList`. Este método retorna um `List` não tipado, por isso fizemos um *cast* na iteração dos objetos.

A consulta JPQL que escrevemos seleciona todos os objetos de veículos que possuem o ano de fabricação igual a 2019.

```
select v from Veiculo v where anoFabricacao = 2019
```

Essa consulta é o mesmo que:

```
from Veiculo where anoFabricacao = 2019
```

Veja que não somos obrigados a incluir a instrução `select` neste caso e nem informar um *alias* para a entidade `Veiculo`, mas essa segunda forma só funciona com o Hibernate. De acordo com a especificação JPA é preciso incluir a instrução e o *alias*.

6.3. Usando parâmetros nomeados

Se você precisar fazer uma consulta filtrando por valores informados pelo usuário, não é recomendado que faça concatenações na string da consulta, principalmente para evitar *SQL Injection*. O ideal é que use parâmetros da Query.

```
EntityManager manager = JpaUtil.getEntityManager();

Query query = manager.createQuery(
    "select v from Veiculo v where anoFabricacao >= :ano " +
    " and valor <= :preco");
query.setParameter("ano", 2020);
query.setParameter("preco", new BigDecimal(107_000));
List veiculos = query.getResultList();

for (Object obj : veiculos) {
    Veiculo veiculo = (Veiculo) obj;

    System.out.println(veiculo.getModelo() + " " + veiculo.getFabricante()
        + ": " + veiculo.getAnoFabricacao());
}

manager.close();
JpaUtil.close();
```

Nomeamos os parâmetros com um prefixo `:` e vinculamos os valores para cada parâmetro usando o método `setParameter` de Query.

Veja a consulta SQL gerada:

```
Hibernate:
select
    veiculo0_.codigo as codigol_2_,
    veiculo0_.ano_fabricacao as ano_fabr2_2_,
```

```

        veiculo0_.ano_modelo as ano_mode3_2_,
        veiculo0_.data_cadastro as data_cad4_2_,
        veiculo0_.fabricante as fabrican5_2_,
        veiculo0_.modelo as modelo6_2_,
        veiculo0_.cod_proprietario as cod_prop9_2_,
        veiculo0_.tipo_combustivel as tipo_com7_2_,
        veiculo0_.valor as valor8_2_
    from
        veiculo veiculo0_
    where
        veiculo0_.ano_fabricacao>=?
        and veiculo0_.valor<=?

```

6.4. Consultas tipadas

Quando fazemos uma consulta, podemos obter o resultado de um tipo específico, sem precisar fazer *casting* dos objetos ou até mesmo conversões não checadas para coleções genéricas.

```

EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Veiculo> query = manager
    .createQuery("select v from Veiculo v", Veiculo.class);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo veiculo : veiculos) {
    System.out.println(veiculo.getModelo() + " " + veiculo.getFabricante()
        + ": " + veiculo.getAnoFabricacao());
}

manager.close();
JpaUtil.close();

```

Usamos a interface `TypedQuery`, que é um subtipo de `Query`. A diferença é que o método `getResultList` já retorna uma lista do tipo que especificamos na criação da *query*, no segundo parâmetro.

6.5. Paginação

Uma consulta que retorna muitos objetos pode ser um problema para a muitas aplicações.

Se existir a necessidade de exibir um conjunto de dados grande, é interessante implementar uma paginação de dados e deixar o usuário navegar entre as páginas.

As interfaces `Query` e `TypedQuery` suportam paginação através dos métodos `setFirstResult` e `setMaxResults`, que definem a posição do primeiro registro (começando de 0) e o número máximo de registros que podem ser retornados, respectivamente.

```
EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Veiculo> query = manager
    .createQuery("select v from Veiculo v", Veiculo.class);
query.setFirstResult(0);
query.setMaxResults(10);

List<Veiculo> veiculos = query.getResultList();

for (Veiculo veiculo : veiculos) {
    System.out.println(veiculo.getModelo()
        + " " + veiculo.getFabricante()
        + ": " + veiculo.getAnoFabricacao());
}

manager.close();
JpaUtil.close();
```

No código acima, definimos que queremos receber apenas os 10 primeiros registros.

Agora vamos criar algo mais dinâmico, onde o usuário pode digitar o número de registros por página e navegar entre elas.

```
EntityManager manager = JpaUtil.getEntityManager();
Scanner scanner = new Scanner(System.in);

System.out.print("Registros por página: ");
```

```

int registrosPorPagina = scanner.nextInt();
int numeroDaPagina = 0;

TypedQuery<Veiculo> query = manager
    .createQuery("from Veiculo", Veiculo.class);

do {
    System.out.print("Número da página: ");
    numeroDaPagina = scanner.nextInt();

    if (numeroDaPagina != 0) {
        int primeiroRegistro = (numeroDaPagina - 1) * registrosPorPagina;

        query.setFirstResult(primeiroRegistro);
        query.setMaxResults(registrosPorPagina);
        List<Veiculo> veiculos = query.getResultList();

        for (Veiculo veiculo : veiculos) {
            System.out.println(veiculo.getModelo() + " "
                + veiculo.getFabricante()
                + ": " + veiculo.getAnoFabricacao());
        }
    }
} while (numeroDaPagina != 0);

scanner.close();
manager.close();
JpaUtil.close();

```

6.6. Projeções

Projeções é uma técnica muito útil para quando precisamos de apenas algumas poucas informações de entidades.

Dependendo de como o mapeamento é feito, o provedor JPA pode gerar uma *query* SQL grande e complexa para buscar o estado da entidade, podendo deixar o sistema lento.

Suponha que precisamos listar apenas os modelos dos veículos que temos armazenados. Não há necessidade de outras informações dos veículos.

```
EntityManager manager = JpaUtil.getEntityManager();
```

```

TypedQuery<Veiculo> query = manager
    .createQuery("select v from Veiculo v", Veiculo.class);

List<Veiculo> veiculos = query.getResultList();

for (Veiculo veiculo : veiculos) {
    System.out.println(veiculo.getModelo());
}

manager.close();
JpaUtil.close();

```

Tendo em vista que precisamos apenas das descrições dos modelos dos veículos, essa consulta é muito cara!

