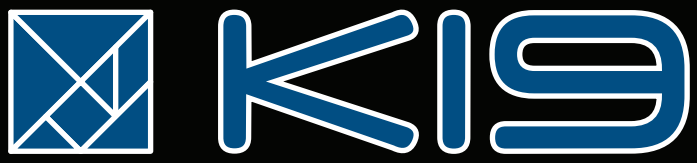


TREINAMENTOS

Persistência com JPA2 e Hibernate

Persistência com JPA 2.1 e Hibernate

22 de agosto de 2015
As apostilas atualizadas estão disponíveis em **www.k19.com.br**

Esta apostila contém:

- 211 exercícios de fixação.
- 5 exercícios complementares.
- 0 desafios.
- 0 questões de prova.

Sumário i

Sobre a K19 1

Seguro Treinamento 2

Termo de Uso 3

Cursos 4

Preparação do ambiente 5

1 Introdução 1
- 1.1 Persistência . . . 1
- 1.2 Configuração . . . 1
- 1.3 Mapeamento . . . 2
- 1.4 Gerando o banco . . . 3
- 1.5 Exercícios de Fixação . . . 3
- 1.6 Exercícios Complementares . . . 5
- 1.7 Manipulando entidades . . . 6
- 1.8 Exercícios de Fixação . . . 7
- 1.9 Exercícios Complementares . . . 9

2 Mapeamento 11
- 2.1 Entidades . . . 11

2.2 Definindo Restrições . . . 12
2.3 Gerando chaves primárias automaticamente . . . 12
2.4 Mapeamento Automático . . . 14
2.5 Objetos Grandes (LOB) . . . 15
2.6 Data e Hora . . . 15
2.7 Dados Transientes . . . 16
2.8 Field Access e Property Access . . . 16
2.9 Exercícios de Fixação . . . 17
2.10 Enums . . . 19
2.11 Coleções . . . 20
2.12 Relacionamentos . . . 21
2.13 One to One . . . 22
2.14 Exercícios de Fixação . . . 24
2.15 One to Many . . . 25
2.16 Exercícios de Fixação . . . 27
2.17 Many to One . . . 29
2.18 Exercícios de Fixação . . . 30
2.19 Many to Many . . . 32
2.20 Exercícios de Fixação . . . 33
2.21 Relacionamentos Bidirecionais . . . 35
2.22 Exercícios de Fixação . . . 36
2.23 Objetos Embutidos . . . 38
2.24 Exercícios de Fixação . . . 40
2.25 Herança . . . 42
2.26 Exercícios de Fixação . . . 45
2.27 Attribute Converter . . . 46
2.28 Exercícios de Fixação . . . 48

3 Entity Manager 51
3.1 Estados . . . 51
3.2 Sincronização com o Banco de Dados . . . 51
3.3 Transições . . . 53
3.4 Exercícios de Fixação . . . 54
3.5 LAZY e EAGER . . . 59
3.6 Exercícios de Fixação . . . 61
3.7 Caching . . . 64
3.8 Persistence Context ou Cache de Primeiro Nível . . . 65
3.9 Exercícios de Fixação . . . 65
3.10 Shared Cache ou Cache de Segundo Nível . . . 65
3.11 Exercícios de Fixação . . . 66
3.12 Cascade . . . 68
3.13 Exercícios de Fixação . . . 69
3.14 Remoção de Objetos Órfãos . . . 70
3.15 Exercícios de Fixação . . . 72
3.16 Callbacks . . . 74
3.17 Exercícios de Fixação . . . 77
3.18 Concorrência . . . 78
3.19 Exercícios de Fixação . . . 79
3.20 Locking Otimista . . . 80

3.21 Exercícios de Fixação . . . 81
3.22 Locking Pessimista . . . 81
3.23 Exercícios de Fixação . . . 82

4 JPQL **83**
4.1 Consultas Dinâmicas . . . 83
4.2 Named Query . . . 83
4.3 Parâmetros . . . 84
4.4 Exercícios de Fixação . . . 85
4.5 Tipos de Resultado . . . 89
4.6 Exercícios de Fixação . . . 93
4.7 Paginação . . . 96
4.8 Exercícios de Fixação . . . 96
4.9 O Problema do $N+1$ . . . 97
4.10 Exercícios de Fixação . . . 98
4.11 Operações em Lote (Bulk Operations) . . . 99
4.12 Exercícios de Fixação . . . 100
4.13 Operadores . . . 102
4.14 Exemplos . . . 106
4.15 Referências . . . 107
4.16 Consultas Nativas . . . 107
4.17 Exercícios de Fixação . . . 107
4.18 Stored Procedures . . . 108
4.19 Exercícios de Fixação . . . 108

5 Criteria **111**
5.1 Necessidade . . . 111
5.2 Estrutura Geral . . . 111
5.3 Exercícios de Fixação . . . 112
5.4 Exercícios Complementares . . . 115
5.5 Tipos de Resultados . . . 115
5.6 Exercícios de Fixação . . . 117
5.7 Filtros e Predicados . . . 119
5.8 Exercícios de Fixação . . . 119
5.9 Lista de Predicados . . . 120
5.10 Funções . . . 123
5.11 Ordenação . . . 124
5.12 Subqueries . . . 124
5.13 Exemplos . . . 125
5.14 O Problema do $N+1$ . . . 126
5.15 Exercícios de Fixação . . . 127
5.16 Operações em Lote (Bulk Operations) . . . 128
5.17 Exercícios de Fixação . . . 129
5.18 Metamodel . . . 131
5.19 Exercícios de Fixação . . . 132

A Hibernate Search **135**
A.1 Configuração . . . 135
A.2 Mapeamento . . . 136
A.3 Indexação . . . 136

A.4 Busca . . . 137
A.5 Exercícios de Fixação . . . 137

B Hibernate Envers **141**
B.1 Mapeamento . . . 141
B.2 Consultas . . . 142
B.3 Exercícios de Fixação . . . 142

C Bean Validation e Hibernate Validator **147**
C.1 Regras de Validação . . . 147
C.2 Processando as Validações . . . 148
C.3 Exercícios de Fixação . . . 149

D Mapeamento com XML **153**
D.1 Entidades . . . 153
D.2 Definindo Restrições . . . 155
D.3 Gerando chaves primárias automaticamente . . . 156
D.4 Mapeamento Automático . . . 156
D.5 Objetos Grandes (LOB) . . . 157
D.6 Data e Hora . . . 157
D.7 Dados Transientes . . . 158
D.8 Field Access e Property Access . . . 159
D.9 Exercícios de Fixação . . . 159
D.10 Enums . . . 161
D.11 Coleções . . . 163
D.12 Relacionamentos . . . 164
D.13 One to One . . . 165
D.14 Exercícios de Fixação . . . 168
D.15 One to Many . . . 169
D.16 Exercícios de Fixação . . . 171
D.17 Many to One . . . 173
D.18 Exercícios de Fixação . . . 175
D.19 Many to Many . . . 177
D.20 Exercícios de Fixação . . . 179
D.21 Relacionamentos Bidirecionais . . . 180
D.22 Exercícios de Fixação . . . 182
D.23 Objetos Embutidos . . . 185
D.24 Exercícios de Fixação . . . 187
D.25 Herança . . . 189
D.26 Exercícios de Fixação . . . 193

E Respostas **195**

Sobre a K19

A K19 é uma empresa especializada na capacitação de desenvolvedores de software. Sua equipe é composta por profissionais formados em Ciência da Computação pela Universidade de São Paulo (USP) e que possuem vasta experiência em treinamento de profissionais para área de TI.

O principal objetivo da K19 é oferecer treinamentos de máxima qualidade e relacionados às principais tecnologias utilizadas pelas empresas. Através desses treinamentos, seus alunos tornam-se capacitados para atuar no mercado de trabalho.

Visando a máxima qualidade, a K19 mantém as suas apostilas em constante renovação e melhoria, oferece instalações físicas apropriadas para o ensino e seus instrutores estão sempre atualizados didática e tecnicamente.

Seguro Treinamento

Na K19 o aluno faz o curso quantas vezes quiser!

Comprometida com o aprendizado e com a satisfação dos seus alunos, a K19 é a única que possui o Seguro Treinamento. Ao contratar um curso, o aluno poderá refazê-lo quantas vezes desejar mediante a disponibilidade de vagas e pagamento da franquia do Seguro Treinamento.

As vagas não preenchidas até um dia antes do início de uma turma da K19 serão destinadas ao alunos que desejam utilizar o Seguro Treinamento. O valor da franquia para utilizar o Seguro Treinamento é 10% do valor total do curso.

Termo de Uso

Termo de Uso

Todo o conteúdo desta apostila é propriedade da K19 Treinamentos. A apostila pode ser utilizada livremente para estudo pessoal . Além disso, este material didático pode ser utilizado como material de apoio em cursos de ensino superior desde que a instituição correspondente seja reconhecida pelo MEC (Ministério da Educação) e que a K19 seja citada explicitamente como proprietária do material.

É proibida qualquer utilização desse material que não se enquadre nas condições acima sem o prévio consentimento formal, por escrito, da K19 Treinamentos. O uso indevido está sujeito às medidas legais cabíveis.

Conheça os nossos cursos

K01- Lógica de Programação

K02 - Desenvolvimento Web com HTML, CSS e JavaScript

K03 - SQL e Modelo Relacional

K11 - Orientação a Objetos em Java

K12 - Desenvolvimento Web com JSF2 e JPA2

K21 - Persistência com JPA2 e Hibernate

K22 - Desenvolvimento Web Avançado com JFS2, EJB3.1 e CDI

K23 - Integração de Sistemas com Webservices, JMS e EJB

K41 - Desenvolvimento Mobile com Android

K51 - Design Patterns em Java

K52 - Desenvolvimento Web com Struts

K31 - C# e Orientação a Objetos

K32 - Desenvolvimento Web com ASP.NET MVC

www.k19.com.br/cursos

Preparação do ambiente

Para realizar os exercícios desta apostila, recomendamos que você utilize um dos sistemas operacionais abaixo.

- Ubuntu 12.04 (ou superior) (preferencialmente)
- Windows 7 (ou superior)

Ubuntu

Se você optar por utilizar o Ubuntu, siga o tutorial abaixo para preparar o seu ambiente.

- www.k19.com.br/artigos/como-instalar-o-mysql-server-no-ubuntu-13-10
- www.k19.com.br/artigos/como-instalar-o-jdk-7-no-ubuntu-13-10
- www.k19.com.br/artigos/como-instalar-o-eclipse-kepler

Windows

Se você optar por utilizar o Windows, siga os tutoriais abaixo para preparar o seu ambiente.

- www.k19.com.br/artigos/como-instalar-mysql-server-no-windows-8
- www.k19.com.br/artigos/como-instalar-o-jdk-7-no-windows-8
- www.k19.com.br/artigos/como-instalar-o-eclipse-kepler

Arquivos

Diversos arquivos são utilizados nos exercícios desta apostila, você pode obtê-los no endereço:

- https://github.com/K19/K19-Arquivos/archive/K19-Arquivos.zip

CAPÍTULO 1

INTRODUÇÃO

Persistência

Aplicações corporativas manipulam dados em grande quantidade. Na maioria dos casos, esses dados são armazenados em bancos de dados relacionais, pois os principais sistemas gerenciadores de bancos de dados do mercado utilizam o modelo relacional. Por outro lado, hoje em dia, as aplicações corporativas costumam ser desenvolvidas com linguagens orientadas a objetos.

Como o modelo relacional e o modelo orientado a objetos diferem no modo de estruturar os dados, uma transformação deve ocorrer toda vez que alguma informação trafegar da aplicação para o banco de dados ou vice-versa. Essa transformação não é simples, pois os dois modelos são bem diferentes.

No contexto das aplicações Java, para facilitar o processo de transformação dos dados que trafegam entre as aplicações e os bancos de dados, podemos utilizar algumas ferramentas de persistência como o **Hibernate** ou o **EclipseLink**.

