

THIAGO FARIA
NORMANDES JR

1ª Edição 

JPA E HIBERNATE



JPA e Hibernate

por Thiago Faria e Normandes Junior

1ª Edição, 31/07/2015

© 2015 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

+55 (11) 2626-9415

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

A solid blue square containing the word "Facebook" in white, bold, sans-serif font.

Facebook

A solid red square containing the word "YouTube" in white, bold, sans-serif font.

YouTube

CURSOS ONLINE

COM VÍDEO AULAS E SUPORTE



Java e Orientação a Objetos



Desenvolvimento Web
com JSF 2



JPA e Hibernate além do básico
- um projeto completo



Sistemas Comerciais Java EE
com CDI, JPA e PrimeFaces

www.algaworks.com

Cursos presenciais in-company? Entre em contato.

Sobre os autores



Thiago Faria de Andrade

Fundador, instrutor e consultor da AlgaWorks. Graduado em Sistemas de Informação e certificado como programador Java pela Sun. Iniciou seu interesse por programação em 1995, quando desenvolveu um software para entretenimento e se tornou um dos mais populares no Brasil e outros

países de língua portuguesa. Já foi sócio e trabalhou em outras empresas de software como programador, gerente e diretor de tecnologia, mas nunca deixou de programar.

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



Normandes José Moreira Junior

Sócio e instrutor da AlgaWorks, formado em Engenharia Elétrica pela Universidade Federal de Uberlândia e detentor das certificações LPIC-1, SCJP e SCWCD. Palestrante internacional reconhecido por contribuir e liderar projetos open source.

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

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 e 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 seus estudos, temos certeza de uma coisa: você vai ter muitas dúvidas e problemas para resolver!

Nós gostaríamos muito de te ajudar pessoalmente nesses problemas e sanar todas as suas dúvidas, mas infelizmente não conseguimos fazer isso com todos os leitores do livro, afinal, ocupamos grande parte do dia ajudando os nossos 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 no StackOverflow em português. É só acessar:

<http://pt.stackoverflow.com/>

Para aumentar as chances de receber uma resposta para seu problema, veja algumas dicas:

<http://pt.stackoverflow.com/help/how-to-ask>

Ao perguntar sobre um problema no seu código, você conseguirá melhores respostas se mostrar às pessoas código que elas possam usar para reproduzir o problema. Veja algumas recomendações no link a seguir:

<http://pt.stackoverflow.com/help/mcve>

Você também pode ajudar outras pessoas com o conhecimento que adquiriu, assim todos serão ajudados de alguma forma.

Comunidade Java da AlgaWorks no Facebook

Outra forma de obter ajuda, contribuir com o seu conhecimento e manter contato com outros programadores Java é ingressando na **Comunidade Java da AlgaWorks** (Facebook). Acesse:

<http://alga.works/comunidade-java-algaworks/>

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.

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?	13
1.2	Mapeamento Objeto Relacional (ORM)	13
1.3	Porque usar ORM?	15
1.4	Java Persistence API e Hibernate	16

2 Primeiros passos com JPA e Hibernate

2.1	Download e configuração do Hibernate ORM	17
2.2	Criação do Domain Model	19
2.3	Implementação do equals() e hashCode()	21
2.4	Mapeamento básico	22
2.5	O arquivo persistence.xml	24
2.6	Gerando as tabelas do banco de dados	26
2.7	Definindo detalhes físicos de tabelas	27
2.8	Criando EntityManager	30
2.9	Persistindo objetos	31
2.10	Buscando objetos pelo identificador	34
2.11	Listando objetos	36
2.12	Atualizando objetos	38
2.13	Excluindo objetos	40

3 Gerenciando estados

3.1	Estados e ciclo de vida	42
3.2	Contexto de persistência	44
3.3	Sincronização de dados	46
3.4	Salvando objetos desanexados com merge()	49

4 Mapeamento

4.1	Identificadores	52
4.2	Chaves compostas	54
4.3	Enumerações	57
4.4	Propriedades temporais	58
4.5	Propriedades transientes	59
4.6	Objetos grandes	60
4.7	Objetos embutidos	64
4.8	Associações um-para-um	67
4.9	Associações muitos-para-um	75
4.10	Coleções um-para-muitos	77
4.11	Coleções muitos-para-muitos	79
4.12	Coleções de tipos básicos e objetos embutidos	85
4.13	Herança	90
4.14	Modos de acesso	101

5 Recursos avançados

5.1	Lazy loading e eager loading	105
5.2	Operações em cascata	113
5.3	Exclusão de objetos órfãos	117

5.4	Operações em lote	118
5.5	Concorrência e locking	121
5.6	Métodos de callback e auditores de entidades	125
5.7	Cache de segundo nível	129

6 Java Persistence Query Language

6.1	Introdução à JPQL	134
6.2	Consultas simples e iteração no resultado	140
6.3	Usando parâmetros nomeados	141
6.4	Consultas tipadas	142
6.5	Paginação	142
6.6	Projeções	144
6.7	Resultados complexos e o operador new	146
6.8	Funções de agregação	147
6.9	Queries que retornam um resultado único	148
6.10	Associações e joins	149
6.11	Queries nomeadas	154
6.12	Queries SQL nativas	156

7 Criteria API

7.1	Introdução e estrutura básica	158
7.2	Filtros e queries dinâmicas	159
7.3	Projeções	162
7.4	Funções de agregação	162
7.5	Resultados complexos, tuplas e construtores	163

7.6	Funções	165
7.7	Ordenação de resultado	166
7.8	Join e fetch	167
7.9	Subqueries	168
7.10	Metamodel	170

8 Conclusão

8.1	Próximos passos	172
-----	-----------------------	-----

Introdução

1.1. O que é persistência?

A maioria dos sistemas desenvolvidos em uma empresa precisa de dados persistentes, portanto persistência 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 sistemas gerenciadores de banco de dados relacionais e SQL, 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 de dados entre banco de dados relacionais e linguagens de programação orientada 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 de 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 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 entre objetos e banco de dados.

Uma solução ORM consiste de 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 ambiente ORM, as aplicações interagem com APIs 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 objetos a banco de dados.

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. JPA elimina muito do trabalho e deixa você se concentrar na lógica de negócio, possibilitando uma produtividade imensa para você.

A manutenabilidade de sistemas desenvolvidos com ORM é excelente, pois o mecanismo faz com que menos linhas de código sejam necessárias. Além de facilitar o entendimento, menos linhas de código deixam o sistema mais fácil de ser alterado.

Existem outras razões que fazem com que um sistema desenvolvido utilizando JPA seja melhor de ser mantido. Em sistemas com a camada de persistência desenvolvida usando JDBC e SQL, 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 readequação que pode custar caro.

ORM abstrai sua aplicação do banco de dados e do dialeto SQL. Com JPA, você pode desenvolver um sistema usando um banco de dados e colocá-lo em produção usando diversos outros bancos de dados, sem precisar alterar códigos-fontes para adequar sintaxe de queries que só funcionam em SGBDs de determinados fornecedores.

1.4. Java Persistence API e Hibernate

A *Java Persistence API* (JPA) é um framework para persistência em Java, que oferece uma API de mapeamento objeto-relacional e soluções para integrar persistência com sistemas corporativos escaláveis.

Com JPA, os objetos são POJO (*Plain Old Java Objects*), ou seja, não é necessário nada de 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.

JPA é uma especificação, e não um produto. Para trabalhar com JPA, precisamos de uma implementação.

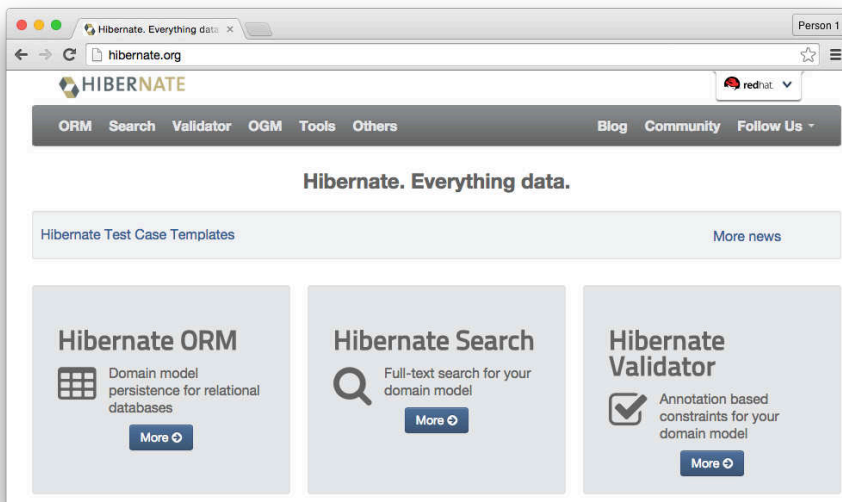
O projeto do Hibernate ORM possui alguns módulos, sendo que o **Hibernate EntityManager** é a implementação da JPA que encapsula o Hibernate Core.

O **Hibernate Core** é a base para o funcionamento da persistência, com APIs nativas e metadados de mapeamentos em arquivos XML, uma linguagem de consultas chamada HQL (parecido com SQL), um conjunto de interfaces para consultas usando critérios (Criteria API), etc.

Primeiros passos com JPA e Hibernate

2.1. Download e configuração do Hibernate ORM

Antes de criarmos um projeto, precisamos fazer o download do Hibernate ORM no site www.hibernate.org. O arquivo fornecido no site possui todos os módulos e dependências necessárias.



Depois de concluir o download, descompacte o arquivo em um diretório temporário.

Crie um novo projeto no Eclipse do tipo “Java Project” e crie uma pasta dentro dele chamada “lib”, para que possamos colocar os JARs necessários.