Além de executar uma *query* SQL buscando todas as informações de veículos, o provedor ainda executou outras *queries* para buscar os proprietários deles.

Hibernate:

```

select
    veiculo0_.codigo as codigo1_2_,
    veiculo0_.ano_fabricacao as ano_fabr2_2_,
    veiculo0_.ano_modelo as ano_mode3_2_,
    veiculo0_.data_cadastro as data_cad4_2_,
    veiculo0_.fabricante as fabrican5_2_,
    veiculo0_.modelo as modelo6_2_,
    veiculo0_.cod_proprietario as cod_prop9_2_,
    veiculo0_.tipo_combustivel as tipo_com7_2_,
    veiculo0_.valor as valor8_2_
from
    veiculo veiculo0_

```

Hibernate:

```

select
    proprietario0_.codigo as codigo1_1_0_,
    proprietario0_.email_proprietario as email_pr2_1_0_,
    proprietario0_.nome_proprietario as nome_pro3_1_0_,
    proprietario0_.telefone_proprietario as telefone4_1_0_
from
    proprietario proprietario0_
where
    proprietario0_.codigo=?

```

Hibernate:

```

select
    proprietario0.codigo as codigol1_1_0_,
    proprietario0.email_proprietario as email_pr2_1_0_,
    proprietario0.nome_proprietario as nome_pro3_1_0_,
    proprietario0.telefone_proprietario as telefone4_1_0_
from
    proprietario proprietario0_
where
    proprietario0.codigo=?

```

Podemos *projetar* apenas a propriedade da entidade que nos interessa usando a cláusula `select`.

```

EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<String> query = manager.createQuery(
    "select modelo from Veiculo", String.class);

List<String> modelos = query.getResultList();

for (String modelo : modelos) {
    System.out.println(modelo);
}

manager.close();
JpaUtil.close();

```

A *query* SQL executada é muito mais simples:

```

Hibernate:
select
    veiculo0_.modelo as col_0_0_
from
    veiculo veiculo0_

```

6.7. Resultados complexos e o operador `new`

Quando uma *query* projeta mais de uma propriedade ou expressão na cláusula `select`, o resultado da consulta é um `List<Object[]>`, ou seja, uma lista de vetores de objetos.

Em nosso exemplo, buscaremos todos os nomes e valores de veículos e precisaremos trabalhar com posições de arrays para separar cada informação.

```
EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Object[]> query = manager.createQuery(
    "select modelo, valor from Veiculo", Object[].class);

List<Object[]> resultado = query.getResultList();

for (Object[] valores : resultado) {
    String modelo = (String) valores[0];
    BigDecimal valor = (BigDecimal) valores[1];
    System.out.println(modelo + " - " + valor);
}

manager.close();
JpaUtil.close();
```

O operador new

Trabalhar com vetores de objetos não é nada agradável, é propenso a erros e deixa o código muito feio.

Podemos usar uma classe para representar o resultado da consulta com os atributos que desejamos e um construtor que recebe como parâmetro os dois valores.

Basta usarmos o operador *new* na *query* JPQL, informando o nome completo da classe que criamos para representar o resultado da consulta (incluindo o nome do pacote) e passando como parâmetro do construtor as propriedades *modelo* e *valor*.

```
EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<PrecoVeiculo> query = manager.createQuery(
    "select new com.algaworks.lojaveiculos.dto.PrecoVeiculo("
        + "modelo, valor) from Veiculo", PrecoVeiculo.class);

List<PrecoVeiculo> precos = query.getResultList();

for (PrecoVeiculo preco : precos) {
```



```

        System.out.println(preco.getModelo()
            + " - " + preco.getValor());
    }

    manager.close();
    JpaUtil.close();

```

A consulta SQL não muda em nada, mas o provedor JPA instancia e retorna objetos do tipo que especificamos.

6.8. Funções de agregação

A sintaxe para funções de agregação em JPQL é similar a SQL. Você pode usar as funções avg, count, min, max e sum e agrupar os resultados com a cláusula group by.

Para nosso exemplo, usaremos a classe chamada TotalCarroPorAno, para representar o resultado da consulta.

Na *query* JPQL, usamos as funções de agregação avg e count, além da cláusula group by.

```

EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<TotalCarroPorAno> query = manager.createQuery(
    "select new com.algaworks.lojaveiculos.dto.TotalCarroPorAno("
        + "v.anoFabricacao, avg(v.valor), count(v)) "
        + "from Veiculo v group by v.anoFabricacao",
    TotalCarroPorAno.class);

List<TotalCarroPorAno> resultado = query.getResultList();

for (TotalCarroPorAno valores : resultado) {
    System.out.println("Ano: " + valores.getAnoFabricacao()
        + " - Preço médio: " + valores.getMediaPreco()
        + " - Quantidade: " + valores.getQuantidadeCarros());
}

manager.close();
JpaUtil.close();

```

Veja a consulta SQL executada pelo provedor JPA:

```

Hibernate:
    select
        veiculo0_.ano_fabricacao as col_0_0_,
        avg(veiculo0_.valor) as col_1_0_,
        count(veiculo0_.codigo) as col_2_0_
    from
        tab_veiculo veiculo0_
    group by
        veiculo0_.ano_fabricacao

```

6.9. Queries que retornam um resultado único

As interfaces `Query` e `TypedQuery` fornecem o método `getSingleResult`, que deve ser usado quando estamos esperando que a consulta retorne apenas um resultado.

```

EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Long> query = manager.createQuery(
    "select count(v) from Veiculo v", Long.class);

Long quantidadeVeiculos = query.getSingleResult();

System.out.println("Quantidade de veículos: " + quantidadeVeiculos);

manager.close();
JpaUtil.close();

```

O código acima busca a quantidade de veículos cadastrados e atribui à variável `quantidadeVeiculos`, do tipo `Long`.