Essas ferramentas funcionam como intermediários entre as aplicações e os bancos de dados, automatizando diversos processos importantes relacionados à persistência dos dados. Elas são chamadas de ferramentas ORM (Object Relational Mapping).

Com o intuito de facilitar a utilização dessas ferramentas e torná-las compatíveis com os outros recursos da plataforma Java, elas são padronizadas pela especificação **Java Persistence API (JPA)**.

Veremos, nesse capítulo, os passos principais para utilizar uma implementação da JPA. Em particular, utilizaremos o Hibernate e o sistema gerenciador de banco de dados MySQL.

Configuração

Antes de começar a utilizar o Hibernate, é necessário baixar do site oficial o **bundle** que inclui os jar's do hibernate e todas as suas dependências. Neste curso, utilizaremos a versão 4.3.10. A url do site oficial do Hibernate é `http://www.hibernate.org/`.

Para configurar o Hibernate em uma aplicação, devemos criar um arquivo chamado **persistence.xml**. O conteúdo desse arquivo contém informações sobre o banco de dados, como a url de conexão, usuário e senha, além de dados sobre a implementação JPA que será utilizada.

O arquivo `persistence.xml` deve ser salvo em uma pasta chamada **META-INF**, que deve estar no classpath da aplicação. Veja abaixo um exemplo de configuração para o `persistence.xml`:

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

```
3   version="2.1"
4   xmlns="http://xmlns.jcp.org/xml/ns/persistence"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
7     http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd ">
8
9   <persistence-unit name="livraria-pu" transaction-type="RESOURCE_LOCAL">
10    <provider>org.hibernate.ejb.HibernatePersistence</provider>
11    <properties>
12      <property
13        name="hibernate.dialect"
14        value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15
16      <property
17        name="hibernate.hbm2ddl.auto"
18        value="create" />
19
20      <property
21        name="javax.persistence.jdbc.driver"
22        value="com.mysql.jdbc.Driver" />
23
24      <property
25        name="javax.persistence.jdbc.user"
26        value="root" />
27
28      <property
29        name="javax.persistence.jdbc.password"
30        value="root" />
31
32      <property
33        name="javax.persistence.jdbc.url"
34        value="jdbc:mysql://localhost:3306/livraria" />
35    </properties>
36  </persistence-unit>
37</persistence>
```

Código XML 1.1: persistence.xml

Mapeamento

Um dos principais objetivos dos frameworks ORM é estabelecer o mapeamento entre os conceitos do modelo orientado a objetos e os conceitos do modelo relacional. Este mapeamento pode ser definido através de xml ou de maneira mais prática com anotações Java. Quando utilizamos anotações, evitamos a criação de extensos arquivos em xml.

A seguir, veremos as principais anotações Java de mapeamento do JPA. Essas anotações estão no pacote **javax.persistence**.

@Entity É a principal anotação do JPA. Ela deve aparecer antes do nome de uma classe e deve ser definida em todas as classes que terão objetos persistidos no banco de dados.

As classes anotadas com `@Entity` são mapeadas para tabelas. Por convenção, as tabelas possuem os mesmos nomes das classes. Mas, podemos alterar esse comportamento utilizando a anotação `@Table`.

Os atributos declarados em uma classe anotada com `@Entity` são mapeados para colunas na tabela correspondente à classe. Outra vez, por convenção, as colunas possuem os mesmos nomes dos atributos. E novamente, podemos alterar esse padrão utilizando para isso a anotação `@Column`.

@Id É utilizada para indicar qual atributo de uma classe anotada com `@Entity` será mapeado para a chave primária da tabela correspondente à classe. Geralmente o atributo anotado com `@Id` é do tipo `Long`.

@GeneratedValue Geralmente vem acompanhado da anotação `@Id`. Serve para indicar que o valor de um atributo que compõe uma chave primária deve ser gerado pelo banco no momento em que um novo registro é inserido.

Gerando o banco

Uma das vantagens de utilizar uma implementação JPA é que ela é capaz de gerar as tabelas no banco de dados. Ela faz isso de acordo com as anotações colocadas nas classes e as informações presentes no arquivo `persistence.xml`.

As tabelas são geradas através de um método estático da classe `Persistence`, o `createEntityManagerFactory(string persistenceUnit)`. O parâmetro `persistenceUnit` permite escolher, pelo nome, uma unidade de persistência definida no `persistence.xml`.

```
1 Persistence.createEntityManagerFactory("K19-PU");
```

Código Java 1.1: Inicializando uma unidade de persistência.

Exercícios de Fixação

1 Para apresentar os conceitos básicos de JPA, implementaremos parte de um sistema de gerenciamento de uma livraria. Primeiramente, crie um projeto no Eclipse chamado **K19-JPA2-Hibernate**.

2 Crie uma pasta chamada **lib** dentro do projeto **K19-JPA2-Hibernate**.

3 Entre na pasta **K19-Arquivos/hibernate-release-VERSAO.Final/lib** da Área de Trabalho e copie os jars da pasta **required** e da pasta **jpa** para a pasta **lib** do projeto **K19-JPA2-Hibernate**.

Importante
Você também pode obter esses arquivos através do site da K19: `www.k19.com.br/arquivos`.

4 Entre na pasta **K19-Arquivos/mysql-connector-java-VERSAO** da Área de Trabalho e copie o arquivo `mysql-connector-java-VERSAO-bin.jar` para pasta **lib** do projeto **K19-JPA2-Hibernate**.

Importante
Você também pode obter o arquivo **mysql-connector-java-VERSAO-bin.jar** através do site da K19: www.k19.com.br/arquivos.

5 Adicione os jar's da pasta **lib** ao **build path** do projeto **K19-JPA2-Hibernate**. Peça orientação do instrutor se for necessário.

6 Crie uma pasta chamada **META-INF** dentro da pasta **src** do projeto **K19-JPA2-Hibernate**.

7 Crie o arquivo de configurações **persistence.xml** na pasta **META-INF**. Para não ter de digitar todo o código, copie o modelo **persistence.xml** da pasta **K19-Arquivos/modelos** da sua Área de Trabalho. Altere esse modelo de acordo com o código abaixo.

```
<?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="K21_livraria_pu" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property
        name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5InnoDBDialect" />

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

      <property
        name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver" />

      <property
        name="javax.persistence.jdbc.user"
        value="root" />

      <property
        name="javax.persistence.jdbc.password"
        value="root" />

      <property
        name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/K21_livraria_bd" />
    </properties>
  </persistence-unit>
</persistence>
```

Código XML 1.2: persistence.xml

8 Crie uma classe para modelar as editoras da nossa livraria e acrescente as anotações necessárias para fazer o mapeamento. Essas anotações devem ser importadas do pacote `javax.persistence`. Adicione essa classe em um pacote chamado `br.com.k19.modelo`.

```java
package br.com.k19.modelo;

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

@Entity
public class Editora {
  @Id
  @GeneratedValue
  private Long id;

  private String nome;

  private String email;

  // GETTERS E SETTERS
}
```

Código Java 1.2: Editora.java

9 Apague a base dados **K21_livraria_bd** se ela existir através do MySQL Workbench. Peça orientação do instrutor se for necessário.

10 Crie a base dados **K21_livraria_bd** através do MySQL Workbench. Peça orientação do instrutor se for necessário.

11 Gere as tabelas usando o método `createEntityManagerFactory()` da classe `Persistence`. Para isso, crie uma classe com método `main` em um pacote chamado `br.com.k19.testes` no projeto **K19-JPA2-Hibernate**. Obs: As classes devem ser importadas do pacote `javax.persistence`.

```java
package br.com.k19.testes;

import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class GeraTabelas {
  public static void main(String[] args) {
    EntityManagerFactory factory =
      Persistence.createEntityManagerFactory("K21_livraria_pu");

    factory.close();
  }
}
```

Código Java 1.3: GeraTabelas.java

Execute e verifique através do MySQL Workbench se a tabela `Editora` foi criada corretamente.

Exercícios Complementares

1 Na pacote `br.com.k19.modelo` do projeto **K19-JPA2-Hibernate**, crie uma classe chamada `Autor` para modelar um autor. Essa classe deve conter dois atributos: um para armazenar o id do autor e outra para armazenar o nome do autor.

2 Gere novamente as tabelas da base de dados `K19_livraria_bd` usando o método `createEntityManagerFactory()` da classe `Persistence`. Através do MySQL Workbench, verifique se a tabela `Autor` foi criada corretamente.

Manipulando entidades

Para manipular as entidades da nossa aplicação, devemos utilizar um `EntityManager` que é obtido através de uma `EntityManagerFactory`.

```
1 EntityManagerFactory factory =
2   Persistence.createEntityManagerFactory("K19-PU");
3
4 EntityManager manager = factory.createEntityManager();
```

Código Java 1.6: Obtendo um Entity Manager

Persistindo

Para armazenar as informações de um objeto no banco de dados, devemos utilizar o método `persist()` do `EntityManager`.

```
1 Editora novaEditora = new Editora();
2 novaEditora.setNome("K19 - Livros")
3 novaEditora.setEmail("contato@k19.com.br");
4
5 manager.persist(novaEditora);
```

Código Java 1.7: Associando um objeto a um Entity Manager.

Buscando

Para obter um objeto que contenha informações do banco de dados, podemos utilizar os métodos `find()` ou `getReference()` do `EntityManager`.

```
1 Editora editora1 = manager.find(Editora.class, 1L);
2 Editora editora2 = manager.getReference(Editora.class, 2L);
```

Código Java 1.8: Buscando objetos.

Há uma diferença entre esses métodos de busca. O método `find()` recupera os dados desejados imediatamente. Já o método `getReference()` posterga essa tarefa até a primeira chamada de um método `get` no objeto desejado.

Removendo

Para remover um registro correspondente a um objeto, podemos utilizar o método `remove()` do `EntityManager`.

```
1 Editora editora = manager.find(Editora.class, 1L);
2 manager.remove(editora);
```

Código Java 1.9: Marcando um objeto para ser removido.

Atualizando

Para alterar os dados de um registro correspondente a um objeto, podemos utilizar os próprios métodos setters desse objeto.

```
1 Editora editora = manager.find(Editora.class, 1L);
2 editora.setNome("K19 - Livros e Publicações");
```

Código Java 1.10: Alterando o conteúdo de um objeto.

Listando

Para obter uma listagem com todos os objetos referentes aos registros de uma tabela, podemos utilizar a linguagem de consulta do JPA, a **JPQL**. Essa linguagem é muito parecida com a linguagem SQL. A vantagem da JPQL em relação à linguagem SQL é que a sintaxe é a mesma para bancos de dados diferentes.

```
1 Query query = manager.createQuery("SELECT e FROM Editora e");
2 List<Editora> editoras = query.getResultList();
```

Código Java 1.11: Definindo e executando uma consulta JPQL.

Transações

As modificações realizadas nos objetos administrados por um `EntityManager` são mantidas em memória. Em certos momentos, é necessário sincronizar os dados da memória com os dados do banco de dados. Essa sincronização deve ser realizada através de uma transação JPA criada pelo `EntityManager` que administra os objetos que desejamos sincronizar.

Para abrir uma transação utilizamos o método `begin()`.

```
1 manager.getTransaction().begin();
```

Código Java 1.12: Abrindo uma transação.