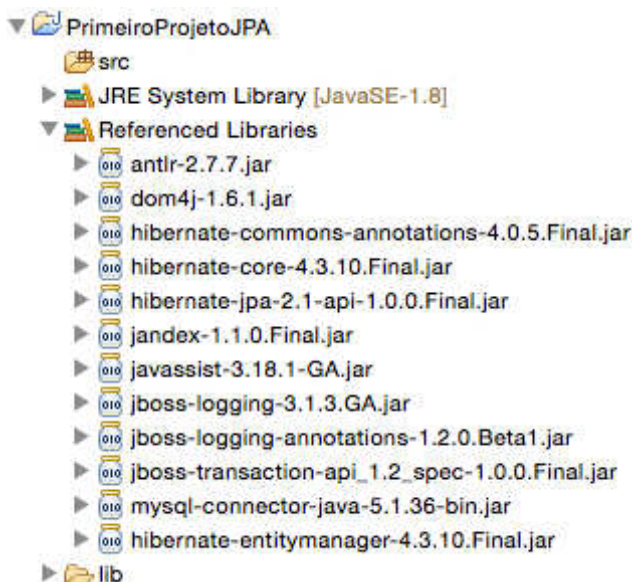
Copie todos os arquivos das pastas “required” e “jpa” para a pasta “lib” do projeto.

Estas bibliotecas que você acabou de copiar fazem parte da distribuição do Hibernate ORM e são obrigatórias para seu funcionamento. As demais são requeridas apenas para casos específicos.

Como usaremos o MySQL neste livro, precisaremos colocar também o driver JDBC do MySQL na pasta “lib”. Baixe em <http://dev.mysql.com/downloads/connector/j/>.

Agora que temos todas as bibliotecas necessárias na pasta “lib” do projeto, precisamos adicioná-las ao *Build Path* do projeto. Encontre e selecione todos os arquivos JAR que estão dentro do seu projeto no Eclipse, clique com o botão direito, acesse a opção “Build Path” e “Add to Build Path”.

Seu projeto deve ficar com as seguintes bibliotecas referenciadas:



2.2. Criação do Domain Model

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

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

```
package com.algaworks.veiculos.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 persistentes devem seguir o estilo de JavaBeans, com métodos *getters* e *setters*. É obrigatório que esta classe possua um construtor sem argumentos.

Como você pode ver, a classe persistente *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 concessionária, sem acumular todos os recursos estudados, para não dificultar o entendimento de cada um desses recursos.

2.3. Implementação do `equals()` e `hashCode()`

Para que os objetos persistentes 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 de 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 Eclipse possui um gerador desses métodos que usa 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 int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((codigo == null) ? 0 : codigo.hashCode());
    return result;
}
```

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
```

```

        return false;
    final Veiculo other = (Veiculo) obj;
    if (codigo == null) {
        if (other.codigo != null)
            return false;
    } else if (!codigo.equals(other.codigo))
        return false;
    return true;
}

```

Agora o Hibernate conseguirá comparar objetos para descobrir se são os mesmos.

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

Entender os motivos por trás da implementação desses métodos é muito importante. Assista uma vídeo aula gratuita sobre esse assunto no nosso blog.

<http://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 informações sobre como a classe `Veiculo` deve se tornar persistente, 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.

```

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

```

```

@Entity
public class Veiculo {

```

```

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

@Id
@GeneratedValue
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;
    }

    // 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, que representa uma tabela do banco de dados.

As anotações nos métodos *getters* configuram a relação dos atributos da classe com as colunas do banco de dados. As anotações `@Id` e `@GeneratedValue` são usadas para declarar o identificador do banco de dados, e esse identificador deve ter um valor gerado no momento de inserção (auto-incremento).

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 os beans de entidade.

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


```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  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_1.xsd">
  <persistence-unit name="AlgaWorksPU">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost/ebook-jpa" />
      <property name="javax.persistence.jdbc.user"
        value="usuario" />
      <property name="javax.persistence.jdbc.password"
        value="senha" />
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver" />

      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5Dialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>

```

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

O provider diz qual é a implementação que será usada como provedor de persistência.

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
- `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)
- `hibernate.hbm2ddl.auto`: cria ou atualiza automaticamente a estrutura das tabelas no banco de dados

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 Hibernate pode fazer isso pra gente, graças à propriedade `hibernate.hbm2ddl.auto` com valor `update`, que incluímos no arquivo `persistence.xml`.

Precisamos apenas criar um `EntityManagerFactory`, que todas as tabelas mapeadas pelas entidades serão criadas ou atualizadas.

```
import javax.persistence.Persistence;

public class CriaTabelas {

    public static void main(String[] args) {
        Persistence.createEntityManagerFactory("AlgaWorksPU");
    }

}
```

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.

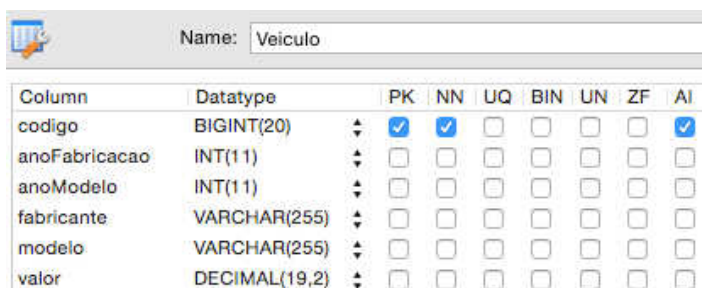
...

```
Jul 11, 2015 1:27:19 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000228: Running hbm2ddl schema update
Jul 11, 2015 1:27:19 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000102: Fetching database metadata
Jul 11, 2015 1:27:19 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000396: Updating schema
Jul 11, 2015 1:27:19 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata [...]
INFO: HHH000262: Table not found: Veiculo
Jul 11, 2015 1:27:19 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata [...]
INFO: HHH000262: Table not found: Veiculo
Jul 11, 2015 1:27:19 AM org.hibernate.tool.hbm2ddl.DatabaseMetadata [...]
INFO: HHH000262: Table not found: Veiculo
Jul 11, 2015 1:27:19 AM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000232: Schema update complete
```

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 com `getters` 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.



The screenshot shows a database table definition window for a table named 'Veiculo'. The table has six columns: 'codigo', 'anoFabricacao', 'anoModelo', 'fabricante', 'modelo', and 'valor'. The 'codigo' column is the primary key (PK) and is not nullable (NN). The other columns are nullable (NN) and have a length of 255 characters for 'fabricante' and 'modelo', and 19,2 for 'valor'. The 'valor' column is a decimal type (DECIMAL(19,2)).

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.Id;
import javax.persistence.Table;

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

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

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

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

    @Column(length = 60, nullable = false)
    public String getFabricante() {
        return fabricante;
    }

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

```

@Column(length = 60, nullable = false)
public String getModelo() {
    return modelo;
}

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

@Column(name = "ano_fabricacao", nullable = false)
public Integer getAnoFabricacao() {
    return anoFabricacao;
}

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

@Column(name = "ano_modelo", nullable = false)
public Integer getAnoModelo() {
    return anoModelo;
}

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

@Column(precision = 10, scale = 2, nullable = true)
public BigDecimal getValor() {
    return valor;
}

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

// equals e hashCode
}

```

Podemos executar o código main anterior, que cria um EntityManagerFactory, e a nova tabela será gerada.

A tabela antiga, chamada `Veiculo`, não será excluída. Você precisará fazer isso “na mão”, se quiser eliminá-la.

Agora, vamos analisar as alterações que fizemos individualmente.

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

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

```
@Column(length = 60, nullable = false)
public String getFabricante() {
```

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

```
@Column(name = "ano_fabricacao", nullable = false)
public Integer getAnoFabricacao() {
```

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(precision = 10, scale = 2, nullable = true)
public BigDecimal getValor() {
```

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

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("AlgaWorksPU");
    }

    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(2012);
        veiculo.setAnoModelo(2013);
        veiculo.setValor(new BigDecimal(71300));

        manager.persist(veiculo);

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

Execute a classe `InsercaoVeiculos` 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.

codigo	ano_fabricacao	ano_modelo	fabricante	modelo	valor
1	2012	2013	Honda	Civic	71300.00

Agora vamos entender o que cada linha significa.

O código abaixo obtém um `EntityManager` da classe `JpaUtil`.

```
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(2012);  
veiculo.setAnoModelo(2013);  
veiculo.setValor(new BigDecimal(71300));
```

Executamos o método `persist`, passando o 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? Nenhum código SQL e JDBC! Trabalhamos apenas com classes, objetos e a API de 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.

```
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();
    }
}
```

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 Civic
```

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 primeiro método *getter* for chamado, 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 Civic

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

2.11. Listando objetos

Agora, você irá aprender como fazer consultas simples de entidades com a linguagem JPQL (*Java Persistence Query Language*). Aprofundaremos um pouco mais nessa linguagem em outro capítulo, porém já usaremos o básico para conseguirmos consultar 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, que foram mapeadas para tabelas.

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:

```
from Veiculo
```

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

Antes de ver como fica no código Java, insira outros veículos na tabela para que os testes fiquem mais interessantes.

```
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.Query;

public class ListandoVeiculos {

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

        Query query = manager.createQuery("from Veiculo");
        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 codigo1_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 - Honda Civic, ano 2012/2013 por R$71300.00
2 - GM Corsa Sedan, ano 2005/2005 por R$16500.00
3 - VW Gol, ano 2009/2009 por R$21000.00

```

A consulta SQL foi gerada baseada 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("from Veiculo");
List<Veiculo> veiculos = query.getResultList();

```

Veja que criamos uma *query* com a JPQL e armazenamos em uma variável *query*. Depois, executamos o método `getResultList` desse objeto e obtemos uma lista de veículos.

As IDEs, como o 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`.

2.12. Atualizando objetos

Os atributos de entidades podem ser manipulados através dos métodos *setters*, 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 *update*.

A saída abaixo foi apresentada na console:

```

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=?

```

```

Valor atual: 71300.00
Novo valor: 71800.00
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.

```

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 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=?
Hibernate:
    delete
    from
        tab_veiculo
    where
        codigo=?

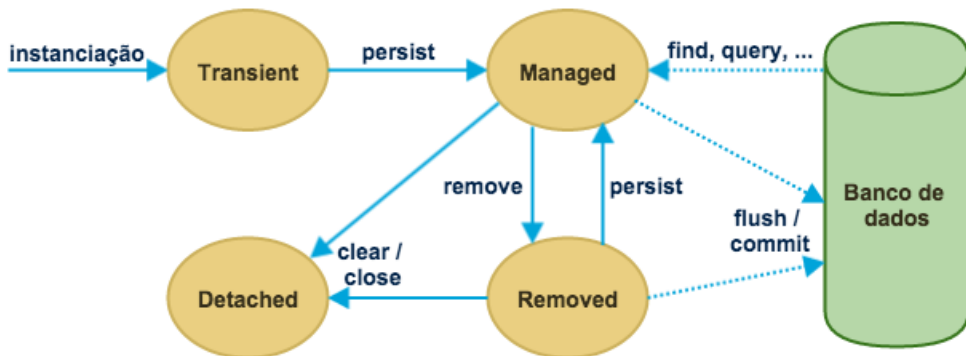
```

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 podem ser perdidos e coletados pelo *garbage collector* quando não estiver mais sendo usado.