6.10. Associações e joins

Quando precisamos combinar resultados de mais de uma entidade, precisamos usar *join*. Os *joins* da JPQL são equivalentes aos da SQL, com a diferença que, em JPQL, trabalhamos com entidades, e não tabelas.

Inner join

Para fazer *inner join* entre duas entidades, podemos usar o operador `inner join` no relacionamento entre elas.

Queremos buscar todos os proprietários que possuem veículos. Veja o exemplo:

```
EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Proprietario> query = manager.createQuery(
    "select p from Proprietario p inner join p.veiculos v",
    Proprietario.class);
List<Proprietario> proprietarios = query.getResultList();

for (Proprietario proprietario : proprietarios) {
    System.out.println(proprietario.getNome());
}

manager.close();
JpaUtil.close();
```

Veja a consulta SQL gerada:

```
Hibernate:
select
    proprietar0.codigo as codigol_1_,
    proprietar0.email_proprietario as email_pr2_1_,
    proprietar0.nome_proprietario as nome_pro3_1_,
    proprietar0.telefone_proprietario as telefone4_1_
from
    proprietario proprietar0_
inner join
    veiculo veiculos1_
        on proprietar0_.codigo=veiculos1_.cod_proprietario
```

Se um proprietário possuir dois ou mais veículos, ele repetirá no resultado da consulta, por isso é melhor usarmos o operador `distinct`.

```
select distinct p
from Proprietario p
inner join p.veiculos v
```

Veja outra forma de fazer a consulta retornar o mesmo resultado:

```
select distinct p
from Veiculo v
inner join v.proprietario p
```

Left join

Em nosso próximo exemplo, consultaremos a quantidade de veículos que cada proprietário possui. Usamos a função de agregação count e a cláusula group by, que já estudamos anteriormente.

```
EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Object[]> query = manager.createQuery(
    "select p.nome, count(v) from Proprietario p "
    + "inner join p.veiculos v group by p.nome",
    Object[].class);
List<Object[]> resultado = query.getResultList();

for (Object[] valores : resultado) {
    System.out.println(valores[0] + " - " + valores[1]);
}

manager.close();
JpaUtil.close();
```

Veja a consulta SQL gerada:

```
Hibernate:
select
    proprietar0_.nome_proprietario as col_0_0_,
    count(veiculos1_.codigo) as col_1_0_
from
    proprietario proprietar0_
inner join
    veiculo veiculos1_
    on proprietar0_.codigo=veiculos1_.cod_proprietario
group by
    proprietar0_.nome_proprietario
```

Nossa *query* funciona muito bem para proprietários que possuem veículos, mas aqueles que não possuem não aparecem no resultado da consulta. Queremos exibir também os nomes de proprietários que não possuem nenhum veículo, pois

alguém pode ter excluído do sistema ou ele já pode ter vendido seu veículo, mas o cadastro continua ativo. Por isso, precisamos usar o operador `left join`.

```
select p.nome, count(v)
from Proprietario p
left join p.veiculos v
group by p.nome
```

Agora sim, a *query* SQL gerada retorna também os proprietários sem veículos, se houver.

Hibernate:

```
select
    proprietar0_.nome_proprietario as col_0_0_,
    count(veiculos1_.codigo) as col_1_0_
from
    proprietario proprietar0_
left outer join
    veiculo veiculos1_
        on proprietar0_.codigo=veiculos1_.cod_proprietario
group by
    proprietar0_.nome_proprietario
```

Problema do N+1

Um problema bastante conhecido é o **Problema do N+1**. Esse *anti-pattern* ocorre quando pesquisamos entidades e seus relacionamentos também são carregados na sequência. Cada objeto retornado gera pelo menos mais uma nova consulta para pesquisar os relacionamentos. Veja um exemplo:

```
EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Veiculo> query = manager
    .createQuery("select v from Veiculo v", Veiculo.class);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo veiculo : veiculos) {
    System.out.println(veiculo.getModelo() + " - "
        + veiculo.getProprietario().getNome());
}

manager.close();
```

```
JpaUtil.close();
```

Embora esse código funcione, ele é extremamente ineficiente, pois diversas consultas SQL são geradas para buscar o proprietário do veículo.

Hibernate:

```
select
    veiculo0_.codigo as codigol_2_,
    veiculo0_.ano_fabricacao as ano_fabr2_2_,
    veiculo0_.ano_modelo as ano_mode3_2_,
    veiculo0_.data_cadastro as data_cad4_2_,
    veiculo0_.fabricante as fabrican5_2_,
    veiculo0_.modelo as modelo6_2_,
    veiculo0_.cod_proprietario as cod_prop9_2_,
    veiculo0_.tipo_combustivel as tipo_com7_2_,
    veiculo0_.valor as valor8_2_
from
    veiculo veiculo0_
```

Hibernate:

```
select
    proprietario0_.codigo as codigol_1_0_,
    proprietario0_.email_proprietario as email_pr2_1_0_,
    proprietario0_.nome_proprietario as nome_pro3_1_0_,
    proprietario0_.telefone_proprietario as telefone4_1_0_
from
    proprietario proprietario0_
where
    proprietario0_.codigo=?
```

Hibernate:

```
select
    proprietario0_.codigo as codigol_1_0_,
    proprietario0_.email_proprietario as email_pr2_1_0_,
    proprietario0_.nome_proprietario as nome_pro3_1_0_,
    proprietario0_.telefone_proprietario as telefone4_1_0_
from
    proprietario proprietario0_
where
    proprietario0_.codigo=?
```

```
...
...
...
```

Para solucionar o problema, podemos usar os operadores `inner join fetch` para fazer *join* com a entidade `Proprietario` e trazer os dados na consulta.

```
EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Veiculo> query = manager.createQuery(
    "select v from Veiculo v inner join fetch v.proprietario",
    Veiculo.class);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo veiculo : veiculos) {
    System.out.println(veiculo.getModelo() + " - "
        + veiculo.getProprietario().getNome());
}

manager.close();
JpaUtil.close();
```