Com a transação aberta, podemos sincronizar os dados com o banco através dos métodos `flush()` (parcialmente) ou `commit()` (definitivamente).

```
1 Editora editora = manager.find(Editora.class, 1L);
2 editora.setNome("K19 - Livros e Publicações");
3
4 manager.getTransaction().begin();
5 manager.flush();
```

Código Java 1.13: Sincronizando parcialmente uma transação.

```
1 Editora editora = manager.find(Editora.class, 1L);
2 editora.setNome("K19 - Livros e Publicações");
3
4 manager.getTransaction().begin();
5 manager.getTransaction().commit();
```

Código Java 1.14: Sincronizando definitivamente uma transação

Exercícios de Fixação

12 No pacote `br.com.k19.testes` do projeto **K19-JPA2-Hibernate**, crie um teste para inserir editoras no banco de dados.

```
1  package br.com.k19.testes;
2
3  import java.util.Scanner;
4
5  import javax.persistence.EntityManager;
6  import javax.persistence.EntityManagerFactory;
7  import javax.persistence.Persistence;
8
9  import br.com.k19.modelo.Editora;
10
11 public class InsereEditoraComJPA {
12
13   public static void main(String[] args) {
14     EntityManagerFactory factory =
15         Persistence.createEntityManagerFactory("K21_livraria_pu");
16
17     EntityManager manager = factory.createEntityManager();
18
19     Editora novaEditora = new Editora();
20
21     Scanner entrada = new Scanner(System.in);
22
23     System.out.println("Digite o nome da editora: ");
24     novaEditora.setNome(entrada.nextLine());
25
26     System.out.println("Digite o email da editora: ");
27     novaEditora.setEmail(entrada.nextLine());
28
29     entrada.close();
30
31     manager.persist(novaEditora);
32
33     manager.getTransaction().begin();
34     manager.getTransaction().commit();
35
36     factory.close();
37   }
38 }
```

Código Java 1.15: InsereEditoraComJPA.java

Execute a classe `InsereEditoraComJPA`.

13 No pacote `br.com.k19.testes` do projeto **K19-JPA2-Hibernate**, crie um teste para listar as editoras inseridas no banco de dados. No código abaixo, a interface `Query` deve ser importada do pacote `javax.persistence` e a interface `List` do pacote `java.util`.

```
1  package br.com.k19.testes;
2
3  import java.util.List;
4
5  import javax.persistence.EntityManager;
6  import javax.persistence.EntityManagerFactory;
7  import javax.persistence.Persistence;
8  import javax.persistence.Query;
9
10 import br.com.k19.modelo.Editora;
11
12 public class ListaEditorasComJPA {
```

```
13
14   public static void main(String[] args) {
15     EntityManagerFactory factory =
16         Persistence.createEntityManagerFactory("K21_livraria_pu");
17
18     EntityManager manager = factory.createEntityManager();
19
20     Query query = manager.createQuery("SELECT e FROM Editora e");
21     List<Editora> editoras = query.getResultList();
22
23     for (Editora e : editoras) {
24       System.out.println("EDITORA: " + e.getNome() + " - " + e.getEmail());
25     }
26   }
27 }
```

Código Java 1.16: ListaEditorasComJPA.java

Execute a classe `ListaEditorasComJPA`.

Exercícios Complementares

3 No pacote `br.com.k19.testes` do projeto **K19-JPA2-Hibernate**, crie um teste para inserir autores no banco de dados.

4 No pacote `br.com.k19.testes` do projeto **K19-JPA2-Hibernate**, crie um teste para listar os autores inseridos no banco de dados.

CAPÍTULO 2

MAPEAMENTO

O mapeamento objeto-relacional é o coração do Hibernate e das outras implementações de JPA. Ele define quais transformações devem ser realizadas nos dados para que essas informações possam navegar da aplicação para o banco de dados ou do banco de dados para a aplicação. Em particular, o mapeamento determina como a ferramenta ORM fará consultas complexas envolvendo mais do que uma tabela.

Entidades

As classes da nossa aplicação que devem ser mapeadas para tabelas do banco de dados são anotadas com **@Entity**. Cada instância de uma classe anotada com `@Entity` deve possuir um identificador único. Em geral, esse identificador é um atributo numérico que deve ser anotado com **@Id**.

```
1 @Entity
2 class Pessoa {
3   @Id
4   private Long id;
5 }
```

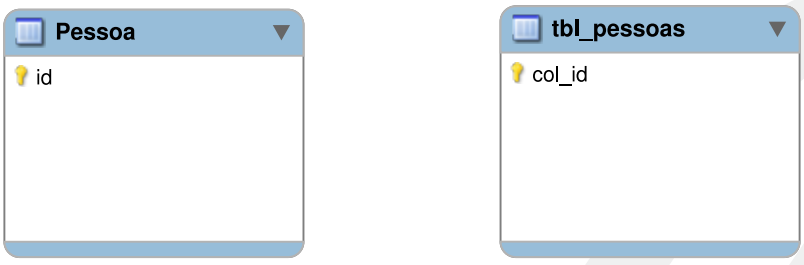
Código Java 2.1: Pessoa.java

Por convenção, a classe `Pessoa` será mapeada para uma tabela com o mesmo nome (`Pessoa`). O atributo `id` será mapeado para uma coluna com o mesmo nome (`id`) na tabela `Pessoa`. As anotações **@Table** e **@Column** podem ser usadas para personalizar os nomes das tabelas e das colunas.

A coluna correspondente ao atributo `id` será definida como chave primária da tabela `Pessoa` devido a presença da anotação `@Id`.

```
1 @Entity
2 @Table(name = "tbl_pessoas")
3 class Pessoa {
4   @Id
5   @Column(name = "col_id")
6   private Long id;
7 }
```

Código Java 2.2: Pessoa.java



(a) (b)

Figura 2.1: Tabelas correspondentes à classe `Pessoa`. *Em (a), os nomes da tabela e da coluna são padrões. Em (b), esses nomes são personalizados.*

Definindo Restrições

Podemos definir algumas restrições para os atributos das nossas entidades através das propriedades da anotação `@Column`. Veja as principais propriedades abaixo:

`length`	Limita a quantidade de caracteres de uma string
`nullable`	Determina se o campo pode possuir valores `null` ou não
`unique`	Determina se uma coluna pode ter valores repetidos ou não
`precision`	Determina a quantidade de dígitos de um número decimal a serem armazenadas
`scale`	Determina a quantidade de casas decimais de um número decimal

Tabela 2.1: Algumas propriedades da anotação `@Column`

No exemplo a seguir, associamos três restrições ao atributo `nome` da classe `Pessoa`. O nome deve possuir no máximo 30 caracteres, não pode ser nulo e duas pessoas não podem ter o mesmo nome. Além disso, definimos que a altura das pessoas será representada por um número de três dígitos, sendo dois deles referentes às casas decimais.

```
1  @Entity
2  class Pessoa {
3    @Id
4    private Long id;
5
6    @Column(length=30,nullable=false,unique=true)
7    private String nome;
8
9    @Column(precision=3,scale=2)
10   private BigDecimal altura;
11 }
```

Código Java 2.3: Pessoa.java

Gerando chaves primárias automaticamente

Em geral, os bancos de dados oferecem algum mecanismo para gerar os valores de uma chave primária simples e numérica. Do ponto de vista do desenvolvedor JPA, basta aplicar a anotação **@GeneratedValue** para que o banco gere os valores de uma chave primária simples e numérica au-

tomaticamente.

```
@Entity
class Pessoa {
  @Id
  @GeneratedValue
  private Long id;
}
```

Código Java 2.4: Pessoa.java

Ao utilizarmos a anotação **@GeneratedValue** sem parâmetros fazemos com que a estratégia **GenerationType.AUTO** seja utilizada, ou seja, deixamos a cargo da implementação do JPA decidir qual a melhor estratégia de geração dos valores para o atributo de acordo com o banco de dados escolhido. O código abaixo é equivalente ao anterior, porém informando explicitamente que queremos utilizar a estratégia automática.

```
@Entity
class Pessoa {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;
}
```

Código Java 2.5: Pessoa.java

Se estivéssemos utilizando o banco de dados MySQL em conjunto com o Hibernate, o uso da estratégia automática faria com que o Hibernate escolhesse a estratégia **GenerationType.IDENTITY**. Dessa forma, o código da classe **Pessoa.java** produziria a intrução SQL de criação da tabela **Pessoa** semelhante à apresentada abaixo:

```
CREATE TABLE `Pessoa` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

Código SQL 2.1: Instrução de criação da tabela 'Pessoa' no MySQL

Caso estivéssemos utilizando o banco de dados PostgreSQL em conjunto com o Hibernate, o uso da estratégia automática faria com que o Hibernate escolhesse a estratégia **GenerationType.SEQUENCE**. Dessa forma, o código da classe **Pessoa.java** produziria as instruções SQL de criação da tabela **Pessoa** semelhante às apresentadas abaixo:

```
CREATE TABLE pessoa
(
  id bigint NOT NULL,
  CONSTRAINT pessoa_pkey PRIMARY KEY (id)
)
WITH (
  OIDS=FALSE
);
ALTER TABLE pessoa
  OWNER TO k19;

CREATE SEQUENCE hibernate_sequence
  INCREMENT 1
  MINVALUE 1
  MAXVALUE 9223372036854775807
  START 1
  CACHE 1;
ALTER TABLE hibernate_sequence
```

```
19   OWNER TO k19;
```

Código SQL 2.2: Instrução de criação da tabela 'pessoa' no PostgreSQL

Repare que a implementação do JPA adotou estratégias diferentes para a geração automática dos valores para a coluna **id** no MySQL e no PostgreSQL. No caso do MySQL foi utilizado o atributo **AUTO_INCREMENT** na coluna **id** enquanto que no PostgreSQL o recurso de sequências foi utilizado.

A anotação **@GeneratedValue** aceita os seguintes valores para o parâmetro **strategy**:

- **GenerationType.AUTO**: a implementação do JPA decide a estratégia mais apropriada de geração dos valores de acordo com o banco de dados utilizado.
- **GenerationType.IDENTITY**: mapeia o atributo para uma *coluna identidade* do banco de dados utilizado para gerar valores únicos.
- **GenerationType.SEQUENCE**: utiliza o recurso de sequências do banco de dados utilizado para gerar valores únicos.
- **GenerationType.TABLE**: utiliza uma tabela auxiliar para gerar valores únicos.

Importante
Para que sua aplicação seja portável para diferentes bancos de dados, a anotação **@GeneratedValue** deve ser utilizada apenas em um único atributo ou propriedade em conjunto com a anotação **@Id**.

Importante
As estratégias **GenerationType.IDENTITY** e **GenerationType.SEQUENCE** não são consideradas portáveis, pois esses recursos não estão disponíveis em todos os bancos de dados.

Mapeamento Automático

Cada banco possui o seu próprio conjunto de tipos de dados. Para que as informações possam navegar da aplicação para o banco e vice-e-versa, os tipos do Java devem ser mapeados para tipos apropriados do banco de dados.

Alguns tipos do Java são mapeados automaticamente para tipos correspondentes do banco de dados. Eis uma lista dos tipos que são mapeados automaticamente:

- Tipos primitivos (byte, short, char, int, long, float, double e boolean)
- Classes Wrappers (Byte, Short, Character, Integer, Long, Float, Double e Boolean)
- String
- BigInteger e BigDecimal

- java.util.Date e java.util.Calendar
- java.sql.Date, java.sql.Time e java.sql.Timestamp
- Array de byte ou char
- Enums
- Serializables

Esses tipos são chamados de tipos básicos.

Objetos Grandes (LOB)

Eventualmente, dados maiores do que o comum devem ser armazenados no banco de dados. Por exemplo, uma imagem, uma música ou um texto com muitas palavras. Para esses casos, os bancos de dados oferecem tipos de dados específicos. Do ponto de vista do desenvolvedor JPA, basta aplicar a anotação **@LOB** (*Large Objects*) em atributos do tipo `String`, `byte[]`, `Byte[]`, `char[]` ou `Character[]` para que o provedor (Hibernate, EclipseLink ou outra implementação de JPA) utilize os procedimentos adequados para manipular esses dados.

```
1 @Entity
2 class Pessoa {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @Lob
8   private byte[] avatar;
9 }
```

Código Java 2.6: Pessoa.java

Data e Hora

Comumente, as aplicações Java utilizam as classes `java.util.Date` e `java.util.Calendar` para trabalhar com datas e horas. Essas classes são mapeadas automaticamente para tipos adequados no banco de dados. Portanto, basta declarar os atributos utilizando um desses dois tipos nas classes que serão mapeadas para tabelas.