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 *Entity Manager* 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 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=?
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` pára 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, 2L);

```

```

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 *update* foi executado apenas no final da execução, exatamente durante o *commit* da transação.

```

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=?
Valor atual: 16500.00
Novo valor: 17000.00
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, 2L);

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 *update* foi executado antes do último *print* 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: 17000.00
Hibernate:
    update
        tab_veiculo
    set
        ano_fabricacao=?,
        ano_modelo=?,
        fabricante=?,
        modelo=?,
        valor=?
    where

```



```
codigo=?  
Novo valor: 17500.00
```

3.4. Salvando objetos desanexados com merge()

Objetos desanexados são objetos em um estado que não é gerenciado pelo `EntityManager`, mas ainda representa uma entidade no banco de dados. As alterações em objetos desanexados não são sincronizadas com o banco de dados.

Quando estamos desenvolvendo sistemas, 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, 2L);
```

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

```
// essa alteração não será sincronizada  
veiculo.setValor(new BigDecimal(5_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, 2L);
```

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

```

manager.close();

veiculo.setValor(new BigDecimal(5_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 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=?
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=?
Hibernate:
    update
        tab_veiculo
    set
        ano_fabricacao=?,
        ano_modelo=?,
        fabricante=?,
        modelo=?,
        valor=?
    where
        codigo=?

```

Mapeamento

4.1. Identificadores

A propriedade de identificação representa a chave primária de uma tabela do banco de dados. Nos exemplos anteriores, a propriedade `codigo` de instâncias de veículos representava o código do veículo (chave primária) no banco de dados.

Um exemplo típico de mapeamento de identificadores é o seguinte:

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

O identificador é mapeado para a chave primária `codigo`. Podemos alterar o nome da coluna usando `@Column`.

```
@Id
@GeneratedValue
@Column(name = "cod_veiculo")
public Long getCodigo() {
    return codigo;
}
```

A anotação `@Id` no método *getter* marca a propriedade como um identificador, e a anotação `@GeneratedValue` diz que o valor do identificador será gerado

automaticamente usando uma estratégia nativa do banco de dados, que no caso do MySQL, é o auto-incremento.

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")
public Long getCodigo() {
    return 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.

Existe um gerador de chave próprio do Hibernate chamado **increment**, que é o famoso *select max + 1*. Para usá-lo, incluímos a anotação `@GenericGenerator` e informamos a propriedade `strategy` igual a `increment`, damos um nome a esse gerador informando o atributo `name` e depois associamos esse gerador na anotação `@GeneratedValue`, informando na propriedade `generator` o mesmo nome escolhido para o gerador.

```
...
import javax.persistence.Id;
import javax.persistence.Column;
import javax.persistence.GeneratedValue;
import org.hibernate.annotations.GenericGenerator;
...

@Id
@GeneratedValue(generator = "inc")
@GenericGenerator(name = "inc", strategy = "increment")
@Column(name = "cod_veiculo")
public Long getCodigo() {
    return codigo;
}
```

Apartir de agora, antes de fazer um *insert* de veículo, o framework executará um comando *select* para buscar o último identificador usado, e então fará um

incremento para utilizar como próximo código. Veja a query gerada quando persistimos um novo veículo:

```
Hibernate:
    select
        max(cod_veiculo)
    from
        tab_veiculo
```

Esses são os identificadores mais usados. Consulte a documentação caso precise de outro tipo de gerador. Se mesmo assim não encontrar um que atenda suas necessidades, você poderá desenvolver o seu próprio gerador customizado.

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.

Criaremos 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 static final long serialVersionUID = 1L;

    private String placa;
    private String cidade;

    public VeiculoId() {
    }

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

```

    }

    public String getPlaca() {
        return placa;
    }

    public void setPlaca(String placa) {
        this.placa = placa;
    }

    public String getCidade() {
        return cidade;
    }

    public void setCidade(String cidade) {
        this.cidade = cidade;
    }

    // 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 seu método *getter* apenas com `@EmbeddedId`.

```

...
import javax.persistence.EmbeddedId;

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

    private VeiculoId id;

    private String fabricante;
    private String modelo;
    private Integer anoFabricacao;
    private Integer anoModelo;
    private BigDecimal valor;
}

```

```

    @EmbeddedId
    public VeiculoId getId() {
        return id;
    }

    // getters, setters, hashCode e equals
}

```

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.

```

Veiculo veiculo = new Veiculo();
veiculo.setId(new VeiculoId("ABC-1234", "Uberlândia"));
veiculo.setFabricante("Honda");
veiculo.setModelo("Civic");
veiculo.setAnoFabricacao(2012);
veiculo.setAnoModelo(2013);
veiculo.setValor(new BigDecimal(71_300));

manager.persist(veiculo);

```

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`.

```

VeiculoId id = new VeiculoId("ABC-1234", "Uberlândia");
Veiculo veiculo = manager.find(Veiculo.class, id);

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

```

Dica: para recriar a tabela, configure no arquivo `persistence.xml` a propriedade `hibernate.hbm2ddl.auto` com valor igual a `create`.

```

<property name="hibernate.hbm2ddl.auto" value="create" />

```

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 {  
  
    ...  
  
    private TipoCombustivel tipoCombustivel;  
  
    ...  
  
    @Column(name = "tipo_combustivel", nullable = false)  
    @Enumerated(EnumType.STRING)  
    public TipoCombustivel getTipoCombustivel() {  
        return tipoCombustivel;  
    }  
  
    public void setTipoCombustivel(TipoCombustivel tipoCombustivel) {  
        this.tipoCombustivel = tipoCombustivel;  
    }  
  
    ...  
}
```

```
}
```

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:

```
Veiculo veiculo = new Veiculo();
veiculo.setFabricante("Ford");
veiculo.setModelo("Focus");
veiculo.setAnoFabricacao(2011);
veiculo.setAnoModelo(2012);
veiculo.setValor(new BigDecimal(41_500));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);

manager.persist(veiculo);
```

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)
public TipoCombustivel getTipoCombustivel() {
    return tipoCombustivel;
}
```

4.4. Propriedades temporais

O tipo de uma propriedade é automaticamente detectado, mas para uma propriedade do tipo `Date` ou `Calendar`, você pode precisar definir a precisão da data/hora, com a anotação `@Temporal`.

```
@Temporal(TemporalType.DATE)
@Column(name = "data_cadastro", nullable = false)
public Date getDataCadastro() {
```

```

        return dataCadastro;
    }

```

O mapeamento acima, do atributo `dataCadastro`, diz que apenas a data é importante, por isso, ao criarmos/atualizarmos a tabela usando o *hbm2ddl*, uma coluna do tipo `DATE` será criada, se usarmos MySQL.

```

Veiculo veiculo = new Veiculo();
veiculo.setFabricante("GM");
veiculo.setModelo("Celta");
veiculo.setAnoFabricacao(2008);
veiculo.setAnoModelo(2008);
veiculo.setValor(new BigDecimal(12_000));
veiculo.setTipoCombustivel(TipoCombustivel.GASOLINA);
veiculo.setDataCadastro(new Date());

manager.persist(veiculo);

```

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`.

4.5. Propriedades transientes

As propriedades de uma entidade são automaticamente mapeadas, se não especificarmos nenhuma anotação. Por diversas vezes, podemos precisar criar atributos e/ou métodos *getters* que não representam uma coluna no banco de dados. Nestes casos, devemos anotar com `@Transient`.

```

@Transient
public String getDescricao() {
    return this.getFabricante() + " " + this.getModelo()
        + " " + this.getAnoFabricacao() + "/" + this.getAnoModelo()
        + " por apenas " + this.getValor();
}

```

A propriedade será ignorada totalmente pelo mecanismo de persistência. O método `getDescricao` é apenas um método de negócio, que retorna uma descrição comercial do veículo.

```
Veiculo veiculo = manager.find(Veiculo.class, 9L);
System.out.println(veiculo.getDescricao());
```

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 {

    ...

    private String especificacoes;

    ...

    @Lob
    public String getEspecificacoes() {
        return especificacoes;
    }

    ...
}
```

Vamos persistir um novo veículo, incluindo uma especificação.

```

EntityManager manager = JpaUtil.getEntityManager();
EntityManager 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(2010);
veiculo.setAnoModelo(2010);
veiculo.setValor(new BigDecimal(17_200));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(new Date());
veiculo.setEspecificacoes(especificacoes.toString());

manager.persist(veiculo);

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

```

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

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

especificacoes	LONGTEXT
----------------	----------

Podemos buscar um veículo normalmente e imprimir na saída as especificações.

```

EntityManager manager = JpaUtil.getEntityManager();

Veiculo veiculo = manager.find(Veiculo.class, 15L);
System.out.println("Veículo: " + veiculo.getModelo());
System.out.println("-----");
System.out.println(veiculo.getEspecificacoes());

```

```
manager.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 {

    ...

    private byte[] foto;

    ...

    @Lob
    public byte[] getFoto() {
        return foto;
    }

    ...
}
```

Vamos persistir um novo veículo, incluindo uma foto.