Agora apenas um comando SQL é executado, deixando nosso código muito mais eficiente.

```
Hibernate:
    select
        veiculo0_.codigo as codigol_2_0_,
        proprietarl_.codigo as codigol_1_1_,
        veiculo0_.ano_fabricacao as ano_fabr2_2_0_,
        veiculo0_.ano_modelo as ano_mode3_2_0_,
        veiculo0_.data_cadastro as data_cad4_2_0_,
        veiculo0_.fabricante as fabrican5_2_0_,
        veiculo0_.modelo as modelo6_2_0_,
        veiculo0_.cod_proprietario as cod_prop9_2_0_,
        veiculo0_.tipo_combustivel as tipo_com7_2_0_,
        veiculo0_.valor as valor8_2_0_,
        proprietarl_.email_proprietario as email_pr2_1_1_,
        proprietarl_.nome_proprietario as nome_pro3_1_1_,
        proprietarl_.telefone_proprietario as telefone4_1_1_
    from
        veiculo veiculo0_
    inner join
        proprietario proprietarl_
        on veiculo0_.cod_proprietario=proprietarl_.codigo
```

Aprenda mais sobre o Problema do N+1

No blog da AlgaWorks tem um artigo sobre o **Problema do N+1**, com uma simulação do problema e a correção dele em um projeto web.

<https://blog.algaworks.com/o-problema-do-n-mais-um/>

6.11. Queries nomeadas

Queries nomeadas, também conhecidas como *named queries*, é uma forma de organizar as consultas JPQL que escrevemos em nossas aplicações.

Além de organizar as *queries*, ganhamos em performance, pois elas são estáticas e processadas apenas na inicialização da unidade de persistência.

Uma *named query* é definida com a anotação `@NamedQuery`, que pode ser colocada na declaração da classe de qualquer entidade JPA. A anotação recebe o nome da *query* e a própria consulta JPQL.

```
@Entity
@Table(name = "tab_veiculo")
@NamedQuery(name = "Veiculo.comProprietarioPorValor",
    query = "select v from Veiculo v "
        + "inner join fetch v.proprietario where v.valor > :valor")
public class Veiculo {

    ...

}
```

Para facilitar e evitar conflitos, nomeamos a *query* com um prefixo “Veiculo”, dizendo que a consulta está relacionada a essa entidade.

Para usar uma *named query*, chamamos o método `createNamedQuery` de `EntityManager`.


```

EntityManager manager = JpaUtil.getEntityManager();

TypedQuery<Veiculo> query = manager.createNamedQuery(
    "Veiculo.comProprietarioPorValor", Veiculo.class);
query.setParameter("valor", new BigDecimal(10_000));

List<Veiculo> veiculos = query.getResultList();

for (Veiculo veiculo : veiculos) {
    System.out.println(veiculo.getModelo() + " - "
        + veiculo.getProprietario().getNome());
}

manager.close();
JpaUtil.close();

```

Para definir mais de uma *named query* em uma entidade, podemos agrupá-las com a anotação `@NamedQueries`.

```

@Entity
@Table(name = "veiculo")
@NamedQueries({
    @NamedQuery(name = "Veiculo.comProprietarioPorValor",
        query = "select v from Veiculo v "
            + "inner join fetch v.proprietario where v.valor > :valor"),
    @NamedQuery(name = "Veiculo.porModelo",
        query = "select v from Veiculo v where modelo like :modelo")
})
public class Veiculo {

    // atributos e métodos
}

```

Queries nomeadas em arquivos externos

Você já viu que definir *named queries* é fácil, deixa nosso código mais limpo e ganhamos performance. Se você quiser organizar ainda mais, eliminando totalmente as *queries* do código Java, pode externalizá-las para arquivos XML.

Podemos criar um arquivo chamado *orm.xml* em *META-INF* com o seguinte conteúdo:

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

```

<entity-mappings
  xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
  http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd"
  version="2.2">

  <named-query name="Veiculo.anoFabricacaoMenor">
    <query><![CDATA[
      select v from Veiculo v where anoFabricacao < :ano
    ]]></query>
  </named-query>

</entity-mappings>

```

No arquivo acima, definimos uma *named query* chamada “Veiculo.anoFabricacaoMenor”, e podemos usá-la normalmente.

Artigo sobre *named queries* em arquivos externos

Para não ficar qualquer dúvida, veja este artigo no blog da AlgaWorks que explica como usar *named queries* em arquivos externos.

<https://blog.algaworks.com/named-queries-em-arquivos-externos/>

6.12. Queries SQL nativas

JPQL é flexível suficiente para executar quase todas as *queries* que você precisará e tem a vantagem de usar o modelo de entidades mapeadas.

Existem alguns casos bastante específicos em que você precisará utilizar SQL nativo para consultar dados em seu banco de dados.

No exemplo abaixo, pesquisamos todos os veículos usando SQL, que retorna entidades gerenciadas pelo contexto de persistência.

```
EntityManager manager = JpaUtil.getEntityManager();
```

```

Query query = manager.createNativeQuery(
    "select * from tab_veiculo", Veiculo.class);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo veiculo : veiculos) {
    System.out.println(veiculo.getModelo() + " - "
        + veiculo.getProprietario().getNome());
}

manager.close();
JpaUtil.close();

```

As colunas retornadas pela *query* devem coincidir com os nomes definidos no mapeamento. É possível configurar o mapeamento do *result set* usando `@SqlResultSetMapping`, mas não entraremos em detalhes.

Criteria API

7.1. Introdução e estrutura básica

A Criteria API da JPA é usada para definir *queries* dinâmicas, criadas a partir de objetos que definem uma consulta, ao invés de puramente texto, como a JPQL.

A principal vantagem da Criteria API é poder construir consultas programaticamente, de forma elegante e com maior integração com a linguagem Java.

Mas antes de começar, só um aviso: as entidades e DTOs usadas nesse capítulo serão as mesmas que foram apresentadas no início do capítulo sobre JPQL.