```
1 @Entity
2 class Pessoa {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   private Calendar nascimento;
8 }
```

Código Java 2.7: Pessoa.java

Por padrão, quando aplicamos o tipo `java.util.Date` ou `java.util.Calendar`, tanto a data quanto a hora serão armazenadas no banco de dados. Para mudar esse comportamento, devemos aplicar a anotação **@Temporal** escolhendo uma das três opções abaixo:

TemporalType.DATE: Armazena apenas a data (dia, mês e ano).

TemporalType.TIME: Armazena apenas o horário (hora, minuto e segundo).

TemporalType.TIMESTAMP (Padrão): Armazena a data e o horário.

```
1 @Entity
2 class Pessoa {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @Temporal(TemporalType.DATE)
8   private Calendar nascimento;
9 }
```

Código Java 2.8: Pessoa.java

Dados Transientes

Eventualmente, não desejamos que alguns atributos de um determinado grupo de objetos sejam persistidos no banco de dados. Nesse caso, devemos aplicar o modificador **transient** ou a anotação **@Transient**.

No exemplo abaixo, marcamos o atributo `idade` com a anotação `@Transient` para que essa informação não seja armazenada no banco de dados. A idade de uma pessoa pode ser deduzida a partir de sua data de nascimento, que já está armazenada no banco.

```
1  @Entity
2  class Pessoa {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    @Temporal(TemporalType.DATE)
8    private Calendar nascimento;
9
10   @Transient
11   private int idade;
12 }
```

Código Java 2.9: Pessoa.java

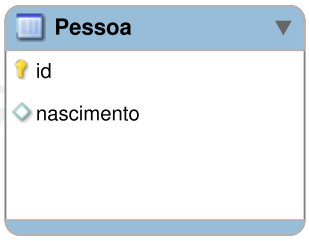


Figura 2.2: Tabela correspondente à classe `Pessoa`. *Note que essa tabela não possui nenhuma coluna associada ao atributo* `idade` *da classe* `Pessoa`

Field Access e Property Access

Os provedores de JPA precisam ter acesso ao estado das entidades para poder administrá-las. Por exemplo, quando persistimos uma instância de uma entidade, o provedor deve "pegar" os dados desse objeto e armazená-los no banco. Quando buscamos uma instância de uma entidade, o provedor recupera as informações correspondentes do banco de dados e "guarda" em um objeto.

O JPA 2 define dois modos de acesso ao estado das instâncias das entidades: **Field Access** e **Property Access**. Quando colocamos as anotações de mapeamento nos atributos, estamos optando pelo modo Field Access. Por outro lado, também podemos colocar essas mesmas anotações nos métodos getters. Nesse caso, estamos optando pelo modo Property Accesss.

No modo Field Access, os atributos dos objetos são acessados diretamente através de **reflection** e não é necessário implementar métodos getters e setters. Nesse modo de acesso, se os métodos getters e setters estiverem implementados, eles não serão utilizados pelo provedor JPA.

No modo Property Access, os métodos getters e setters devem necessariamente ser implementados pelo desenvolvedor. Esses métodos serão utilizados pelo provedor para que ele possa acessar e modificar o estado dos objetos.

Exercícios de Fixação

1 Crie um projeto no Eclipse chamado **K19-Mapeamento**. Copie a pasta **lib** do projeto **K19-JPA2-Hibernate** para o projeto **K19-Mapeamento**. Depois adicione os jars dessa pasta no classpath desse novo projeto.

2 Abra o **MySQL Workbench** e apague a base de dados **K21_mapeamento_bd** se existir. Depois crie a base de dados **K21_mapeamento_bd**.

3 Copie a pasta **META-INF** do projeto **K19-JPA2-Hibernate** para dentro da pasta **src** do projeto **K19-Mapeamento**. Altere o arquivo **persistence.xml** do projeto **K19-Mapeamento**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence
3    version="2.1"
4    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
7      http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd ">
8
9    <persistence-unit name="K21_mapeamento_pu" transaction-type="RESOURCE_LOCAL">
10     <provider>org.hibernate.ejb.HibernatePersistence</provider>
11     <properties>
12       <property
13         name="hibernate.dialect"
14         value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15
16       <property
17         name="hibernate.hbm2ddl.auto"
18         value="update" />
19
```

```
20        <property
21          name="javax.persistence.jdbc.driver"
22          value="com.mysql.jdbc.Driver" />
23
24        <property
25          name="javax.persistence.jdbc.user"
26          value="root" />
27
28        <property
29          name="javax.persistence.jdbc.password"
30          value="root" />
31
32        <property
33          name="javax.persistence.jdbc.url"
34          value="jdbc:mysql://localhost:3306/K21_mapeamento_bd"/>
35      </properties>
36    </persistence-unit>
37  </persistence>
```

Código XML 2.1: persistence.xml

4 Crie uma entidade para modelar os usuários de uma rede social dentro de um pacote chamado `br.com.k19.modelo` no projeto **K19-Mapeamento**.

```
1  @Entity
2  public class Usuario {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    @Column(unique=true)
8    private String email;
9
10   @Temporal(TemporalType.DATE)
11   private Calendar dataDeCadastro;
12
13   @Lob
14   private byte[] foto;
15
16   // GETTERS E SETTERS
17 }
```

Código Java 2.10: Usuario.java

5 Adicione um usuário no banco de dados. Crie uma classe chamada `AdicionaUsuario` em um pacote chamado `br.com.k19.testes` no projeto **K19-Mapeamento**.

```
1  public class AdicionaUsuario {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_mapeamento_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Usuario usuario = new Usuario();
10     usuario.setEmail("contato@k19.com.br");
11     usuario.setDataDeCadastro(Calendar.getInstance());
12
13     manager.persist(usuario);
14
15     manager.getTransaction().commit();
16
```

```
17      manager.close();
18      factory.close();
19    }
20  }
```

Código Java 2.11: AdicionaUsuario.java

Execute a classe `AdicionaUsuario`. Depois, verifique a estrutura da tabela `Usuario` adicionada à base de dados.

6 Abra o MySQL Workbench e observe as propriedades da tabela **Usuario** da base de dados **K21_mapeamento_bd**.

Enums

Por padrão, os tipos enumerados em Java são mapeados para colunas numéricas inteiras no banco de dados. Cada elemento de um Enum é associado a um número inteiro. Essa associação é baseada na ordem em que os elementos do Enum são declarados. O primeiro elemento será associado ao valor 0, o segundo será associado ao valor 1 e assim por diante. Considere o exemplo a seguir.

```
1  public enum Periodo {
2    MATUTINO,
3    NOTURNO
4  }
```

Código Java 2.12: Periodo.java

```
1  @Entity
2  public class Turma {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    private Periodo periodo;
8  }
```

Código Java 2.13: Turma.java

O Enum `Periodo` possui dois elementos: `MATUTINO` e `NOTURNO`. O elemento `MATUTINO` será associado ao valor 0 e o elemento `NOTURNO` será associado ao valor 1.

A tabela correspondente à classe `Turma` possuirá um campo chamado `periodo`. Nos registros correspondentes às turmas de período matutino, esse campo possuirá o valor 0. Já nos registros correspondentes às turmas de período noturno, esse campo possuirá o valor 1.

Imagine que um novo período é adicionado, digamos, o período vespertino. Nesse caso, o Enum `Periodo` poderia vir a ser:

```
1  public enum Periodo {
2    MATUTINO,
3    VESPERTINO,
4    NOTURNO
5  }
```

Código Java 2.14: Periodo.java

Os valores já armazenados no banco de dados poderiam estar incorretos. Por exemplo, antes dessa modificação, o campo `periodo` das turmas noturnas deveria armazenar o valor 1. Após essa modificação, o valor correto passa a ser 2. Assim, os valores do campo `periodo` da tabela `Turma` devem ser atualizados de acordo. No entanto, essa atualização não é automática, e deve ser feita manualmente.

Para evitar esse problema, podemos fazer com que os elementos de um Enum sejam associados a uma string ao invés de um número inteiro. Isso pode ser feito com o uso da anotação `@Enumerated`. Observe o exemplo abaixo.

```
1 @Entity
2 public class Turma {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @Enumerated(EnumType.STRING)
8   private Periodo periodo;
9 }
```

Código Java 2.15: Turma.java

Nesse exemplo, os elementos `MATUTINO`, `VESPERTINO` e `NOTURNO` do Enum `Periodo` serão associados às strings `"MATUTINO"`, `"VESPERTINO"` e `"NOTURNO"`, respectivamente.

Coleções

Considere um sistema que controla o cadastro dos funcionários de uma empresa. Esses funcionários são modelados pela seguinte classe.

```
1 @Entity
2 public class Funcionario {
3
4   @Id @GeneratedValue
5   private Long id;
6
7   private String nome;
8 }
```

Código Java 2.16: Funcionario.java

Devemos também registrar os telefones de contato dos funcionários, sendo que cada funcionário pode ter um ou mais telefones. Em Java, seria razoável utilizar coleções para armazenar os telefones dos funcionários. Veja o exemplo abaixo.

```
1 @Entity
2 public class Funcionario {
3
4   @Id @GeneratedValue
5   private Long id;
6
7   private String nome;
8
```

```
9   @ElementCollection
10  private Collection<String> telefones;
11 }
```

Código Java 2.17: Funcionario.java

A anotação `@ElementCollection` deve ser utilizada para que o mapeamento seja realizado. Nesse exemplo, o banco de dados possuiria uma tabela chamada `Funcionario_telefones` contendo duas colunas. Uma coluna seria usada para armazenar os identificadores dos funcionários e a outra para os telefones. Veja uma ilustração das tabelas do banco de dados na figura abaixo.

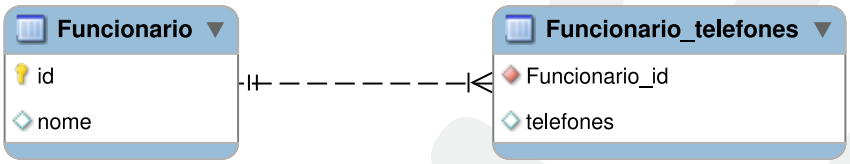


Figura 2.3: Tabelas correspondentes à classe `Funcionario` *e ao atributo* `telefones`

A tabela criada para guardar os telefones dos funcionários também pode ter o seu nome personalizado, assim como os nomes de suas colunas. Para isso, devemos aplicar as anotações `@CollectionTable` e `@Column`.

```
1  @Entity
2  public class Funcionario {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @ElementCollection
10   @CollectionTable(
11     name="Telefones_dos_Funcionarios",
12     joinColumns=@JoinColumn(name="func_id"))
13   @Column(name="telefone")
14   private Collection<String> telefones;
15 }
```

Código Java 2.18: Funcionario.java

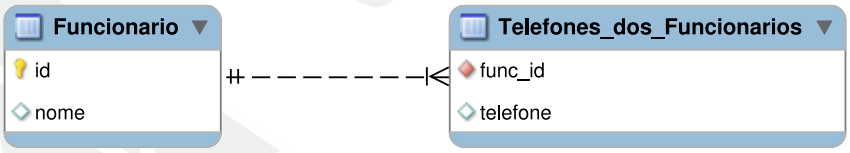
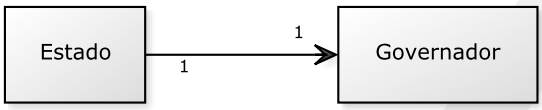


Figura 2.4: Personalizando os nomes da tabela e das colunas

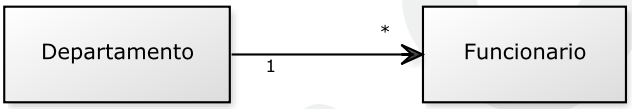
Relacionamentos

Os relacionamentos entre as entidades de um domínio devem ser expressos na modelagem através de vínculos entre classes. De acordo com a JPA, podemos definir quatro tipos de relacionamentos de acordo com a cardinalidade.

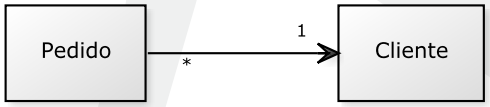
One to One (Um para Um): Por exemplo, um estado é governado por apenas um governador e um governador governa apenas um estado.