```
// lê bytes do arquivo da imagem
Path path = FileSystems.getDefault().getPath("/caminho/da/imagem/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(2013);
veiculo.setAnoModelo(2014);
veiculo.setValor(new BigDecimal(100_000));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(new Date());
veiculo.setFoto(foto);

manager.persist(veiculo);

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

```

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

Podemos buscar um veículo normalmente, obter os bytes da imagem e exibir 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 {
        EntityManager manager = JpaUtil.getEntityManager();
        Veiculo veiculo = manager.find(Veiculo.class, 16L);

        if (veiculo.getFoto() != null) {
            BufferedImage img = ImageIO.read(new ByteArrayInputStream(

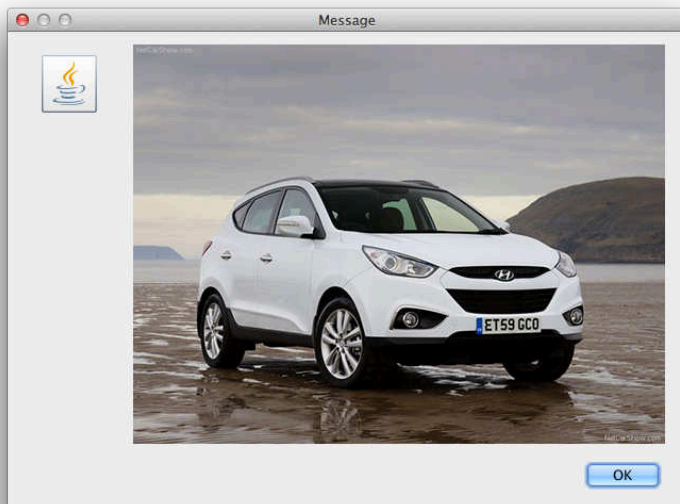
```

```

        veiculo.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, queremos 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 sistemas legados, onde não é permitido alterar a estrutura das tabelas.

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

`@Embeddable`

```
public class Proprietario {

    private String nome;
    private String telefone;
    private String email;

    @Column(name = "nome_proprietario", nullable = false)
    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Column(name = "telefone_proprietario", nullable = false)
    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    @Column(name = "email_proprietario")
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

```
}
```

Como as colunas serão criadas na tabela de veículos, mapeamos as propriedades 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` com o *getter* anotado com `@Embedded`.

```
public class Veiculo {  
  
    ...  
  
    private Proprietario proprietario;  
  
    ...  
  
    @Embedded  
    public Proprietario getProprietario() {  
        return proprietario;  
    }  
  
    ...  
  
}
```

Agora, para persistir um novo veículo, precisamos instanciar e relacionar com um `Proprietario`.

```
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(2010);
```

```

veiculo.setAnoModelo(2010);
veiculo.setValor(new BigDecimal(17_200));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(new Date());
veiculo.setProprietario(proprietario);

manager.persist(veiculo);

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

```

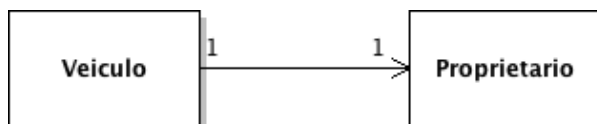
Veja as novas colunas que foram criadas na tabela.

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 fácil de entender.

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 {

    private Long codigo;
    private String nome;
    private String telefone;
    private String email;

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

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

    @Column(length = 60, nullable = false)
    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Column(length = 20, nullable = false)
    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    @Column(length = 255)
    public String getEmail() {
        return email;
    }
}
```

```

    }

    public void setEmail(String email) {
        this.email = email;
    }

    // hashCode e equals

}

```

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

```

public class Veiculo {

    ...

    private Proprietario proprietario;

    ...

    @OneToOne
    public Proprietario getProprietario() {
        return proprietario;
    }

    ...

}

```

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(2010);
veiculo.setAnoModelo(2010);
veiculo.setValor(new BigDecimal(17_200));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(new Date());
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: Veiculo.proprietario -> 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");

manager.persist(proprietario);

Veiculo veiculo = new Veiculo();
veiculo.setFabricante("VW");
veiculo.setModelo("Gol");
veiculo.setAnoFabricacao(2010);
veiculo.setAnoModelo(2010);
veiculo.setValor(new BigDecimal(17_200));
veiculo.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo.setDataCadastro(new Date());
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")
public Proprietario getProprietario() {
    return 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)
public Proprietario getProprietario() {
    return 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("from Veiculo", 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
    proprietario0_.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 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_,
    proprietario1_.codigo as codigol_0_0_,
    proprietario1_.email as email2_0_0_,
    proprietario1_.nome as nome3_0_0_,
    proprietario1_.telefone as telefone4_0_0_
from
    tab_veiculo veiculo0_
inner join
    proprietario proprietario1_
    on veiculo0_.proprietario_codigo=proprietar1_.codigo
```

```
where
    veiculo0_.codigo=?
```

Associação bidirecional

A associação que fizemos entre veículo e proprietário 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 mapearmos com @OneToOne usando o atributo mappedBy.

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

    ...

    private Veiculo veiculo;

    ...

    @OneToOne(mappedBy = "proprietario")
    public Veiculo getVeiculo() {
        return veiculo;
    }

    ...

}
```

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.

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

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

4.9. Associações muitos-para-um

Na última seção, mapeamos a propriedade `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 a propriedade `veiculo` da entidade `Proprietario` e alterar o mapeamento de `proprietario` na classe `Veiculo`.

```
public class Veiculo {

    ...

    private Proprietario proprietario;

    ...

    @ManyToOne
    @JoinColumn(name = "cod_proprietario")
    public Proprietario getProprietario() {
        return proprietario;
    }

    ...

}
```

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(2006);
veiculo1.setAnoModelo(2006);
veiculo1.setValor(new BigDecimal(11_000));
veiculo1.setTipoCombustivel(TipoCombustivel.GASOLINA);
veiculo1.setDataCadastro(new Date());
veiculo1.setProprietario(proprietario);

manager.persist(veiculo1);

Veiculo veiculo2 = new Veiculo();
veiculo2.setFabricante("VW");
veiculo2.setModelo("Gol");
veiculo2.setAnoFabricacao(2010);
veiculo2.setAnoModelo(2010);
veiculo2.setValor(new BigDecimal(17_200));
veiculo2.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo2.setDataCadastro(new Date());
veiculo2.setProprietario(proprietario);

manager.persist(veiculo2);

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

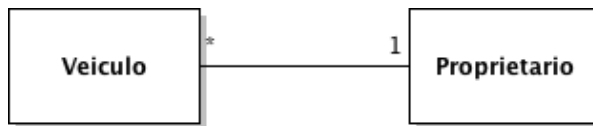
```

Para deixar a associação bidirecional, precisamos mapear um atributo na entidade Proprietario, usando a @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 a propriedade `veiculos` na entidade `Proprietario`, do tipo `List`.

```
public class Proprietario {  
  
    ...  
  
    private List<Veiculo> veiculos;  
  
    ...  
  
    @OneToMany(mappedBy = "proprietario")  
    public List<Veiculo> getVeiculos() {  
        return veiculos;  
    }  
  
    ...  
  
}
```

Para testar esse relacionamento, 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: João das Couves

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: Celta

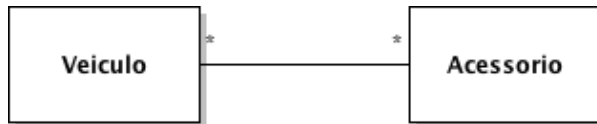
Veículo: Gol2

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 estar associado a vários veículos.



A classe `Acessorio` é uma entidade sem nenhuma novidade nos mapeamentos.

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

    private Long codigo;
    private String descricao;

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

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

    @Column(length = 60, nullable = false)
    public String getDescricao() {
        return descricao;
    }
}
```

```

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

    // hashCode e equals

}

```

Na entidade `Veiculo`, criamos um atributo `acessorios` do tipo `Set`. Definimos como um conjunto, pois um veículo não poderá possuir o mesmo acessório repetido. Usamos a anotação `@ManyToMany` para mapear a propriedade de coleção.

```

public class Veiculo {

    ...

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

    ...

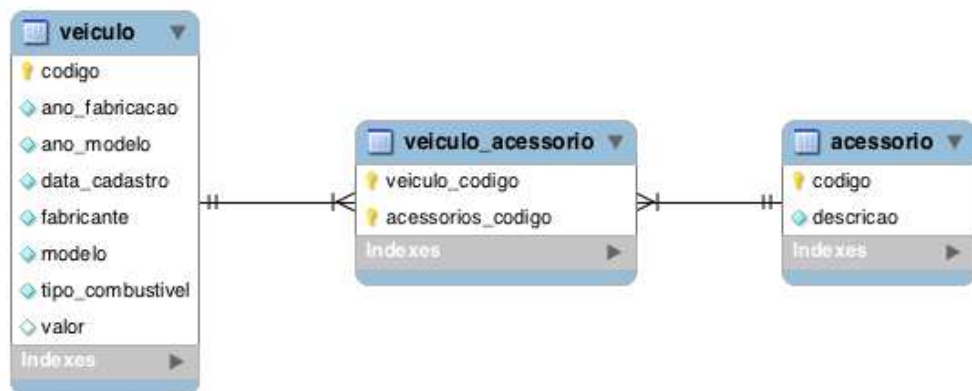
    @ManyToMany
    public Set<Acessorio> getAcessorios() {
        return acessorios;
    }

    ...

}

```

Esse tipo de relacionamento precisará de uma tabela de associação para que a multiplicidade muitos-para-muitos funcione. O recurso `hbm2ddl` poderá criar as tabelas automaticamente.



Por padrão, um mapeamento com `@ManyToMany` cria a tabela de associação com os nomes das entidades relacionadas, separados por *underscore*, com duas 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_acessorios",
    joinColumns = @JoinColumn(name = "cod_veiculo"),
    inverseJoinColumns = @JoinColumn(name = "cod_acessorio"))
public Set<Acessorio> getAcessorios() {
    return acessorios;
}
```

Neste exemplo, definimos o nome da tabela de associação como `veiculo_acessorios`, o nome da coluna que faz referência para a tabela de veículos como `cod_veiculo` e da coluna que referencia à tabela de acessórios como `cod_acessorio` (lado inverso).