Vamos começar com uma das consultas mais simples que poderíamos fazer usando Criteria API.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(Veiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
criteriaQuery.select(veiculo);

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo v : veiculos) {
```

```
        System.out.println(v.getModelo());  
    }  
  
    manager.close();  
    JpaUtil.close();
```

A consulta acima busca todos os veículos cadastrados e imprime o modelo deles. Seria o mesmo que fazer em JPQL:

```
select v from Veiculo v
```

Claro que, em uma consulta simples como essa, seria melhor usarmos JPQL, mas você conhecerá a facilidade da API para alguns casos mais a frente.

Primeiramente, pegamos uma instância do tipo `CriteriaBuilder` do `EntityManager`, através do método `getCriteriaBuilder`. Essa interface funciona como uma fábrica de vários objetos que podemos usar para definir uma consulta.

Usamos o método `createQuery` de `CriteriaBuilder` para instanciar um `CriteriaQuery`. A interface `CriteriaQuery` possui as cláusulas da consulta.

Chamamos o método `from` de `CriteriaQuery` para obtermos um objeto do tipo `Root`. Depois, chamamos o método da cláusula `select`, informando como parâmetro o objeto do tipo `Root`, dizendo que queremos selecionar a entidade `Veiculo`.

Criamos uma `TypedQuery` através do método `EntityManager.createQuery`, e depois recuperamos o resultado da consulta pelo método `getResultList`.

Bastante burocrático, né? Mas esse é o preço que pagamos para ter uma API muito dinâmica.

7.2. Filtros e queries dinâmicas

A cláusula *where* é uma parte importante de consultas, pois definem as condições (predicados) que filtram o resultado.

Para filtrar o resultado usando Criteria API, precisamos de objetos do tipo `Predicate` para passar para a cláusula *where*. Um objeto do tipo `Predicate` é obtido

através do CriteriaBuilder.

No exemplo abaixo, consultamos todos os veículos que o tipo de combustível não seja diesel.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(Veiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
Predicate predicate = builder.not(builder.equal(veiculo.get("tipoCombustivel"),
    TipoCombustivel.DIESEL));

criteriaQuery.select(veiculo);
criteriaQuery.where(predicate);

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo v : veiculos) {
    System.out.println(v.getModelo());
}

manager.close();
JpaUtil.close();
```

Queries dinâmicas

A grande vantagem de Criteria API é poder criar *queries* dinâmicas, com filtros condicionais, por exemplo. Neste caso, não sabemos qual será a estrutura final da consulta, pois ela é montada em tempo de execução.

No exemplo abaixo, criamos um método `pesquisarVeiculos` que retorna uma lista de veículos, dado o tipo do combustível e o valor máximo. É permitido passar null para qualquer um dos parâmetros do método, portanto, os filtros da *query* devem ser montados programaticamente.

```
public static List<Veiculo> pesquisarVeiculos(
    TipoCombustivel tipoCombustivel, BigDecimal maiorValor) {
    EntityManager manager = JpaUtil.getEntityManager();
```

```

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(
    Veiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
criteriaQuery.select(veiculo);

List<Predicate> predicates = new ArrayList<>();

if (tipoCombustivel != null) {
    ParameterExpression<TipoCombustivel> paramTipoCombustivel =
        builder.parameter(TipoCombustivel.class, "tipoCombustivel");
    predicates.add(builder.equal(veiculo.get("tipoCombustivel"),
        paramTipoCombustivel));
}

if (maiorValor != null) {
    ParameterExpression<BigDecimal> paramValor = builder.parameter(
        BigDecimal.class, "maiorValor");
    predicates.add(builder.lessThanOrEqualTo(
        veiculo.<BigDecimal>get("valor"), paramValor));
}

criteriaQuery.where(predicates.toArray(new Predicate[0]));

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);

if (tipoCombustivel != null) {
    query.setParameter("tipoCombustivel", tipoCombustivel);
}

if (maiorValor != null) {
    query.setParameter("maiorValor", maiorValor);
}

List<Veiculo> veiculos = query.getResultList();
manager.close();

return veiculos;
}

public static void main(String[] args) {
    List<Veiculo> veiculos = pesquisarVeiculos(
        TipoCombustivel.BICOMBUSTIVEL, new BigDecimal(50_000));

    for (Veiculo v : veiculos) {

```

```

        System.out.println(v.getModelo() + " - " + v.getValor());
    }
}

```

Aproveitamos o exemplo para mostrar também o uso de parâmetros usando o método `CriteriaBuilder.parameter`, que recebe como argumento o tipo do parâmetro e o nome dele, que depois é definido pelo método `TypedQuery.setParameter`.

7.3. Projeções

Suponha que precisamos listar apenas os modelos dos veículos que temos armazenados. Podemos projetar essa propriedade chamando o método `Root.get` e passando para a cláusula *select*.

```

EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<String> criteriaQuery = builder.createQuery(String.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
criteriaQuery.select(veiculo.<String>get("modelo"));

TypedQuery<String> query = manager.createQuery(criteriaQuery);
List<String> modelos = query.getResultList();

for (String modelo : modelos) {
    System.out.println(modelo);
}

manager.close();
JpaUtil.close();

```

7.4. Funções de agregação

As funções de agregação são representadas por métodos do `CriteriaBuilder`, incluindo `max`, `greatest`, `min`, `least`, `avg`, `sum`, `sumAsLong`, `sumAsDouble`, `count` e `countDistinct`.