One to Many (Um para Muitos): Por exemplo, um departamento possui muitos funcionários e um funcionário trabalha em apenas em um departamento.



Many to One (Muitos para Um): Por exemplo, um pedido pertence a apenas um cliente e um cliente faz muitos pedidos.



Many to Many (Muitos para Muitos): Por exemplo, um livro possui muitos autores e um autor possui muitos livros.

One to One

Suponha que em nosso domínio existam duas entidades: Estado e Governador. Devemos criar uma classe para cada entidade e aplicar nelas as anotações básicas de mapeamento.

```
1 @Entity
2 class Estado {
3   @Id
4   @GeneratedValue
5   private Long id;
6 }
```

Código Java 2.19: Estado.java

```
1 @Entity
2 class Governador {
3   @Id
4   @GeneratedValue
5   private Long id;
6 }
```

Código Java 2.20: Governador.java

Como existe um relacionamento entre estados e governadores, devemos expressar esse vínculo através de um atributo que pode ser inserido na classe `Estado`.

```
1 @Entity
2 class Estado {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   private Governador governador;
8 }
```

Código Java 2.21: Estado.java

Além disso, devemos informar ao provedor JPA que o relacionamento que existe entre um estado e um governador é do tipo One to One. Fazemos isso aplicando a anotação **@OneToOne** no atributo que expressa o relacionamento.

```
1 @Entity
2 class Estado {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @OneToOne
8   private Governador governador;
9 }
```

Código Java 2.22: Estado.java

No banco de dados, a tabela referente à classe `Estado` possuirá uma coluna de relacionamento chamada de **join column**. Em geral, essa coluna será definida como uma chave estrangeira associada à tabela referente à classe `Governador`.

Por padrão, o nome da coluna de relacionamento é formado pelo nome do atributo que estabelece o relacionamento, seguido pelo caractere "_" e pelo nome do atributo que define a chave primária da entidade alvo. No exemplo de estados e governadores, a join column teria o nome **governador_id**.

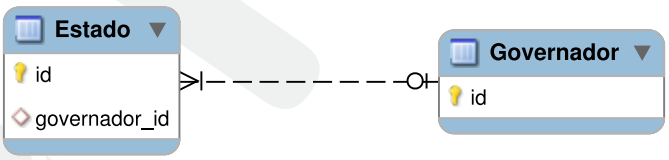


Figura 2.5: Tabelas correspondentes às classes `Estado` e `Governador`

Podemos alterar o nome padrão das join columns aplicando a anotação **@JoinColumn**, conforme apresentado no exemplo abaixo.

```
1 @Entity
2 class Estado {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @OneToOne
8   @JoinColumn(name="gov_id")
9   private Governador governador;
```

```
10 }
```

Código Java 2.23: Estado.java

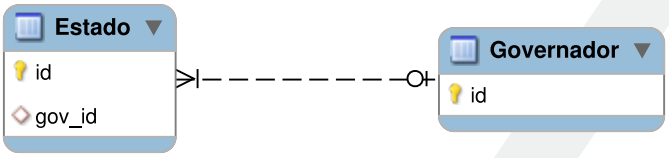


Figura 2.6: Personalizando o nome da coluna de relacionamento

Mais Sobre

Por padrão, em um relacionamento One to One, um objeto da primeira entidade não precisa estar necessariamente relacionado a um objeto da segunda entidade. Para exigir que cada objeto da primeira entidade esteja relacionado a um objeto da segunda entidade, devemos usar o atributo **optional** da anotação `OneToOne`.

```
1 @Entity
2 class Estado {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @OneToOne(optional=false)
8   private Governador governador;
9 }
```

Código Java 2.24: Estado.java

Exercícios de Fixação

7 Implemente duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**: `Estado` e `Governador`.

```
1  @Entity
2  public class Governador {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    // GETTERS E SETTERS
10 }
```

Código Java 2.25: Governador.java

```
1  @Entity
2  public class Estado {
3    @Id
4    @GeneratedValue
5    private Long id;
6
```

```
7   private String nome;
8
9   @OneToOne
10  private Governador governador;
11
12  // GETTERS E SETTERS
13 }
```

Código Java 2.26: Estado.java

8 Adicione um governador e um estado no banco de dados. Crie uma classe chamada `AdicionaEstadoGovernador` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento**.

```
1  public class AdicionaEstadoGovernador {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_mapeamento_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Governador g = new Governador();
10     g.setNome("Rafael Cosentino");
11
12     Estado e = new Estado();
13     e.setNome("São Paulo");
14     e.setGovernador(g);
15
16     manager.persist(g);
17     manager.persist(e);
18
19     manager.getTransaction().commit();
20
21     manager.close();
22     factory.close();
23   }
24 }
```

Código Java 2.27: AdicionaEstadoGovernador.java

Execute a classe `AdicionaEstadoGovernador`.

9 Abra o MySQL Workbench e observe as propriedades das tabelas `Estado` e `Governador` da base de dados **K21_mapeamento_bd**.

One to Many

Suponha que em nosso domínio existam as entidades Departamento e Funcionário. Criaríamos duas classes com as anotações básicas de mapeamento.

```
1 @Entity
2 class Departamento {
3   @Id
4   @GeneratedValue
5   private Long id;
6 }
```

Código Java 2.28: Departamento.java

```
@Entity
class Funcionario {
  @Id
  @GeneratedValue
  private Long id;
}
```

Código Java 2.29: Funcionario.java

Como existe um relacionamento entre departamentos e funcionários, devemos expressar esse vínculo através de um atributo que pode ser inserido na classe `Departamento`. Supondo que um departamento possa ter muitos funcionários, devemos utilizar uma coleção para expressar esse relacionamento.

```
@Entity
class Departamento {
  @Id
  @GeneratedValue
  private Long id;

  private Collection<Funcionario> funcionarios;
}
```

Código Java 2.30: Departamento.java

Para informar a cardinalidade do relacionamento entre departamentos e funcionários, devemos utilizar a anotação **@OneToMany** na coleção.

```
@Entity
class Departamento {
  @Id
  @GeneratedValue
  private Long id;

  @OneToMany
  private Collection<Funcionario> funcionarios;
}
```

Código Java 2.31: Departamento.java

No banco de dados, além das duas tabelas correspondentes às classes `Departamento` e `Funcionario`, deve existir uma terceira tabela para relacionar os registros dos departamentos com os registros dos funcionários. Essa terceira tabela é chamada de tabela de relacionamento ou **join table**.

Por padrão, o nome da join table é a concatenação com "_" dos nomes das duas entidades. No exemplo de departamentos e funcionários, o nome do join table seria **Departamento_Funcionario**. Essa tabela possuirá duas colunas vinculadas às entidades que formam o relacionamento. No exemplo, a join table `Departamento_Funcionario` possuirá uma coluna chamada **Departamento_id** e outra chamada **funcionarios_id**.

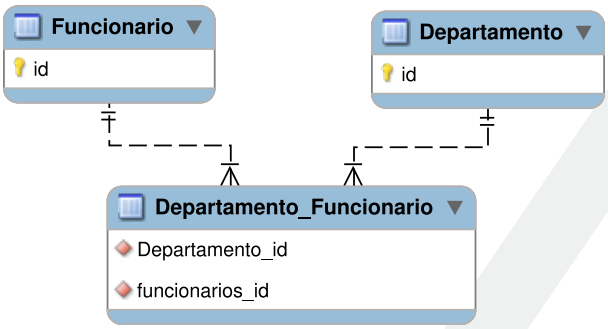


Figura 2.7: Tabelas correspondentes às classes `Departamento` *e* `Funcionario`

Para personalizar os nomes das colunas da join table e da própria join table, podemos aplicar a anotação **@JoinTable** no atributo que define o relacionamento.

```
1  @Entity
2  class Departamento {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    @OneToMany
8    @JoinTable(name="DEP_FUNC",
9      joinColumns=@JoinColumn(name="DEP_ID"),
10     inverseJoinColumns=@JoinColumn(name="FUNC_ID"))
11   private Collection<Funcionario> funcionarios;
12 }
```

Código Java 2.32: Departamento.java

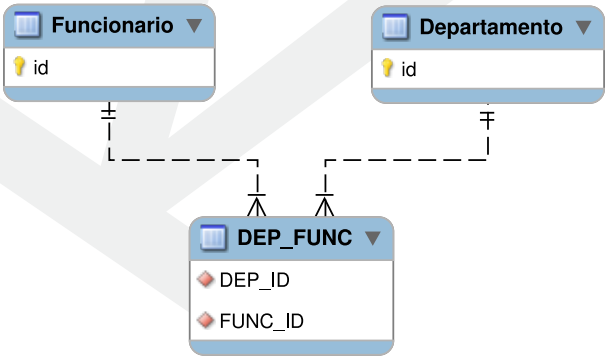


Figura 2.8: Personalizando a tabela de relacionamento

Exercícios de Fixação

10 Implemente duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**: `Funcionario` e `Departamento`.

```
1  @Entity
2  public class Funcionario {
3    @Id
4    @GeneratedValue
```

```
5   private Long id;
6
7   private String nome;
8
9   // GETTERS E SETTERS
10 }
```

Código Java 2.33: Funcionario.java

```
1  @Entity
2  public class Departamento {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @OneToMany
10   private Collection<Funcionario> funcionarios = new ArrayList<Funcionario>();
11
12   // GETTERS E SETTERS
13 }
```

Código Java 2.34: Departamento.java

11 Adicione um departamento e um funcionário no banco de dados. Crie uma classe chamada `AdicionaDepartamentoFuncionario` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento**.

```
1  public class AdicionaDepartamentoFuncionario {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_mapeamento_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Funcionario f = new Funcionario();
10     f.setNome("Rafael Cosentino");
11
12     Departamento d = new Departamento();
13     d.setNome("Financeiro");
14     d.getFuncionarios().add(f);
15
16     manager.persist(f);
17     manager.persist(d);
18
19     manager.getTransaction().commit();
20
21     manager.close();
22     factory.close();
23   }
24 }
```

Código Java 2.35: AdicionaDepartamentoFuncionario.java

Execute a classe `AdicionaDepartamentoFuncionario`.

12 Abra o MySQL Workbench e observe as propriedades das tabelas `Departamento`, `Funcionario` e `Departamento_Funcionario` da base de dados **K21_mapeamento_bd**.

Many to One

Suponha que em nosso domínio existam as entidades Pedido e Cliente. As duas classes que modelariam essas entidades seriam definidas com as anotações principais de mapeamento.

```
1 @Entity
2 class Pedido {
3   @Id
4   @GeneratedValue
5   private Long id;
6 }
```

Código Java 2.36: Pedido.java

```
1 @Entity
2 class Cliente {
3   @Id
4   @GeneratedValue
5   private Long id;
6 }
```

Código Java 2.37: Cliente.java

Como existe um relacionamento entre pedidos e clientes, devemos expressar esse vínculo através de um atributo que pode ser inserido na classe `Pedido`. Supondo que um pedido pertença a um único cliente, devemos utilizar um atributo simples para expressar esse relacionamento.

```
1 @Entity
2 class Pedido {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   private Cliente cliente;
8 }
```

Código Java 2.38: Pedido.java

Para informar a cardinalidade do relacionamento entre pedidos e clientes, devemos utilizar a anotação **@ManyToOne** no atributo.

```
1 @Entity
2 class Pedido {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @ManyToOne
8   private Cliente cliente;
9 }
```

Código Java 2.39: Pedido.java

No banco de dados, a tabela referente à classe `Pedido` possuirá uma **join column** vinculada à tabela da classe `Cliente`. Por padrão, o nome da join column é formado pelo nome da entidade alvo do relacionamento, seguido pelo caractere "_" e pelo nome do atributo que define a chave primária da entidade alvo.