Vamos inserir e relacionar alguns acessórios e veículos.

```

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

// instancia acessórios
Acessorio alarme = new Acessorio();
alarme.setDescricao("Alarme");

Acessorio arCondicionado = new Acessorio();
arCondicionado.setDescricao("Ar condicionado");

Acessorio bancosDeCouro = new Acessorio();
bancosDeCouro.setDescricao("Bancos de couro");

Acessorio direcaoHidraulica = new Acessorio();
direcaoHidraulica.setDescricao("Direção hidráulica");

// persiste acessórios
manager.persist(alarme);
manager.persist(arCondicionado);
manager.persist(bancosDeCouro);
manager.persist(direcaoHidraulica);

// instancia veículos
Veiculo veiculo1 = new Veiculo();
veiculo1.setFabricante("VW");
veiculo1.setModelo("Gol");
veiculo1.setAnoFabricacao(2010);
veiculo1.setAnoModelo(2010);
veiculo1.setValor(new BigDecimal(17_200));
veiculo1.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo1.setDataCadastro(new Date());
veiculo1.getAcessorios().add(alarme);
veiculo1.getAcessorios().add(arCondicionado);

Veiculo veiculo2 = new Veiculo();
veiculo2.setFabricante("Hyundai");
veiculo2.setModelo("i30");
veiculo2.setAnoFabricacao(2012);
veiculo2.setAnoModelo(2012);
veiculo2.setValor(new BigDecimal(53_500));
veiculo2.setTipoCombustivel(TipoCombustivel.BICOMBUSTIVEL);
veiculo2.setDataCadastro(new Date());
veiculo2.getAcessorios().add(alarme);

```

```

veiculo2.getAcessorios().add(arCondicionado);
veiculo2.getAcessorios().add(bancosDeCouro);
veiculo2.getAcessorios().add(direcaoHidraulica);
veiculo2.getAcessorios().add(direcaoHidraulica);

// 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`.

```

EntityManager manager = JpaUtil.getEntityManager();

Veiculo veiculo = manager.find(Veiculo.class, 2L);
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 codigol_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: i30
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 {

    ...

    private Set<Veiculo> veiculos = new HashSet<>();

    ...

    @ManyToMany(mappedBy = "acessorios")
    public Set<Veiculo> getVeiculos() {

```

```

        return veiculos;
    }

    ...

}

```

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`.

```

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

    private Long codigo;
    private String nome;
    private List<String> telefones = new ArrayList<>();

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

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

    @Column(name = "nome", length = 60, nullable = false)
    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {

```

```

        this.nome = nome;
    }

    @ElementCollection
    @CollectionTable(name = "proprietario_telefones",
        joinColumns = @JoinColumn(name = "cod_proprietario"))
    @Column(name = "numero_telefone", length = 20, nullable = false)
    public List<String> getTelefones() {
        return telefones;
    }

    public void setTelefones(List<String> telefones) {
        this.telefones = telefones;
    }

    // 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 o atributo `joinColumns`.

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

Agora, podemos persistir um proprietário com diversos telefones.

```

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.

```
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 {

    private String prefixo;
    private String numero;
    private String ramal;

    public Telefone() {
    }

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

    @Column(length = 3, nullable = false)
    public String getPrefixo() {
        return prefixo;
    }
}
```

```

public void setPrefixo(String prefixo) {
    this.prefixo = prefixo;
}

@Column(length = 20, nullable = false)
public String getNumero() {
    return numero;
}

public void setNumero(String numero) {
    this.numero = numero;
}

@Column(length = 5)
public String getRamal() {
    return ramal;
}

public void setRamal(String ramal) {
    this.ramal = ramal;
}
}

```

Na entidade Proprietario, criamos um atributo do tipo List e mapeamos o *getter* com @ElementCollection.

```

public class Proprietario {

    ...

    private List<Telefone> telefones = new ArrayList<>();

    ...

    @ElementCollection
    @CollectionTable(name = "proprietario_telefones",
        joinColumns = @JoinColumn(name = "cod_proprietario"))
    @AttributeOverrides({ @AttributeOverride(name = "numero",
        column = @Column(name = "num_telefone", length = 20, nullable = false)) })
    public List<Telefone> getTelefones() {
        return telefones;
    }
}

```



```

    }

    ...

}

```

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

Agora, podemos persistir um proprietário com telefones.

```

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* das propriedades.

```

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 {

    private Long id;
    private String nome;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

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

```

}

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

public void setNome(String nome) {
    this.nome = nome;
}

// 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 {

    private BigDecimal limiteCredito;
    private BigDecimal rendaMensal;
    private boolean bloqueado;

    @Column(name = "limite_credito", nullable = true)
    public BigDecimal getLimiteCredito() {
        return limiteCredito;
    }

    public void setLimiteCredito(BigDecimal limiteCredito) {
        this.limiteCredito = limiteCredito;
    }

    @Column(name = "renda_mensal", nullable = true)
    public BigDecimal getRendaMensal() {

```

```

        return rendaMensal;
    }

    public void setRendaMensal(BigDecimal rendaMensal) {
        this.rendaMensal = rendaMensal;
    }

    @Column(nullable = true)
    public boolean isBloqueado() {
        return bloqueado;
    }

    public void setBloqueado(boolean bloqueado) {
        this.bloqueado = bloqueado;
    }
}

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

    private BigDecimal salario;
    private String cargo;

    @Column(nullable = true)
    public BigDecimal getSalario() {
        return salario;
    }

    public void setSalario(BigDecimal salario) {
        this.salario = salario;
    }

    @Column(length = 60, nullable = true)
    public String getCargo() {
        return cargo;
    }

    public void setCargo(String cargo) {
        this.cargo = cargo;
    }
}

```

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.

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

```
List<Cliente> clientes = manager.createQuery("from Cliente",
    Cliente.class).getResultList();

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

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

```
Hibernate:
select
    cliente0_.id as id2_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'
```

É possível também fazer uma consulta polimórfica de pessoas.

```
List<Pessoa> pessoas = manager.createQuery("from Pessoa",
    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_.id as id2_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 tipo1_0_
from
    pessoa pessoa0_
```

Uma tabela para cada classe da hierarquia

Outra forma de fazer 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 {

    ...

}

```

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_id")
public class Funcionario extends Pessoa {

    ...

}

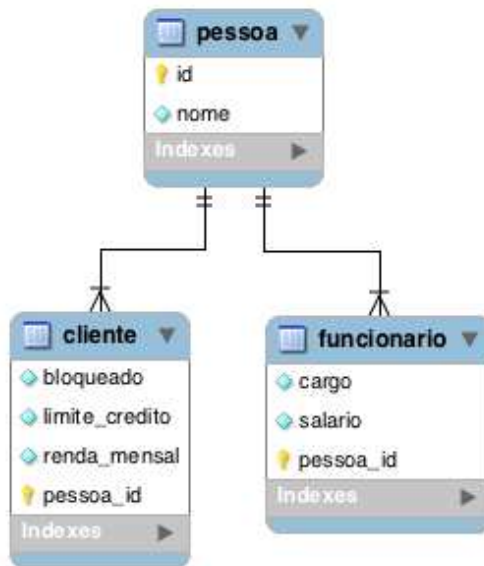
@Entity
@Table(name = "cliente")
@PrimaryKeyJoinColumn(name = "pessoa_id")
public class Cliente extends Pessoa {

    ...

}

```

Este tipo de mapeamento criará 3 tabelas.



Podemos persistir funcionários e clientes e depois executar uma pesquisa polimórfica, para listar todas as pessoas. Veja a consulta SQL usada:

Hibernate:

```

select
    pessoa0_.id as id1_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 cargol_1_,
    pessoa0_2_.salario as salario2_1_,
    case
        when pessoa0_1_.pessoa_id is not null then 1
        when pessoa0_2_.pessoa_id is not null then 2
        when pessoa0_.id is not null then 0
    end as clazz_
from
    pessoa pessoa0_
left outer join
    cliente pessoa0_1_
        on pessoa0_.id=pessoa0_1_.pessoa_id
left outer join
    funcionario pessoa0_2_
        on pessoa0_.id=pessoa0_2_.pessoa_id

```

```
funcionario pessoa0_2_  
    on pessoa0_.id=pessoa0_2_.pessoa_id
```

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 {  
  
    private Long id;  
    private String nome;  
  
    @Id  
    @GeneratedValue(generator = "inc")  
    @GenericGenerator(name = "inc", strategy = "increment")  
    public Long getId() {  
        return id;  
    }  
  
    ...  
}
```

Tivemos que mudar a estratégia de geração de identificadores. Não podemos usar a geração automática de chaves nativa do banco de dados.

Veja a estrutura das tabelas criadas:



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

Hibernate:

```
select
    pessoa0_.id as id1_0_,
    pessoa0_.nome as nome2_0_,
    pessoa0_.bloqueado as bloquead1_1_,
    pessoa0_.limite_credito as limite_c2_1_,
    pessoa0_.renda_mensal as renda_me3_1_,
    pessoa0_.cargo as cargo1_2_,
    pessoa0_.salario as salario2_2_,
    pessoa0_.clazz_ as clazz_
from
    ( select
        id,
        nome,
        bloqueado,
        limite_credito,
        renda_mensal,
        null as cargo,
        null as salario,
        1 as clazz_
    from
        cliente
    union
    select
        id,
        nome,
        null as bloqueado,
        null as limite_credito,
        null as renda_mensal,
        cargo,
        salario,
        2 as clazz_
    from
        funcionario
    )
```

```

        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 {

    private Long id;
    private String nome;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

    ....

}

```

As subclasses são mapeadas normalmente, sem nada especial:

```

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

    ...

}

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

```

```
...  
}
```

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 [from Pessoa]
```

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. Os métodos *getters* e *setters* não são obrigatórios, neste caso, mas caso eles existam, serão ignorado 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 id;

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

@Temporal(TemporalType.TIMESTAMP)
@Column(nullable = false)
private Date dataLimite;

public Long getId() {
    return id;
}

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

public String getDescricao() {
    return descricao;
}

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

public Date getDataLimite() {
    return dataLimite;
}

public void setDataLimite(Date 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*, como fizemos nos exemplos das seções anteriores. 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 id;
    private String descricao;
    private Date dataLimite;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

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

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

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

    @Temporal(TemporalType.TIMESTAMP)
    @Column(nullable = false)
    public Date getDataLimite() {
        return dataLimite;
    }
}
```

```
public void setDataLimite(Date 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 {

    private Long id;
    private String nome;
    private Categoria categoria;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

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

    @Column(length = 60, nullable = false)
```

```

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @ManyToOne(optional = false)
    public Categoria getCategoria() {
        return categoria;
    }

    public void setCategoria(Categoria categoria) {
        this.categoria = categoria;
    }

    // hashCode e equals
}

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

    private Long id;
    private String nome;
    private List<Produto> produtos = new ArrayList<>();

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

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

    @Column(length = 60, nullable = false)
    public String getNome() {
        return nome;
    }
}

```

```

    public void setNome(String nome) {
        this.nome = nome;
    }

    @OneToMany(mappedBy = "categoria")
    public List<Produto> getProdutos() {
        return produtos;
    }

    public void setProdutos(List<Produto> produtos) {
        this.produtos = produtos;
    }

    // hashCode e equals
}

```

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

Vamos inserir uma categoria e alguns produtos, para podermos testar.