No exemplo abaixo, buscamos a soma dos valores de todos os veículos armazenados.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<BigDecimal> criteriaQuery = builder.createQuery(
    BigDecimal.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
criteriaQuery.select(builder.sum(veiculo.<BigDecimal>get("valor")));

TypedQuery<BigDecimal> query = manager.createQuery(criteriaQuery);
BigDecimal total = query.getSingleResult();

System.out.println("Valor total: " + total);

manager.close();
JpaUtil.close();
```

7.5. Resultados complexos, tuplas e construtores

Quando projetamos mais de uma propriedade em uma consulta, podemos configurar como desejamos receber o resultado, sendo: uma lista de `Object[]`, uma lista de `Tuple` ou uma lista de um objeto de uma classe qualquer.

Lista de `Object[]`

No exemplo abaixo, projetamos duas propriedades de `Veiculo` usando o método `multiselect` de `CriteriaQuery` e obtemos o resultado da consulta como um `List<Object[]>`.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Object[]> criteriaQuery = builder.createQuery(Object[].class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
criteriaQuery.multiselect(veiculo.<String>get("modelo"),
    veiculo.<String>get("valor"));
```

```

TypedQuery<Object[]> query = manager.createQuery(criteriaQuery);
List<Object[]> resultado = query.getResultList();

for (Object[] valores : resultado) {
    System.out.println(valores[0] + " - " + valores[1]);
}

manager.close();
JpaUtil.close();

```

Lista de tuplas

Trabalhar com índices de arrays deixa o código feio e a chance de errar é muito maior. Podemos retornar um *array* de *Tuple*, onde cada tupla representa um registro encontrado.

```

EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Tuple> criteriaQuery = builder.createTupleQuery();

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
criteriaQuery.multiselect(
    veiculo.<String>get("modelo").alias("modeloVeiculo"),
    veiculo.<String>get("valor").alias("valorVeiculo"));

TypedQuery<Tuple> query = manager.createQuery(criteriaQuery);

List<Tuple> resultado = query.getResultList();

for (Tuple tupla : resultado) {
    System.out.println(tupla.get("modeloVeiculo")
        + " - " + tupla.get("valorVeiculo"));
}

manager.close();
JpaUtil.close();

```

Veja que chamamos o método `createTupleQuery` para receber uma instância de `CriteriaQuery` e apelidamos cada propriedade projetada com o método `alias`. Esses apelidos foram usados na iteração, através do método `get` da tupla.

Construtores

O jeito mais elegante de retornar um resultado projetado é criando uma classe para representar os valores de cada tupla, com um construtor que recebe os valores e atribui às variáveis de instância.

Basta criar a consulta e passar como parâmetro do método `select` o retorno de `CriteriaBuilder.construct`. Este último método deve receber a classe de retorno e as propriedades projetadas.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<PrecoVeiculo> criteriaQuery = builder
    .createQuery(PrecoVeiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
criteriaQuery.select(builder.construct(PrecoVeiculo.class,
    veiculo.<String>get("modelo"), veiculo.<String>get("valor")));

TypedQuery<PrecoVeiculo> query = manager.createQuery(criteriaQuery);
List<PrecoVeiculo> resultado = query.getResultList();

for (PrecoVeiculo tupla : resultado) {
    System.out.println(tupla.getModelo() + " - " + tupla.getValor());
}

manager.close();
JpaUtil.close();
```

7.6. Funções

A Criteria API suporta diversas funções de banco de dados, que estão definidas em `CriteriaBuilder`.

No exemplo abaixo usamos a função `upper` para passar o modelo do veículo para maiúsculo e fazer uma comparação *case-insensitive*.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
```

```

CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(Veiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
Predicate predicate = builder.equal(builder.upper(
    veiculo.<String>get("modelo")), "Gol".toUpperCase());

criteriaQuery.select(veiculo);
criteriaQuery.where(predicate);

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo v : veiculos) {
    System.out.println(v.getModelo());
}

manager.close();
JpaUtil.close();

```

Já no próximo exemplo, programamos uma consulta que retorna uma lista de *strings*, com o fabricante e modelo concatenados e separados pelo caractere “-”.

```

EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<String> criteriaQuery = builder.createQuery(String.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);

Expression<String> expression = builder.concat(builder.concat(
    veiculo.<String> get("fabricante"), " - "),
    veiculo.<String> get("modelo"));
criteriaQuery.select(expression);

TypedQuery<String> query = manager.createQuery(criteriaQuery);
List<String> veiculos = query.getResultList();

for (String v : veiculos) {
    System.out.println(v);
}

manager.close();
JpaUtil.close();

```

7.7. Ordenação de resultado

Para ordenar o resultado de uma consulta podemos usar o método `orderBy` de `CriteriaQuery`. Precisamos passar como parâmetro para esse método um objeto do tipo `Order`, que é instanciado usando `CriteriaBuilder.desc`, para ordenação decrescente. Poderíamos usar `asc` para ordenação crescente.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(Veiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
Order order = builder.desc(veiculo.<String>get("anoFabricacao"));

criteriaQuery.select(veiculo);
criteriaQuery.orderBy(order);

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo v : veiculos) {
    System.out.println(v.getModelo() + " - " + v.getAnoFabricacao());
}

manager.close();
JpaUtil.close();
```

7.8. Join e fetch

Para fazer *join* entre duas entidades podemos usar seus relacionamentos mapeados. No exemplo abaixo, chamamos o método `join` de um objeto do tipo `From`. Depois, filtramos a consulta a partir da entidade usada pelo *join*.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(Veiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
Join<Veiculo, Proprietario> proprietario = veiculo.join("proprietario");
```

```

criteriaQuery.select(veiculo);
criteriaQuery.where(builder.equal(proprietario.get("nome"),
    "Fernando Martins"));

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo v : veiculos) {
    System.out.println(v.getModelo() + " - " +
        v.getProprietario().getNome());
}

manager.close();
JpaUtil.close();

```

Fetch

Para evitar o **Problema do N+1**, que já estudamos anteriormente, podemos fazer um *fetch* no relacionamento da entidade Veiculo.

```

EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(Veiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
Join<Veiculo, Proprietario> proprietario = (Join) veiculo.fetch(
    "proprietario");

criteriaQuery.select(veiculo);
criteriaQuery.where(builder.equal(proprietario.get("nome"),
    "Fernando Martins"));

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo v : veiculos) {
    System.out.println(v.getModelo() + " - " +
        v.getProprietario().getNome());
}

manager.close();
JpaUtil.close();