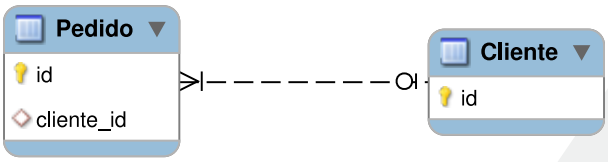


Figura 2.9: Tabelas correspondentes às classes `Pedido` *e* `Cliente`

No exemplo de pedidos e clientes, o nome da join column seria **cliente_id**. Podemos alterar o nome padrão das join columns aplicando a anotação **@JoinColumn**.

```
1  @Entity
2  class Pedido {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    @ManyToOne
8    @JoinColumn(name="cli_id")
9    private Cliente cliente;
10 }
```

Código Java 2.40: Pedido.java

Figura 2.10: Personalizando a tabela `Pedido`

Mais Sobre

Por padrão, em um relacionamento Many to One, um objeto da primeira entidade não precisa estar necessariamente relacionado a um objeto da segunda entidade. Para exigir que cada objeto da primeira entidade esteja relacionado a um objeto da segunda entidade, devemos usar o atributo **optional** da anotação `ManyToOne`.

```
1  @Entity
2  class Pedido {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    @ManyToOne(optional=false)
8    private Cliente cliente;
9  }
```

Código Java 2.41: Pedido.java

Exercícios de Fixação

13 Implemente duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**: `Pedido` e `Cliente`.

```
@Entity
public class Cliente {
  @Id
  @GeneratedValue
  private Long id;

  private String nome;

  // GETTERS E SETTERS
}
```

Código Java 2.42: Cliente.java

```
@Entity
public class Pedido {
  @Id
  @GeneratedValue
  private Long id;

  @Temporal(TemporalType.DATE)
  private Calendar data;

  @ManyToOne
  private Cliente cliente;

  // GETTERS E SETTERS
}
```

Código Java 2.43: Pedido.java

14 Adicione um cliente e um departamento no banco de dados. Crie uma classe chamada `AdicionaPedidoCliente` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento**.

```
public class AdicionaPedidoCliente {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_mapeamento_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    Cliente c = new Cliente();
    c.setNome("Rafael Cosentino");

    Pedido p = new Pedido();
    p.setData(Calendar.getInstance());
    p.setCliente(c);

    manager.persist(c);
    manager.persist(p);

    manager.getTransaction().commit();

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

Código Java 2.44: AdicionaPedidoCliente.java

Execute a classe `AdicionaPedidoCliente`.

15 Abra o MySQL Workbench e observe as propriedades das tabelas `Cliente` e `Pedido` da base de dados **K21_mapeamento_bd**.

Many to Many

Suponha que em nosso domínio existam as entidades Livro e Autor. As classes com as anotações básicas de mapeamento seriam mais ou menos assim:

```
1 @Entity
2 class Livro {
3   @Id
4   @GeneratedValue
5   private Long id;
6 }
```

Código Java 2.45: Livro.java

```
1 @Entity
2 class Autor {
3   @Id
4   @GeneratedValue
5   private Long id;
6 }
```

Código Java 2.46: Autor.java

Como existe um relacionamento entre livros e autores, devemos expressar esse vínculo através de um atributo que pode ser inserido na classe `Livro`. Supondo que um livro possa ser escrito por muitos autores, devemos utilizar uma coleção para expressar esse relacionamento.

```
1 @Entity
2 class Livro {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   private Collection<Autor> autores;
8 }
```

Código Java 2.47: Livro.java

Para informar a cardinalidade do relacionamento entre livros e autores, devemos utilizar a anotação **@ManyToMany** na coleção.

```
1 @Entity
2 class Livro {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @ManyToMany
8   private Collection<Autor> autores;
9 }
```

Código Java 2.48: Livro.java

No banco de dados, além das duas tabelas correspondentes às classes `Livro` e `Autor`, uma join table é criada para relacionar os registros dos livros com os registros dos autores. Por padrão, o nome

da join table é a concatenação com "_" dos nomes das duas entidades. No exemplo de livros e autores, o nome do join table seria **Livro_Autor**. Essa tabela possuirá duas colunas vinculadas às entidades que formam o relacionamento. No exemplo, a join table `Livro_Autor` possuirá uma coluna chamada **Livro_id** e outra chamada **autores_id**.

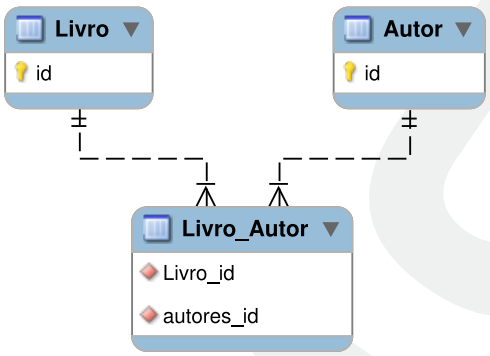


Figura 2.11: Tabelas correspondentes às classes `Livro` *e* `Autor`

Para personalizar o nome join table e os nomes de suas colunas, podemos aplicar a anotação **@JoinTable** no atributo que define o relacionamento.

```
1  @Entity
2  class Livro {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    @ManyToMany
8    @JoinTable(name="Liv_Aut",
9      joinColumns=@JoinColumn(name="Liv_ID"),
10     inverseJoinColumns=@JoinColumn(name="Aut_ID"))
11   private Collection<Autor> autores;
12 }
```

Código Java 2.49: Livro.java

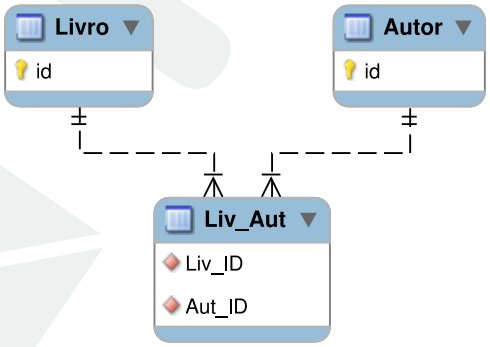


Figura 2.12: Personalizando a tabela de relacionamento

Exercícios de Fixação

16 Implemente duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**: `Livro` e `Autor`.

```
@Entity
public class Autor {
  @Id
  @GeneratedValue
  private Long id;

  private String nome;

  // GETTERS E SETTERS
}
```

Código Java 2.50: Autor.java

```
@Entity
public class Livro {
  @Id
  @GeneratedValue
  private Long id;

  private String nome;

  @ManyToMany
  private Collection<Autor> autores = new ArrayList<Autor>();

  // GETTERS E SETTERS
}
```

Código Java 2.51: Livro.java

17 Adicione um livro e um autor no banco de dados. Crie uma classe chamada `AdicionaLivroAutor` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento**.

```
public class AdicionaLivroAutor {
  public static void main(String[] args) {
    EntityManagerFactory factory =
      Persistence.createEntityManagerFactory("K21_mapeamento_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    Autor a = new Autor();
    a.setNome("Rafael Cosentino");

    Livro l = new Livro();
    l.setNome("JPA2");
    l.getAutores().add(a);

    manager.persist(a);
    manager.persist(l);

    manager.getTransaction().commit();

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

Código Java 2.52: AdicionaLivroAutor.java

Execute a classe `AdicionaLivroAutor`.

18 Abra o MySQL Workbench e observe as propriedades das tabelas `Livro`, `Autor` e `Livro_Autor` da base de dados **K21_mapeamento_bd**.

Relacionamentos Bidirecionais

Quando expressamos um relacionamento colocando um atributo em uma das entidades, podemos acessar a outra entidade a partir da primeira. Por exemplo, considere o relacionamento entre governadores e estados.

```
@Entity
class Estado {
  @Id
  @GeneratedValue
  private Long id;

  @OneToOne
  private Governador governador;

  // GETTERS E SETTERS
}
```

Código Java 2.53: Estado.java

Como o relacionamento está definido na classe `Estado`, podemos acessar o governador a partir de um estado.

```
Estado e = manager.find(Estado.class, 1L);
Governador g = e.getGovernador();
```

Código Java 2.54: Acessando o governador a partir de um estado

Também podemos expressar o relacionamento na classe `Governador`. Dessa forma, poderíamos acessar um estado a partir de um governador.

```
@Entity
class Governador {
  @Id
  @GeneratedValue
  private Long id;

  @OneToOne
  private Estado estado;

  // GETTERS E SETTERS
}
```

Código Java 2.55: Governador.java

```
Governador g = manager.find(Governador.class, 1L);
Estado e = g.getEstado();
```

Código Java 2.56: Acessando um estado a partir de um governador

A figura abaixo ilustra as tabelas `Estado` e `Governador` no banco de dados, assim como as join columns correspondentes aos relacionamentos.

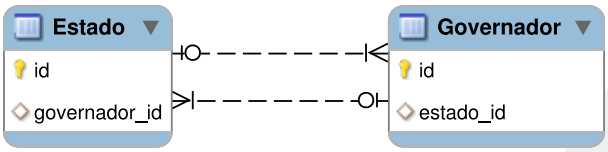


Figura 2.13: Tabelas `Estado` *e* `Governador` *no banco de dados*

Note que foram criadas duas colunas de relacionamentos. A primeira na tabela `Estado` com o nome `governador_id` e a segunda na tabela `Governador` com o nome `estado_id`. Nesse caso, o provedor JPA está considerando dois relacionamentos unidirecionais distintos entre essas entidades.

No entanto, de acordo com o modelo relacional, a relação entre estados e governadores deveria ser expressa com apenas uma coluna de relacionamento. Ou seja, o relacionamento entre governadores e estados deveria ser bidirecional. Assim, devemos indicar em uma das classes que esse relacionamento bidirecional é a junção de dois relacionamentos unidirecionais. Para isso, devemos adicionar o atributo **mappedBy** na anotação `@OneToOne` em uma das classes. O valor do `mappedBy` deve ser o nome do atributo que expressa o mesmo relacionamento na outra entidade.

```
1  @Entity
2  class Governador {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    @OneToOne(mappedBy="governador")
8    private Estado estado;
9
10   // GETTERS E SETTERS
11 }
```

Código Java 2.57: Governador.java

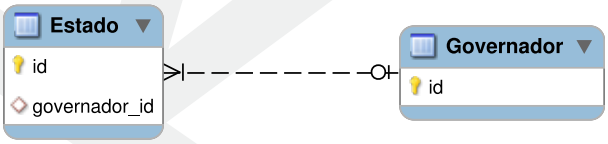


Figura 2.14: Tabelas `Estado` *e* `Governador` *no banco de dados*

Exercícios de Fixação

19 Considere um sistema de cobrança de ligações telefônicas. Nesse sistema, temos uma entidade chamada Ligação e uma entidade chamada Fatura. Cada ligação está associada a uma única fatura, enquanto que uma fatura está associada a múltiplas ligações. Implemente classes para modelar essas duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**.

```
1  @Entity
2  public class Ligacao {
3    @Id @GeneratedValue
4    private Long id;
5
6    @ManyToOne
```

```
7   private Fatura fatura;
8
9   private Integer duracao;
10
11  // GETTERS E SETTERS
12 }
```

Código Java 2.58: Ligacao.java

```
1  @Entity
2  public class Fatura {
3    @Id @GeneratedValue
4    private Long id;
5
6    @OneToMany
7    private Collection<Ligacao> ligacoes = new ArrayList<Ligacao>();
8
9    @Temporal(TemporalType.DATE)
10   private Calendar vencimento;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 2.59: Fatura.java

20 Faça um teste para adicionar algumas ligações e uma fatura. Adicione no pacote `br.com.k19.testes` do projeto **K19-Mapeamento** uma classe chamada `AdicionaFaturaLigacao`.