```

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

```

```

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

```

```

Produto produto1 = new Produto();
produto1.setNome("Refrigerante");
produto1.setCategoria(categoria);

```

```

Produto produto2 = new Produto();
produto2.setNome("Água");
produto2.setCategoria(categoria);

```

```

Produto produto3 = new Produto();
produto3.setNome("Cerveja");
produto3.setCategoria(categoria);

```

```

manager.persist(categoria);
manager.persist(produto1);

```

```

manager.persist(produto2);
manager.persist(produto3);

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

```

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_.id as id1_1_1_,
        produto0_.categoria_id as categori3_1_1_,
        produto0_.nome as nome2_1_1_,
        categorial_.id as id1_0_0_,
        categorial_.nome as nome2_0_0_
    from
        produto produto0_
    inner join
        categoria categorial_
        on produto0_.categoria_id=categorial_.id
    where
        produto0_.id=?
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)
public Categoria getCategoria() {
    return categoria;
}
```

Quando executamos a consulta anterior novamente, veja a saída:

```
Hibernate:
  select
    produto0_.id as id1_1_0_,
    produto0_.categoria_id as categori3_1_0_,
    produto0_.nome as nome2_1_0_
  from
    produto produto0_
  where
    produto0_.id=?
Nome: Cerveja
```

O provedor JPA não buscou as informações de categoria, pois o carregamento passou a ser tardia, 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_.id as id1_1_0_,
        produto0_.categoria_id as categori3_1_0_,
        produto0_.nome as nome2_1_0_
    from
        produto produto0_
    where
        produto0_.id=?

```

Nome: Cerveja

```

Hibernate:
    select
        categoria0_.id as id1_0_0_,
        categoria0_.nome as nome2_0_0_
    from
        categoria categoria0_
    where
        categoria0_.id=?

```

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 - no Session

Lazy loading em mapeamento OneToOne

Antes de configurar o mapeamento one-to-one para usar lazy loading, você precisa assistir essa vídeo aula gratuita em nosso blog.

<http://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*, na 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_.id as id1_0_0_,
        categoria0_.nome as nome2_0_0_
    from
        categoria categoria0_
    where
        categoria0_.id=?
Categoria: Bebidas
Hibernate:
    select
        produtos0_.categoria_id as categori3_0_1_,
        produtos0_.id as id1_1_1_,
        produtos0_.id as id1_1_0_,
        produtos0_.categoria_id as categori3_1_0_,
        produtos0_.nome as nome2_1_0_
    from
        produto produtos0_
    where
        produtos0_.categoria_id=?
Produto: Refrigerante
Produto: Água
Produto: Cerveja

```

Embora não recomendado na maioria dos casos, podemos mudar a estratégia de carregamento de produtos na entidade `Categoria` para *eager*.

```

@OneToMany(mappedBy = "categoria", fetch = FetchType.EAGER)
public List<Produto> getProdutos() {
    return 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_.id as id1_0_1_,
    categoria0_.nome as nome2_0_1_,
    produtos1_.categoria_id as categori3_0_3_,
    produtos1_.id as id1_1_3_,
    produtos1_.id as id1_1_0_,
    produtos1_.categoria_id as categori3_1_0_,
    produtos1_.nome as nome2_1_0_
from
    categoria categoria0_
left outer join
    produto produtos1_
        on categoria0_.id=produtos1_.categoria_id
where
    categoria0_.id=?
```

Categoria: Bebidas

Produto: Refrigerante

Produto: Água

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("Bebidas");

Produto produto = new Produto();
produto.setNome("Refrigerante");
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: Produto.categoria -> 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.

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

manager.persist(categoria);

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

manager.persist(produto);
```

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*, e para configurá-lo, basta adicionarmos

um atributo cascade na anotação de relacionamento e adicionar a operação CascadeType.PERSIST.

```
@ManyToOne(optional = false, cascade = CascadeType.PERSIST)
public Categoria getCategory() {
    return categoria;
}
```

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

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

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

manager.persist(produto);
```

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.

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

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

categoria.getProdutos().add(produto);

manager.persist(categoria);
```

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

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

```
@OneToMany(mappedBy = "categoria", cascade = CascadeType.PERSIST)
public List<Produto> getProdutos() {
    return 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.

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

O provedor JPA tentará remover apenas a categoria, mas isso não será possível, pois o banco de dados possui integridade.

```
Caused by: com.mysql.jdbc.exceptions.jdbc4.
MySQLIntegrityConstraintViolationException:
Cannot delete or update a parent row: a foreign key constraint fails
(`ebookjpa`.`produto`, CONSTRAINT `FK_el0d58htywfs914w4grf6aoa0`
FOREIGN KEY (`categoria_id`) REFERENCES `categoria` (`id`))
```

Vamos configurar a operação de exclusão em cascata no relacionamento `produtos` da entidade `Categoria`. Para isso, basta adicionar a constante `CascadeType.REMOVE` no atributo `cascade` do mapeamento.

```
@OneToMany(mappedBy = "categoria", cascade = { CascadeType.PERSIST,
    CascadeType.REMOVE })
public List<Produto> getProdutos() {
    return produtos;
}
```

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)
public Categoria getCategoria() {
    return 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 o atributo `orphanRemoval` no mapeamento.

```

@OneToMany(mappedBy = "categoria", cascade = CascadeType.PERSIST,
    orphanRemoval = true)
public List<Produto> getProdutos() {
    return 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 {

    private Long id;
    private String email;
    private boolean ativo;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

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

    @Column(length = 255, nullable = false)
    public String getEmail() {
        return email;
    }
}

```

```

    public void setEmail(String email) {
        this.email = email;
    }

    public boolean isAtivo() {
        return ativo;
    }

    public void setAtivo(boolean ativo) {
        this.ativo = ativo;
    }

    // hashCode e equals
}

```

Inserimos alguns usuários para nosso teste:

```

Usuario u1 = new Usuario();
u1.setEmail("joao@algaworks.com");
u1.setAtivo(true);

Usuario u2 = new Usuario();
u2.setEmail("manoel@algaworks.com");
u2.setAtivo(true);

Usuario u3 = new Usuario();
u3.setEmail("sebastiao123@gmail.com");
u3.setAtivo(true);

manager.persist(u1);
manager.persist(u2);
manager.persist(u3);

```

Queremos inativar todos os usuários que possuem e-mails do domínio “@algaworks.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", "%@algaworks.com");

int linhasAfetadas = query.executeUpdate();

System.out.println(linhasAfetadas + " registros atualizados.");

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 “@algaworks.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 à primeira transação
Usuario u1 = manager1.find(Usuario.class, 1L);
u1.setEmail("maria@algaworks.com");

// altera objeto associado à segunda transação
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_.id as id1_0_0_,
    usuario0_.ativo as ativo2_0_0_,
    usuario0_.email as email3_0_0_
  from
    usuario usuario0_
  where
    usuario0_.id=?

```

```

Hibernate:
  select
    usuario0_.id as id1_0_0_,
    usuario0_.ativo as ativo2_0_0_,
    usuario0_.email as email3_0_0_
  from
    usuario usuario0_
  where
    usuario0_.id=?

```

```

Hibernate:
  update
    usuario
  set
    ativo=?,
    email=?
  where
    id=?

```

```

Hibernate:
  update
    usuario
  set
    ativo=?,
    email=?
  where
    id=?

```

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 transações?

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 simples da entidade. Para isso funcionar, precisamos mapear uma propriedade para armazenar a versão da entidade, usando a anotação `@Version`.

```
public class Usuario {  
  
    ...  
  
    private Long versao;  
  
    ...  
  
    @Version  
    public Long getVersao() {  
        return versao;  
    }  
  
    ...  
  
}
```

Agora, se houver concorrência na alteração do mesmo registro, uma exceção será lançada:

```
Caused by: javax.persistence.OptimisticLockException:
org.hibernate.StaleObjectStateException: Row was updated or deleted
by another transaction (or unsaved-value mapping was incorrect):
[Usuario#1]
```

Vídeo aula sobre Lock Otimista

Para não ficar qualquer dúvida, assista a vídeo aula gratuita no blog da AlgaWorks que explica, através de um projeto web, como usar locking otimista.

<http://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 sistemas, 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`.

```
Usuario usuario = manager1.find(Usuario.class, 1L,
    LockModeType.PESSIMISTIC_WRITE);
```

O lock é feito adicionando `for update` na consulta SQL executada, veja:

```
Hibernate:
select
    usuario0_.id as id1_0_0_,
```

```

        usuario0_.ativo as ativo2_0_0_,
        usuario0_.email as email3_0_0_,
        usuario0_.versao as versao4_0_0_
from
    usuario usuario0_
where
    usuario0_.id=? for update

```

5.6. Métodos de callback e auditores de entidades

Em algumas situações específicas, podemos precisar 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 {

    private Long id;
    private String nome;
    private Date dataNascimento;
    private Date dataUltimaAtualizacao;
    private Integer idade;

```

```

@Id
@GeneratedValue
public Long getId() {
    return id;
}

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

@Column(length = 60, nullable = false)
public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

@Temporal(TemporalType.DATE)
@Column(name = "data_nascimento", nullable = false)
public Date getDataNascimento() {
    return dataNascimento;
}

public void setDataNascimento(Date dataNascimento) {
    this.dataNascimento = dataNascimento;
}

@Temporal(TemporalType.TIMESTAMP)
@Column(name = "data_ultima_atualizacao", nullable = false)
public Date getDataUltimaAtualizacao() {
    return dataUltimaAtualizacao;
}

public void setDataUltimaAtualizacao(Date dataUltimaAtualizacao) {
    this.dataUltimaAtualizacao = dataUltimaAtualizacao;
}

@Transient
public Integer getIdade() {
    return idade;
}

```

```

    public void setIdade(Integer idade) {
        this.idade = idade;
    }

    @PostLoad
    @PostPersist
    @PostUpdate
    public void calcularIdade() {
        long idadeEmMillis = new Date().getTime()
            - this.getDataNascimento().getTime();
        this.setIdade((int) (idadeEmMillis / 1000d / 60 / 60 / 24 / 365));
    }

    // 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();

Calendar dataNascimento = Calendar.getInstance();
dataNascimento.set(2011, 2, 1);

Animal animal = new Animal();
animal.setNome("Mimosa");
animal.setDataNascimento(dataNascimento.getTime());
animal.setDataUltimaAtualizacao(new Date());

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(new Date());
    }

}
```


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 {

    ...