```

Todos os dados necessários são retornados por apenas uma consulta SQL:

```
select
    veiculo0_.codigo as codigol_9_0_,
    proprietario1_.codigo as codigol_7_1_,
    veiculo0_.ano_fabricacao as ano_fabr2_9_0_,
    veiculo0_.ano_modelo as ano_mode3_9_0_,
    veiculo0_.data_cadastro as data_cad4_9_0_,
    veiculo0_.especificacoes as especifici5_9_0_,
    veiculo0_.fabricante as fabrican6_9_0_,
    veiculo0_.foto as foto7_9_0_,
    veiculo0_.modelo as modelo8_9_0_,
    veiculo0_.proprietario_codigo as propriel1_9_0_,
    veiculo0_.tipo_combustivel as tipo_com9_9_0_,
    veiculo0_.valor as valorl0_9_0_,
    proprietario1_.email as email2_7_1_,
    proprietario1_.nome as nome3_7_1_
from
    tab_veiculo veiculo0_
inner join
    proprietario proprietario1_
        on veiculo0_.proprietario_codigo=proprietar1_.codigo
where
    proprietario1_.nome=?
```

7.9. Subqueries

Subqueries podem ser usadas pela Criteria API nas cláusulas *select*, *where*, *order*, *group by* e *having*. Para criar uma *subquery*, chamamos o método `CriteriaQuery.subquery`.

No exemplo abaixo, consultamos os veículos que possuem valores a partir do valor médio de todos os veículos armazenados.

```
EntityManager manager = JpaUtil.getEntityManager();

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(Veiculo.class);
Subquery<Double> subquery = criteriaQuery.subquery(Double.class);
```

```

Root<Veiculo> veiculoA = criteriaQuery.from(Veiculo.class);
Root<Veiculo> veiculoB = subquery.from(Veiculo.class);

subquery.select(builder.avg(veiculoB.<Double>get("valor")));

criteriaQuery.select(veiculoA);
criteriaQuery.where(builder.greaterThanOrEqualTo(
    veiculoA.<Double>get("valor"), subquery));

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo v : veiculos) {
    System.out.println(v.getModelo() + " - " + v.getProprietario().getNome());
}

manager.close();
JpaUtil.close();

```

Veja a consulta SQL gerada:

```

select
    veiculo0_.codigo as codigo1_2_,
    veiculo0_.ano_fabricacao as ano_fabr2_2_,
    veiculo0_.ano_modelo as ano_mode3_2_,
    veiculo0_.data_cadastro as data_cad4_2_,
    veiculo0_.fabricante as fabrican5_2_,
    veiculo0_.modelo as modelo6_2_,
    veiculo0_.cod_proprietario as cod_prop9_2_,
    veiculo0_.tipo_combustivel as tipo_com7_2_,
    veiculo0_.valor as valor8_2_
from
    veiculo veiculo0_
where
    veiculo0_.valor>=(
        select
            avg(veiculo1_.valor)
        from
            veiculo veiculo1_
    )

```


7.10. Metamodel

A JPA define um metamodelo que pode ser usado em tempo de execução para obter informações sobre o mapeamento ORM feito.

Esse metamodelo inclui a lista das propriedades mapeadas de uma entidade, seus tipos e cardinalidades.

O metamodelo pode ser usado com Criteria API para substituir as strings que referenciam propriedades.

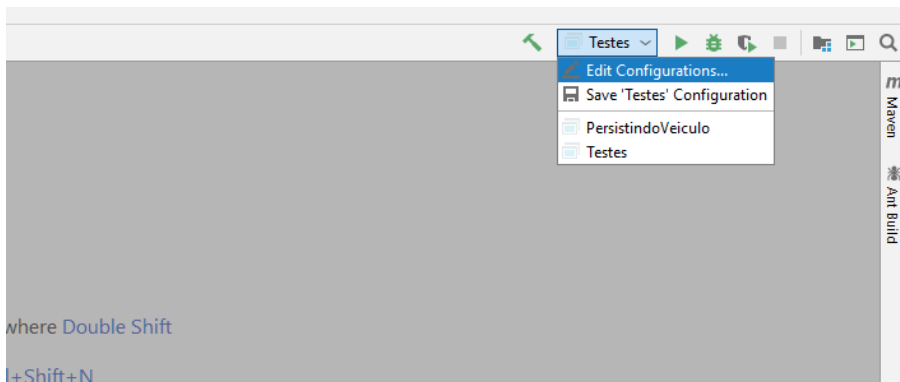
O uso de metamodelo evita erros em tempo de execução, pois qualquer alteração no código-fonte de mapeamento altera também o metamodelo.

Podemos gerar o *metamodel* automaticamente. Para isso vamos usar a dependência *hibernate-jpamodelgen*.

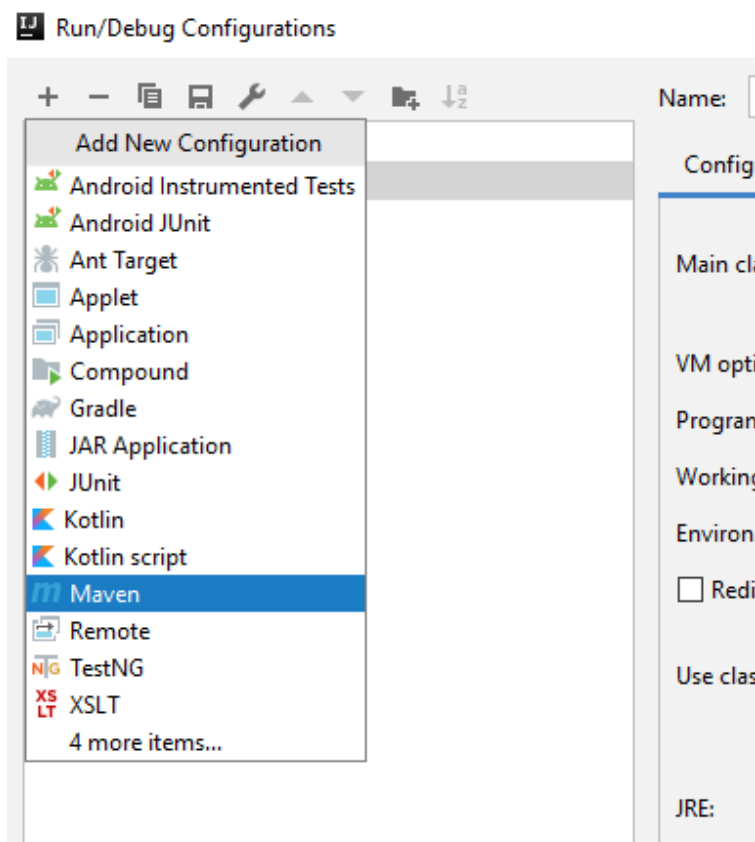
```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-jpamodelgen</artifactId>
  <version>5.4.10.Final</version>
  <scope>provided</scope>
</dependency>
```