```
1  public class AdicionaFaturaLigacao {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_mapeamento_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Ligacao ligacao1 = new Ligacao();
10     ligacao1.setDuracao(162);
11
12     Ligacao ligacao2 = new Ligacao();
13     ligacao2.setDuracao(324);
14
15     Fatura fatura = new Fatura();
16     fatura.setVencimento(new GregorianCalendar(2012, 11, 20));
17
18     fatura.getLigacoes().add(ligacao1);
19     fatura.getLigacoes().add(ligacao2);
20
21     ligacao1.setFatura(fatura);
22     ligacao2.setFatura(fatura);
23
24     manager.persist(fatura);
25     manager.persist(ligacao1);
26     manager.persist(ligacao2);
27
28     manager.getTransaction().commit();
29
30     manager.close();
31     factory.close();
32   }
33 }
```

Código Java 2.60: AdicionaFaturaLigacao.java

Execute a classe `AdicionaFaturaLigacao`.

21 Através do MySQL Workbench, verifique as tabelas criadas. Observe que a tabela `Ligacao` possui uma coluna de relacionamento chamada `fatura_id` e a tabela `Fatura_Ligacao` vincula os registros das tabelas `Ligacao` e `Fatura`.

22 Através do MySQL Workbench, apague primeiro a tabela `Fatura_Ligacao` e, em seguida, apague as tabelas `Fatura` e `Ligacao`.

23 Altere o código da classe `Fatura` de forma a criar um relacionamento bidirecional entre as faturas e as ligações.

```
1  @Entity
2  public class Fatura {
3    @Id @GeneratedValue
4    private Long id;
5
6    @OneToMany(mappedBy="fatura")
7    private Collection<Ligacao> ligacoes = new ArrayList<Ligacao>();
8
9    @Temporal(TemporalType.DATE)
10   private Calendar vencimento;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 2.61: Fatura.java

24 Execute a classe `AdicionaFaturaLigacao` para adicionar uma fatura e algumas ligações. Através do MySQL Workbench, verifique as tabelas criadas. Note que foram criadas apenas duas tabelas: `Fatura` e `Ligacao`.

Objetos Embutidos

Suponha que em nosso domínio exista uma entidade chamada `Pessoa`. Toda pessoa possui um endereço, que é formado por país, estado, cidade, logradouro, número, complemento e CEP. Para melhorar a organização da nossa aplicação, podemos criar duas classes: `Pessoa` e `Endereco`.

```
1  @Entity
2  class Pessoa {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @Temporal(TemporalType.DATE)
10   private Calendar nascimento;
11
12   @OneToOne
13   private Endereco endereco;
14 }
```

Código Java 2.62: Pessoa.java

```
1  @Entity
2  class Endereco {
```

```
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   private String pais;
8
9   private String estado;
10
11  private String cidade;
12
13  private String logradouro;
14
15  private int numero;
16
17  private String complemento;
18
19  private int cep;
20 }
```

Código Java 2.63: Endereco.java

Da forma como os mapeamentos estão definidos, duas tabelas serão criadas: uma para a classe `Pessoa` e outra para a classe `Endereco`. Na tabela `Pessoa`, haverá uma coluna de relacionamento.

Para recuperar os dados do endereço de uma pessoa, duas tabelas precisam ser consultadas através de uma operação de join. Esse tipo de operação no banco de dados é custoso.

Suponha que a tabela `Endereco` esteja relacionada apenas com a tabela `Pessoa`. Nesse caso, seria interessante se pudéssemos guardar os endereços das pessoas na própria tabela `Pessoa`, tornando desnecessária a existência da tabela `Endereco`. No entanto, gostaríamos de manter as classes `Pessoa` e `Endereco`.

Isso pode ser feito da seguinte forma. Na classe `Pessoa`, devemos remover a anotação de cardinalidade `@OneToOne`. Na classe `Endereco`, devemos substituir a anotação `@Entity` por **@Embeddable**. Além disso, não devemos definir uma chave para a classe `Endereco`, pois ela não define uma entidade.

```
1  @Entity
2  class Pessoa {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @Temporal(TemporalType.DATE)
10   private Calendar nascimento;
11
12   private Endereco endereco;
13 }
```

Código Java 2.64: Pessoa.java

```
1  @Embeddable
2  class Endereco {
3    private String pais;
4
5    private String estado;
6
7    private String cidade;
8
9    private String logradouro;
10
```

```
11   private int numero;
12
13   private String complemento;
14
15   private int cep;
16 }
```

Código Java 2.65: Endereco.java

Podemos conseguir o mesmo resultado da seguinte forma. Na classe `Pessoa`, devemos substituir a anotação de cardinalidade `@OneToOne` por `@Embedded`. Na classe `Endereco`, devemos remover a anotação `@Entity`. Também, não devemos definir uma chave para a classe `Endereco`, pois ela não define uma entidade.

```
1  @Entity
2  class Pessoa {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @Temporal(TemporalType.DATE)
10   private Calendar nascimento;
11
12   @Embedded
13   private Endereco endereco;
14 }
```

Código Java 2.66: Pessoa.java

```
1  class Endereco {
2    private String pais;
3
4    private String estado;
5
6    private String cidade;
7
8    private String logradouro;
9
10   private int numero;
11
12   private String complemento;
13
14   private int cep;
15 }
```

Código Java 2.67: Endereco.java

Exercícios de Fixação

25 Crie uma classe para modelar endereços no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**.

```
1  public class Endereco {
2
3    private String estado;
4
5    private String cidade;
```

```
6
7     private String logradouro;
8
9     private int numero;
10
11    // GETTERS E SETTERS
12  }
```

Código Java 2.68: Endereco.java

26 No pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**, crie uma classe chamada `Candidato`.

```
1  @Entity
2  public class Candidato {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @Temporal(TemporalType.DATE)
10   private Calendar nascimento;
11
12   @Embedded
13   private Endereco endereco;
14
15   // GETTERS E SETTERS
16 }
```

Código Java 2.69: Candidato.java

27 Crie uma classe chamada `AdicionaCandidatoEndereco` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento** para adicionar alguns candidatos e endereços.

```
1  public class AdicionaCandidatoEndereco {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_mapeamento_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Endereco e = new Endereco();
10     e.setEstado("São Paulo");
11     e.setCidade("São Paulo");
12     e.setLogradouro("Av. Brigadeiro Faria Lima");
13     e.setNumero(1571);
14
15     Candidato p = new Candidato();
16     p.setNome("Rafael Cosentino");
17     p.setNascimento(new GregorianCalendar(1984, 10, 30));
18     p.setEndereco(e);
19
20     manager.persist(p);
21
22     manager.getTransaction().commit();
23
24     manager.close();
25     factory.close();
26   }
27 }
```

Código Java 2.70: AdicionaPessoaEndereco.java

28 Execute a classe `AdicionaCandidatoEndereco` para adicionar um candidato e seu endereço. Através do MySQL Workbench, verifique a tabelas criada. Note que foi criada apenas a tabela `Candidato`.

Herança

O mapeamento objeto-relacional descreve como os conceitos de orientação a objetos são mapeados para os conceitos do modelo relacional. De todos os conceitos de orientação a objetos, um dos mais complexos de se mapear é o de Herança.

A especificação JPA define três estratégias para realizar o mapeamento de herança.

- Single Table
- Joined
- Table Per Class

Single Table

A estratégia Single Table é a mais comum e a que possibilita melhor desempenho em relação a velocidade das consultas. Nessa estratégia, a super classe deve ser anotada com

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE).
```

O provedor JPA criará apenas uma tabela com o nome da super classe para armazenar os dados dos objetos criados a partir da super classe ou das sub classes. Todos os atributos da super classe e os das sub classes serão mapeados para colunas dessa tabela. Além disso, uma coluna especial chamada **DTYPE** será utilizada para identificar a classe do objeto correspondente ao registro.

```
1 @Entity
2 @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
3 public class Pessoa {
4   @Id @GeneratedValue
5   private Long id;
6
7   private String nome;
8 }
```

Código Java 2.71: Pessoa.java

```
1 @Entity
2 public class PessoaJuridica extends Pessoa{
3   private String cnpj;
4 }
```

Código Java 2.72: PessoaJuridica.java

```
1 @Entity
2 public class PessoaFisica extends Pessoa{
3   private String cpf;
4 }
```

Código Java 2.73: PessoaFisica.java

Figura 2.15: Tabela correspondente às classes `Pessoa`, `PessoaJuridica` *e* `PessoaFisica`

A desvantagem da Single Table é o consumo desnecessário de espaço, já que nem todos os campos são utilizados para todos os registros. Por exemplo, se uma pessoa jurídica fosse cadastrada, o campo `cpf` não seria utilizado. Da mesma forma, se uma pessoa física fosse cadastrada, o campo `cnpj` não seria utilizado.

Joined

Nessa estratégia, uma tabela para cada classe da hierarquia é criada. Em cada tabela, apenas os campos referentes aos atributos da classe correspondente são adicionados. Para relacionar os registros das diversas tabelas e remontar os objetos quando uma consulta for realizada, as tabelas relacionadas às sub-classes possuem chaves estrangeiras vinculadas à tabela associada à super-classe.

```
1 @Entity
2 @Inheritance(strategy=InheritanceType.JOINED)
3 public class Pessoa {
4   @Id @GeneratedValue
5   private Long id;
6
7   private String nome;
8 }
```

Código Java 2.74: Pessoa.java

```
1 @Entity
2 public class PessoaJuridica extends Pessoa {
3   private String cnpj;
4 }
```

Código Java 2.75: PessoaJuridica.java

```
1 @Entity
2 public class PessoaFisica extends Pessoa {
3   private String cpf;
4 }
```

Código Java 2.76: PessoaFisica.java

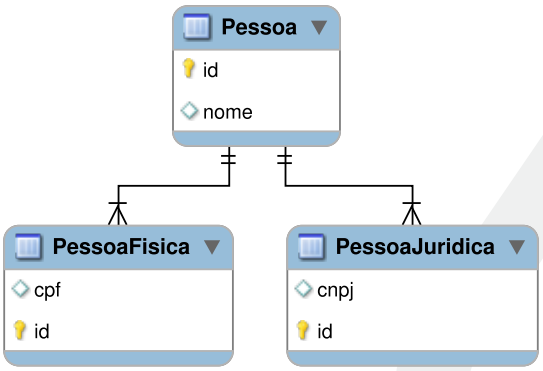


Figura 2.16: Tabelas correspondentes às classes `Pessoa`, `PessoaJuridica` *e* `PessoaFisica`

O consumo de espaço utilizando a estratégia **Joined** é menor do que o utilizado pela estratégia **Single Table**. Contudo, as consultas são mais lentas, pois é necessário realizar operações de **join** para recuperar os dados dos objetos.

Table Per Class

Nessa estratégia, uma tabela para cada classe concreta da hierarquia é criada. Contudo, os dados de um objeto não são colocados em tabelas diferentes. Dessa forma, para remontar um objeto não é necessário realizar operações de **join**. A desvantagem desse modo é que não existe um vínculo explícito no banco de dados entres as tabelas correspondentes às classes da hierarquia.