}
```

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.

```
Calendar dataNascimento = Calendar.getInstance();
dataNascimento.set(2009, 8, 20);

Animal animal = new Animal();
animal.setNome("Campeão");
animal.setDataNascimento(dataNascimento.getTime());

manager.persist(animal);
```

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 `EntityManager`s 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 version="2.1"
    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_1.xsd">
```

```

<persistence-unit name="ErpPU">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>

  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>

  <properties>
    <property name="javax.persistence.jdbc.url"
      value="jdbc:mysql://localhost/erp" />
    <property name="javax.persistence.jdbc.user"
      value="usuario" />
    <property name="javax.persistence.jdbc.password"
      value="senha" />
    <property name="javax.persistence.jdbc.driver"
      value="com.mysql.jdbc.Driver" />

    <property name="hibernate.dialect"
      value="org.hibernate.dialect.MySQL5Dialect" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.format_sql" value="true" />
    <property name="hibernate.hbm2ddl.auto" value="update" />

    <property name="hibernate.cache.region.factory_class"
      value="org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory" />
  </properties>
</persistence-unit>

</persistence>

```

Usaremos o EhCache, que já vem na distribuição do Hibernate. Definimos a implementação do cache no elemento `hibernate.cache.region.factory_class` do arquivo *persistence.xml*. Adicione os arquivos JAR da pasta *lib/optional/ehcache* no *classpath* do projeto.

O elemento `shared-cache-mode` 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 {

    private static final long serialVersionUID = 1L;

    private Long id;
    private String nome;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

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

    @Column(length = 60, nullable = false)
    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    // hashCode e equals
}
```

Para testar, iremos inserir alguns centros de custo.

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

CentroCusto cc1 = new CentroCusto();
cc1.setNome("Tecnologia");

CentroCusto cc2 = new CentroCusto();
cc2.setNome("Comercial");

manager.persist(cc1);
manager.persist(cc2);

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

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_.id as id1_0_0_,
        centrocust0_.nome as nome2_0_0_
    from
        centro_custo centrocust0_
```

```
where
    centrocust0_.id=?
Centro de custo: Tecnologia
-----
Centro de custo: Tecnologia
```

Vídeo aula sobre cache de segundo nível

No blog da AlgaWorks tem uma vídeo aula gratuita que explica um pouco mais sobre cache de segundo nível, demonstrado em um projeto web com JSF e CDI.

<http://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 banco de dados.

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 mapeadas a seguir.

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

    private Long codigo;
    private String fabricante;
    private String modelo;
```

```

private Integer anoFabricacao;
private Integer anoModelo;
private BigDecimal valor;
private TipoCombustivel tipoCombustivel;
private Date dataCadastro;
private Proprietario proprietario;
private Set<Acessorio> acessorios = new HashSet<>();

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

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

@Column(length = 60, nullable = false)
public String getFabricante() {
    return fabricante;
}

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

@Column(length = 60, nullable = false)
public String getModelo() {
    return modelo;
}

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

@Column(name = "ano_fabricacao", nullable = false)
public Integer getAnoFabricacao() {
    return anoFabricacao;
}

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

```

```

@Column(name = "ano_modelo", nullable = false)
public Integer getAnoModelo() {
    return anoModelo;
}

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

@Column(precision = 10, scale = 2, nullable = true)
public BigDecimal getValor() {
    return valor;
}

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

@Column(name = "tipo_combustivel", nullable = false)
@Enumerated(EnumType.STRING)
public TipoCombustivel getTipoCombustivel() {
    return tipoCombustivel;
}

public void setTipoCombustivel(TipoCombustivel tipoCombustivel) {
    this.tipoCombustivel = tipoCombustivel;
}

@Temporal(TemporalType.DATE)
@Column(name = "data_cadastro", nullable = false)
public Date getDataCadastro() {
    return dataCadastro;
}

public void setDataCadastro(Date dataCadastro) {
    this.dataCadastro = dataCadastro;
}

@ManyToOne
@JoinColumn(name = "cod_proprietario")
public Proprietario getProprietario() {
    return proprietario;
}

```



```

    public void setProprietario(Proprietario proprietario) {
        this.proprietario = proprietario;
    }

    @ManyToMany
    @JoinTable(name = "veiculo_acessorios",
        joinColumns = @JoinColumn(name = "cod_veiculo"),
        inverseJoinColumns = @JoinColumn(name = "cod_acessorio"))
    public Set<Acessorio> getAcessorios() {
        return acessorios;
    }

    public void setAcessorios(Set<Acessorio> acessorios) {
        this.acessorios = acessorios;
    }

    // hashCode e equals

}

public enum TipoCombustivel {

    ALCOOL, GASOLINA, DIESEL, BICOMBUSTIVEL

}

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

    private Long codigo;
    private String nome;
    private String telefone;
    private String email;
    private List<Veiculo> veiculos;

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

    public void setCodigo(Long codigo) {

```

```

        this.codigo = codigo;
    }

    @Column(name = "nome_proprietario", length = 60, nullable = false)
    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Column(name = "telefone_proprietario", length = 20, nullable = false)
    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    @Column(name = "email_proprietario", length = 255)
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @OneToMany(mappedBy = "proprietario")
    public List<Veiculo> getVeiculos() {
        return veiculos;
    }

    public void setVeiculos(List<Veiculo> veiculos) {
        this.veiculos = veiculos;
    }

    // hashCode e equals
}

```

```

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

    private Long codigo;
    private String descricao;
    private Set<Veiculo> veiculos = new HashSet<>();

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

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

    @Column(length = 60, nullable = false)
    public String getDescricao() {
        return descricao;
    }

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

    @ManyToMany(mappedBy = "acessorios", cascade = CascadeType.PERSIST)
    public Set<Veiculo> getVeiculos() {
        return veiculos;
    }

    public void setVeiculos(Set<Veiculo> veiculos) {
        this.veiculos = veiculos;
    }

    // hashCode e equals
}

```

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.

```
Query query = manager.createQuery(
    "select v from Veiculo v where anoFabricacao = 2012");
List veiculos = query.getResultList();

for (Object obj : veiculos) {
    Veiculo veiculo = (Veiculo) obj;

    System.out.println(veiculo.getModelo() + " " + veiculo.getFabricante()
        + ": " + veiculo.getAnoFabricacao());
}
```

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 veículos que possuem o ano de fabricação igual a 2012.

```
select v from Veiculo v where anoFabricacao = 2012
```

Essa consulta é o mesmo que:

```
from Veiculo where anoFabricacao = 2012
```

Veja que não somos obrigados a incluir a instrução `select` neste caso, e nem informar um *alias* para a entidade `Veiculo`.

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.

```
Query query = manager.createQuery(
    "from Veiculo where anoFabricacao >= :ano and valor <= :preco");
query.setParameter("ano", 2009);
query.setParameter("preco", new BigDecimal(60_000));
List veiculos = query.getResultList();

for (Object obj : veiculos) {
    Veiculo veiculo = (Veiculo) obj;

    System.out.println(veiculo.getModelo() + " " + veiculo.getFabricante()
        + ": " + veiculo.getAnoFabricacao());
}
```

Nomeamos os parâmetros com um prefixo `:`, e atribuímos 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.

```
TypedQuery<Veiculo> query = manager.createQuery("from Veiculo",  
    Veiculo.class);  
List<Veiculo> veiculos = query.getResultList();  
  
for (Veiculo veiculo : veiculos) {  
    System.out.println(veiculo.getModelo() + " " + veiculo.getFabricante()  
        + ": " + veiculo.getAnoFabricacao());  
}
```

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 maioria das 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 define a posição do primeiro registro (começando de 0) e o número máximo de registros que podem ser retornados, respectivamente.

```
TypedQuery<Veiculo> query = manager.createQuery("from Veiculo",  
    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());
}

```

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.

```
TypedQuery<Veiculo> query = manager.createQuery("from Veiculo",  
    Veiculo.class);  
  
List<Veiculo> veiculos = query.getResultList();  
  
for (Veiculo veiculo : veiculos) {  
    System.out.println(veiculo.getModelo());  
}
```

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 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 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=?

```

Podemos *projetar* apenas a propriedade da entidade que nos interessa usando a cláusula select.

```

TypedQuery<String> query = manager.createQuery(
    "select modelo from Veiculo", String.class);

List<String> modelos = query.getResultList();

for (String modelo : modelos) {
    System.out.println(modelo);
}

```

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*, 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.

```
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);
}
```

O operador new

Trabalhar com vetores de objetos não é nada agradável, é propenso a erros e deixa o código muito feio. Podemos criar uma classe para representar o resultado da consulta, com os atributos que desejamos e um construtor que recebe como parâmetro os dois valores.

```
public class PrecoVeiculo {

    private String modelo;
    private BigDecimal valor;

    public PrecoVeiculo(String modelo, BigDecimal valor) {
        super();
        this.modelo = modelo;
        this.valor = valor;
    }

    // getters e setters

}
```

Agora, 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*.

```
TypedQuery<PrecoVeiculo> query = manager.createQuery(
    "select new com.algaworks.vo.PrecoVeiculo(modelo, valor) from Veiculo",
    PrecoVeiculo.class);

List<PrecoVeiculo> precos = query.getResultList();

for (PrecoVeiculo preco : precos) {
    System.out.println(preco.getModelo() + " - " + preco.getValor());
}
```

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, criaremos uma classe chamada *TotalCarroPorAno*, para representar o resultado da consulta.

```
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 e setters

    }

```

Na *query* JPQL, usamos as funções de agregação `avg` e `count`, além da cláusula `group by`.

```

TypedQuery<TotalCarroPorAno> query = manager.createQuery(
    "select new com.algaworks.vo.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());
}

```

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
        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.

```

TypedQuery<Long> query = manager.createQuery(
    "select count(v) from Veiculo v", Long.class);

```

```
Long quantidadeVeiculos = query.getSingleResult();

System.out.println("Quantidade de veículos: " + quantidadeVeiculos);
```

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:

```
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());
}
```

Veja a consulta SQL gerada:

```
Hibernate:
    select
        proprietar0.codigo as codigol1_,
        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.