Uma vez que a dependência foi adicionada no *pom.xml*, basta executar o projeto normalmente com o IntelliJ ou rodar o comando *package* do Maven.

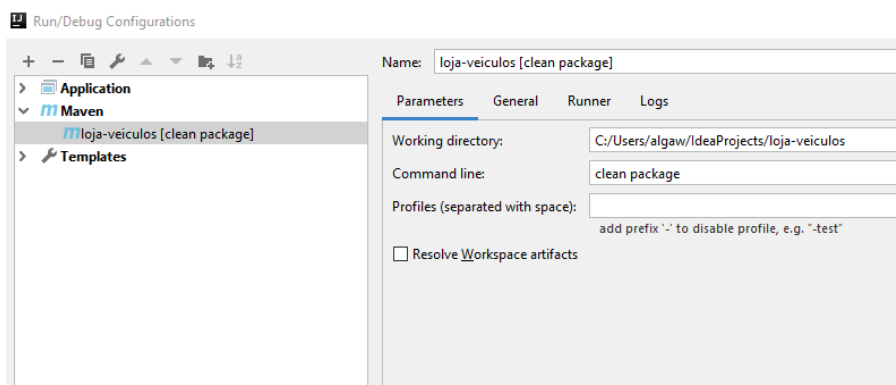
Para executar o *package* no IntelliJ, você pode selecionar *Edit Configurations...*



Na tela que se abre clique no sinal “+”, depois na opção *Maven*.

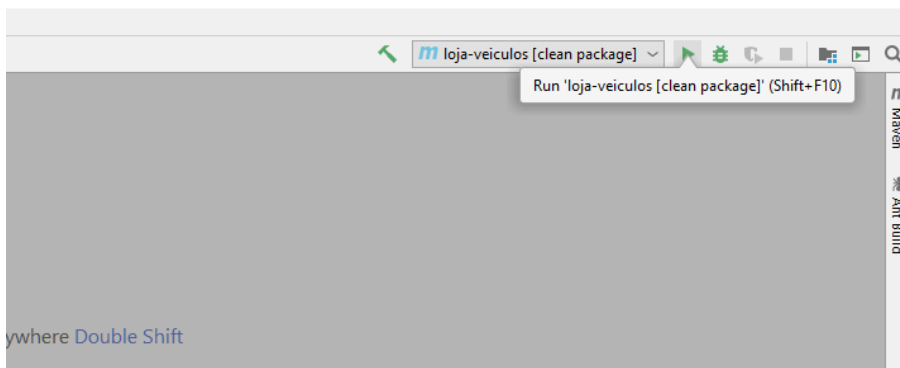


Dê um nome para essa configuração e digite o comando para ser executado. Pode nomear como *loja-veiculos [clean package]* e no campo *Command Line*, informe *clean package*.



Repare que além de *package*, foi usado também o *clean*, que serve para limpar o conteúdo da pasta *target* antes de construir o projeto.

Por fim, execute o comando que acabou de configurar.



Assim que aparecer no console a mensagem *BUILD SUCCESS*, já pode conferir no diretório *target/generated-sources/annotations/com/algaworks/lojaveiculos/dominio* que as classes do *metamodel* já estarão criadas.

O IntelliJ já vai colocar esse diretório no *path* do seu projeto para que você consiga utilizar essas classes.

Agora podemos referenciar as propriedades das entidades através das classes geradas. Por exemplo, podemos usar as classes *Veiculo_* e *Proprietario_*.

```
CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Veiculo> criteriaQuery = builder.createQuery(Veiculo.class);

Root<Veiculo> veiculo = criteriaQuery.from(Veiculo.class);
Join<Veiculo, Proprietario> proprietario = veiculo.join(
    Veiculo_.proprietario);

criteriaQuery.select(veiculo);
criteriaQuery.where(builder.equal(proprietario.get(Proprietario_.nome),
    "Fernando Martins"));

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();
```

Tente alterar o nome de uma propriedade que está sendo usada na consulta,

diretamente na classe da entidade, e veja que um erro de compilação irá aparecer, deixando evidente que existe um problema na consulta.

Capítulo 8

Conclusão

Chegamos ao final desse livro!

Nós esperamos que você tenha praticado e aprendido bastante sobre JPA.

Se você gostou desse livro, por favor, nos ajude a manter esse trabalho. Recomende para seus amigos de trabalho, faculdade e/ou compartilhe nos grupos do Facebook, WhatsApp, Telegram, etc.

8.1. Próximos passos

O conteúdo que você aprendeu aqui já é o suficiente para que você construa sua camada de persistência, mas ainda tem MUITO mais para aprender!

Caso você tenha interesse em **mergulhar fundo** em conteúdos ainda mais avançados, recomendo que você conheça e faça a sua matrícula no **Especialista JPA**, que é o nosso curso online completo e avançado:

<https://www.algaworks.com/curso/especialista-jpa/>

Além de tudo que você vai aprender, nós ainda oferecemos suporte para as suas dúvidas!

JPA

GUIA DEFINITIVO