```

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]);
}

```

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 proprietario0_.codigo=veiculos1_.cod_proprietario
group by
    proprietario0_.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.

Hibernate:

```

select
    proprietario0_.nome_proprietario as col_0_0_,
    count(veiculos1_.codigo) as col_1_0_
from
    proprietario proprietario0_
left outer join
    veiculo veiculos1_
        on proprietario0_.codigo=veiculos1_.cod_proprietario
group by
    proprietario0_.nome_proprietario

```

Problema do N+1

Um problema bastante conhecido é o **Problema 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:

```

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

```

```

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

```

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 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 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=?

```

...

...
...

Para solucionar o problema, podemos usar os operadores `inner join fetch` para fazer *join* com a entidade `Proprietario` e trazer os dados na consulta.

```
TypedQuery<Veiculo> query = manager.createQuery(  
    "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());  
}
```

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
```

Vídeo aula sobre o Problema do N+1

No blog da AlgaWorks tem uma vídeo aula gratuita sobre o **Problema do N+1**, com uma simulação do problema e a correção dele em um projeto web.

<http://blog.algaworks.com/o-problema-do-n-mais-um/>

6.11. Queries nomeadas

As 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 = "veiculo")
@NamedQuery(name = "Veiculo.comProprietarioPorValor",
    query = "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`.

```

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());
}

```

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 = "from Veiculo v "
            + "inner join fetch v.proprietario where v.valor > :valor"),
    @NamedQuery(name = "Veiculo.porModelo",
        query = "from Veiculo where modelo like :modelo")
})
public class Veiculo {

    ...

}

```

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* de 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_1.xsd"
    version="2.1">

    <named-query name="Veiculo.anoFabricacaoMenor">
        <query><![CDATA[
            from Veiculo where anoFabricacao < :ano
        ]]></query>
    </named-query>

</entity-mappings>

```

No arquivo acima, definimos uma *named query* chamada “Veiculo.anoFabricacaoMenor”, e podemos usá-la normalmente.

Vídeo aula sobre *named queries* em arquivos externos

Para não ficar qualquer dúvida, assista a vídeo aula gratuita no blog da AlgaWorks que explica passo a passo como usar *named queries* em arquivos externos.

<http://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 objetos mapeados. Existem alguns casos bastante específicos, em que você precisará utilizar SQL nativo para consultar seu banco de dados.

No exemplo abaixo, pesquisamos todos os veículos usando SQL, que retorna entidades gerenciadas pelo contexto de persistência.

```

Query query = manager.createNativeQuery("select * from veiculo",
    Veiculo.class);
List<Veiculo> veiculos = query.getResultList();

```

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

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.

Vamos começar com uma das consultas mais simples que poderíamos fazer usando Criteria API.

```
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());
}
```

A consulta acima busca todos os veículos cadastrados e imprime o modelo deles. Seria o mesmo que fazer em JPQL:

`from Veiculo`

Claro que, em uma consulta simples como essa, seria melhor usarmos JPQL, mas você conhecerá a facilidade da API, em 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*.

```
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());
}

```

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*.

```
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);
}
```

7.4. Funções de agregação

As funções de agregação são representadas por métodos de `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.

```
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);
```

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`.

```
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]);
}
```

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.

```
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"));
}
```

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.

```
public class PrecoVeiculo {

    private String modelo;
    private BigDecimal valor;

    public PrecoVeiculo(String modelo, BigDecimal valor) {
        super();
        this.modelo = modelo;
        this.valor = valor;
    }

    // getters e setters

}
```

Depois, 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.

```
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());
}

```

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*.

```

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"), "i30".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());
}

```

Já no próximo exemplo, programamos uma consulta que retorna uma lista de *strings*, com o fabricante e modelo concatenados e separados pelo caractere "-".

```

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);
}

```

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.

```

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());
}

```

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*.

```
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"), "João"));

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

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

Fetch

Para evitar o **Problema N+1**, que já estudamos anteriormente, podemos fazer um *fetch* no relacionamento da entidade *Veiculo*.

```
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"), "João"));

TypedQuery<Veiculo> query = manager.createQuery(criteriaQuery);
List<Veiculo> veiculos = query.getResultList();

for (Veiculo v : veiculos) {
```

```

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

```

Todos os dados necessários são retornados por apenas uma consulta SQL:

Hibernate:

```

select
    veiculo0_.codigo as codigo1_2_0_,
    proprietario1_.codigo as codigo1_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_,
    proprietario1_.email_proprietario as email_pr2_1_1_,
    proprietario1_.nome_proprietario as nome_pro3_1_1_,
    proprietario1_.telefone_proprietario as telefone4_1_1_
from
    veiculo veiculo0_
inner join
    proprietario proprietario1_
        on veiculo0_.cod_proprietario=proprietario1_.codigo
where
    proprietario1_.nome_proprietario=?

```

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.

```

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());
}

```

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_
    )

```

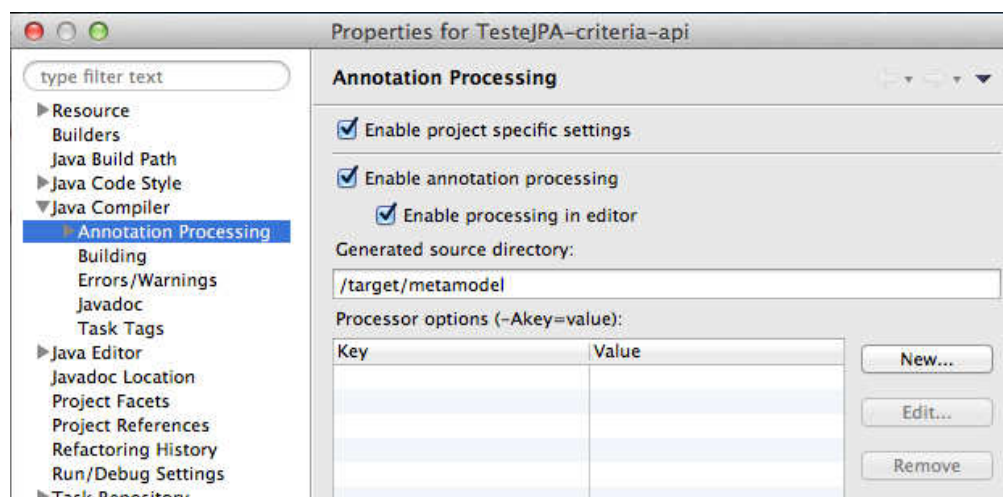
7.10. Metamodel

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.

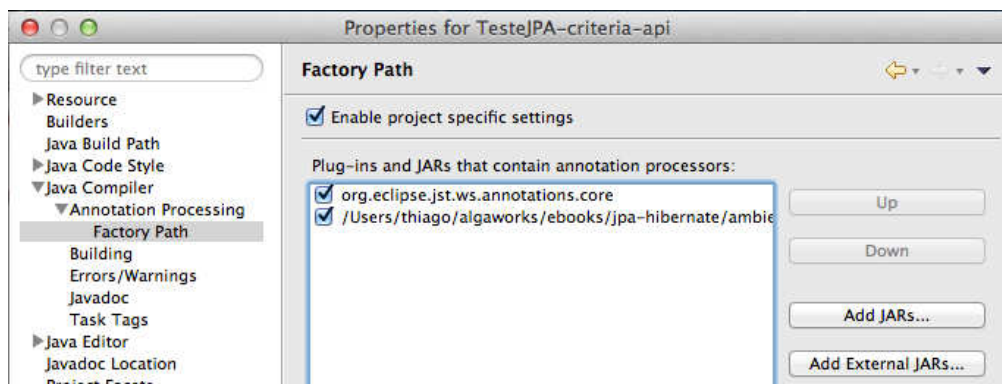
O uso de metamodelo evita erros em tempo de execução, pois qualquer alteração no código-fonte de mapeamento altera também o metamodelo.

Podemos gerar o *metamodel* automaticamente. Para isso, precisamos configurar o **Hibernate Metamodel Generator**.

No Eclipse, acesse as propriedades do projeto, encontre **Java Compiler** e depois **Annotation Processing**. Deixe as opções como a tela abaixo:



Abra o menu **Annotation Processing** e clique em **Factory Path**. Adicione o arquivo JAR *hibernate-jpamodelgen-x.x.x.Final.jar*, depois, clique em **OK**. Esse arquivo é distribuído junto com o Hibernate ORM e fica na pasta *lib/jpa-metamodel-generator*.



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),
    "João"));
```

```
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 vários truques sobre JPA e Hibernate.

Se você gostou desse livro, por favor, nos ajude a manter esse trabalho. Recomende o livro para seus amigos de trabalho, faculdade e/ou compartilhe no Facebook e Twitter.

8.1. Próximos passos

Embora a gente tenha dedicado bastante para escrever esse livro, o conteúdo que você aprendeu nele é só a ponta do iceberg!

É claro que você não perdeu tempo com o que acabou de estudar, mas o que queremos dizer é que há muito mais coisas importantes para aprofundar em JPA e Hibernate.

Caso você tenha interesse em mergulhar fundo em conteúdos ainda mais avançados, recomendo que você dê uma olhada em nosso curso online:

- JPA e Hibernate além do básico - um projeto completo
<http://www.algaworks.com/curso/jpa-e-hibernate/>

Além de tudo que você vai aprender, nós ainda oferecemos suporte para as suas dúvidas!

CURSOS ONLINE

COM VÍDEO AULAS E SUPORTE



Java e Orientação a Objetos



Desenvolvimento Web
com JSF 2



JPA e Hibernate além do básico
- um projeto completo



Sistemas Comerciais Java EE
com CDI, JPA e PrimeFaces

www.algaworks.com

Cursos presenciais in-company? Entre em contato.

JPA E HIBERNATE

THIAGO FARIA
NORMANDES JR