```
1 @Entity
2 @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
3 public class Pessoa {
4   @Id
5   private Long id;
6
7   private String nome;
8 }
```

Código Java 2.77: Pessoa.java

```
1 @Entity
2 public class PessoaJuridica extends Pessoa {
3   private String cnpj;
4 }
```

Código Java 2.78: PessoaJuridica.java

```
1 @Entity
2 public class PessoaFisica extends Pessoa{
3   private String cpf;
4 }
```

Código Java 2.79: PessoaFisica.java

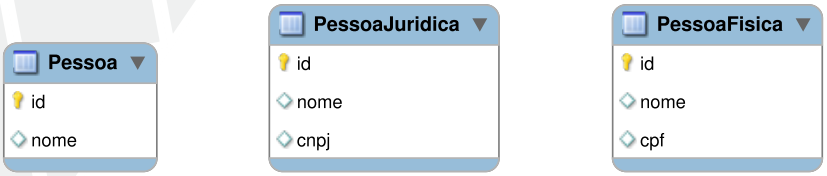


Figura 2.17: Tabelas correspondentes às classes `Pessoa`, `PessoaJuridica` *e* `PessoaFisica`

Na estratégia **Table Per Class**, não podemos utilizar a geração automática de chave primárias simples e numéricas.

Exercícios de Fixação

29 Adicione uma classe chamada `Pessoa` no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Pessoa {
  @Id @GeneratedValue
  private Long id;

  private String nome;

  // GETTERS E SETTERS
}
```

Código Java 2.80: Pessoa.java

30 Faça a classe `PessoaJuridica` no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**.

```
@Entity
public class PessoaJuridica extends Pessoa {
  private String cnpj;

  // GETTERS E SETTERS
}
```

Código Java 2.81: PessoaJuridica.java

31 Faça a classe `PessoaFisica` no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**.

```
@Entity
public class PessoaFisica extends Pessoa {
  private String cpf;

  // GETTERS E SETTERS
}
```

Código Java 2.82: PessoaFisica.java

32 Faça um teste para adicionar pessoas. Crie uma classe chamada `AdicionaPessoa` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento**.

```
public class AdicionaPessoa {
  public static void main(String[] args) {
    EntityManagerFactory factory =
      Persistence.createEntityManagerFactory("K21_mapeamento_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

```

```
9       Pessoa p1 = new Pessoa();
10      p1.setNome("Marcelo");
11
12      PessoaFisica p2 = new PessoaFisica();
13      p2.setNome("Rafael");
14      p2.setCpf("1234");
15
16      PessoaJuridica p3 = new PessoaJuridica();
17      p3.setNome("K19");
18      p3.setCnpj("567788");
19
20      manager.persist(p1);
21      manager.persist(p2);
22      manager.persist(p3);
23
24      manager.getTransaction().commit();
25
26      manager.close();
27      factory.close();
28    }
29  }
```

Código Java 2.83: AdicionaPessoa.java

33 Execute a classe `AdicionaPessoa` para adicionar algumas pessoas. Através do MySQL Workbench, verifique a tabela criada. Note que foi criada apenas a tabela `Pessoa`.

Attribute Converter

A conversão dos dados enviados ou recuperados da base de dados se tornou mais flexível na versão 2.1 da especificação JPA. Com a adição dos attribute converters, é possível personalizar essa conversão.

Por exemplo, considere um Enum que define os possíveis estados de uma tarefa.

```
1   public enum TarefaStatus {
2     ABERTA,
3     DESENVOLVIDA,
4     TESTADA,
5     HOMOLOGADA
6   }
```

Código Java 2.84: TarefaStatus.java

Como visto anteriormente, os Enums podem ser mapeados para colunas numéricas ou para colunas que armazenam texto. No primeiro caso, os valores ordinais dos enums são armazenados nos banco de dados. No segundo caso, os nomes dos enums são armazenados.

Contudo, essa conversão pode ser personzalizada através de um attribute converter. Para criar um attribute converter, é necessário implementar a interface `AttributeConverter`. Essa interface deve ser parametrizada com dois tipos. O primeiro tipo poderá ser utilizado para definir atributos nas entidades. O segundo tipo define a tipagem da coluna no banco de dados. As conversões entre esses dois tipos devem ser implementadas nos métodos `convertToDatabaseColumn` e `convertToEntityAttribute`. Além disso, a classe que define o attribute converter deve ser anotada com `@Converter`.

```
import javax.persistence.AttributeConverter;
import javax.persistence.Converter;

@Converter
public class TarefaStatusConverter
  implements AttributeConverter<TarefaStatus, String> {

  @Override
  public String convertToDatabaseColumn(TarefaStatus attribute) {
    switch (attribute) {
      case ABERTA:
        return "A";
      case DESENVOLVIDA:
        return "D";
      case TESTADA:
        return "T";
      case HOMOLOGADA:
        return "H";
      default:
        throw new IllegalArgumentException("Valor desconhecido: " + attribute);
    }
  }

  @Override
  public TarefaStatus convertToEntityAttribute(String dbData) {
    switch (dbData) {
      case "A":
        return ABERTA;
      case "D":
        return DESENVOLVIDA;
      case "T":
        return TESTADA;
      case "H":
        return HOMOLOGADA;
      default:
        throw new IllegalArgumentException("Valor desconhecido: " + dbData);
    }
  }
}
```

Código Java 2.85: TarefaStatusConverter.java

Há duas formas de aplicar o conversor acima. A primeira é definir o atributo `autoApply` da anotação `@Converter` com o valor `true`. Nesse caso, o provedor JPA utilizará o conversor sempre que encontrar um atributo do tipo passado no primeiro parâmetro da interface `AttributeConverter`.

```
@Converter(autoApply = true)
```

A segunda forma é anotar os atributos que devem ser convertidos pelo conversor com a anotação `@Convert`.

```
@Entity
public class Tarefa {
  @Id
  @GeneratedValue
  private Long id;

  private String titulo;

  @Lob
  private String descricao;

  @Convert(converter = TarefaStatusConverter.class)
  private TarefaStatus tarefaStatus;

  // GETTERS E SETTERS
```

```
16 }
```

Código Java 2.87: Tarefa.java

Exercícios de Fixação

34 Crie uma classe para modelar reservas no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento**.

```
1  @Entity
2  public class Reserva {
3    @Id
4    @GeneratedValue
5    private Long id;
6
7    private LocalDate inicio;
8
9    private LocalDate termino;
10
11   // GETTERS E SETTERS
12 }
```

Código Java 2.88: Reserva.java

35 Adicione um pacote chamado **br.com.k19.conversores** no projeto **K19-Mapeamento**. Depois, crie uma classe chamada `LocalDateConverter` nesse pacote.

```
1  @Converter(autoApply = true)
2  public class LocalDateConverter
3    implements AttributeConverter<LocalDate, Date> {
4
5    @Override
6    public Date convertToDatabaseColumn(LocalDate attribute) {
7      return Date.from(attribute.atStartOfDay().atZone(
8        ZoneId.systemDefault()).toInstant());
9    }
10
11   @Override
12   public LocalDate convertToEntityAttribute(Date dbData) {
13     return LocalDateTime.ofInstant(dbData.toInstant(),
14       ZoneId.systemDefault()).toLocalDate();
15   }
16 }
```

Código Java 2.89: LocalDateConverter.java

36 Crie uma classe chamada `AdicionaReserva` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento** para adicionar algumas reservas.

```
1  public class AdicionaReserva {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_mapeamento_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
```

```
8
9       Reserva reserva = new Reserva();
10      reserva.setInicio(LocalDate.of(2015, Month.AUGUST, 3));
11      reserva.setTermino(LocalDate.of(2015, Month.AUGUST, 14));
12
13      manager.persist(reserva);
14
15      manager.getTransaction().commit();
16
17      manager.close();
18      factory.close();
19    }
20  }
```

Código Java 2.90: AdicionaPessoaEndereco.java

37 Execute a classe `AdicionaReserva` para adicionar uma reserva. Através do MySQL Workbench, verifique a tabelas criada.

ENTITY MANAGER

CAPÍTULO 3

Segundo a especificação JPA, as instâncias das entidades são administradas pelos Entity Managers. As duas principais responsabilidades dos Entity Managers são gerenciar o estado dos objetos e sincronizar os dados da aplicação e do banco de dados.

Estados

É necessário conhecer o ciclo de vida das entidades para saber como os objetos são administrados pelos Entity Managers. Uma instância de uma entidade pode passar pelos seguintes estados:

Novo (New): Um objeto nesse estado não possui uma identidade (chave) e não está associado a um Entity Manager. O conteúdo desse objeto não é enviado para o banco de dados. Toda instância de uma entidade que acabou de ser criada com o comando `new` encontra-se no estado new do JPA.

Administrado (Managed): Um objeto no estado managed possui uma identidade e está associado a um Entity Manager. A cada sincronização, os dados de um objeto no estado managed são atualizados no banco de dados.

Desvinculado (Detached): Um objeto no estado detached possui uma identidade, mas não está associado a um Entity Manager. Dessa forma, o conteúdo desse objeto não é sincronizado com o banco de dados.

Removido (Removed): Um objeto no estado removed possui uma identidade e está associado a um Entity Manager. O conteúdo desse objeto será removido do banco de dados quando houver uma sincronização.

Sincronização com o Banco de Dados

Uma sincronização consiste em propagar para o banco de dados as modificações, remoções e inserções de entidades realizadas em memória através de um Entity Manager.

Quando houver uma sincronização, as modificações realizadas no estado dos objetos managed são propagadas para o banco de dados, assim como os registros referentes aos objetos em estado removed são apagados do banco de dados. De acordo com a especificação, uma sincronização só pode ocorrer se uma transação estiver ativa.

Cada Entity Manager possui uma única transação associada. Para recuperar a transação associada a um Entity Manager, utilizamos o método `getTransaction()`. Uma vez que a transação foi recuperada, podemos ativá-la através do método `begin()`.

Para confirmar uma transação, devemos usar o método `commit()`. Quando esse método é invocado, ocorre uma sincronização com o banco de dados e a transação é finalizada.

```
1 manager.getTransaction().begin();
2 ...
3 manager.getTransaction().commit();
```

Código Java 3.1: Iniciando e confirmando uma transação

Com uma transação ativa, também podemos disparar uma sincronização através do método `flush()`. Apesar dos dados serem enviados para o banco de dados, eles não ficarão visíveis para outras transações. Esses dados serão considerados apenas nas consultas efetuadas dentro da própria transação. Diversas chamadas ao método `flush()` podem ser efetuadas dentro de uma mesma transação.

```
1 manager.getTransaction().begin();
2 ...
3 manager.flush();
4 ...
5 manager.getTransaction().commit();
```

Código Java 3.2: Sincronizações parciais através do método `flush()`

Toda modificação, remoção ou inserção realizada no banco de dados devido às chamadas ao método `flush()` podem ser desfeitas através do método `rollback()`. Uma chamada a esse método também finaliza a transação.

```
1 manager.getTransaction().begin();
2 ...
3 manager.flush();
4 ...
5 manager.getTransaction().rollback();
```

Código Java 3.3: Sincronizações parciais através do método `flush()`

Flush Mode

Há duas políticas adotadas pelos provedores JPA em relação às sincronizações: `FlushModeType.AUTO` (padrão) e `FlushModeType.COMMIT`. No modo AUTO, o provedor JPA realiza sincronizações automáticas antes de uma operação de consulta para garantir que as modificações, remoções e inserções ainda não sincronizadas sejam consideradas na consulta. Já o comportamento no modo COMMIT não está especificado. Consequentemente, cada provedor pode implementar o comportamento que achar mais adequado.

Podemos configurar o flush mode no nível de um Entity Manager afetando o comportamento em todas as consultas realizadas através desse Entity Manager ou configurar apenas para uma consulta.

```
1 manager.setFlushMode(FlushModeType.COMMIT);
```

Código Java 3.4: Configurando o flush mode de um Entity Manager

```
1 query.setFlushMode(FlushModeType.COMMIT);
2 query.setFlushMode(FlushModeType.AUTO);
```

Código Java 3.5: Configurando o flush mode de uma consulta

Transições

Uma instância de uma entidade pode mudar de estado. Veremos a seguir as principais transições.

New → Managed

Um objeto no estado new passa para o estado managed quando utilizamos o método **persist()** dos Entity Managers.

```
1  @Entity
2  class Pessoa {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    // GETTERS E SETTERS
10 }
```

Código Java 3.6: Pessoa.java

```
1  manager.getTransaction().begin();
2
3  Pessoa p = new Pessoa();
4  p.setNome("Rafael Cosentino");
5  manager.persist();
6
7  manager.getTransaction().commit();
```

Código Java 3.7: Persistindo uma instância de uma entidade

BD → Managed

Quando dados são recuperados do banco de dados, o provedor JPA cria objetos para armazenar essas informações. Esses objetos estarão no estado managed.

```
1  Pessoa p = manager.find(Pessoa.class, 1L);
```

```
1  Pessoa p = manager.getReference(Pessoa.class, 1L);
```

```
1  Query query = manager.createQuery("select p from Pessoa p");
2  List<Pessoa> lista = query.getResultList();
```

Managed → Detached

Quando não queremos mais que um objeto no estado managed seja administrado, podemos desvinculá-lo do seu Entity Manager tornando-o detached. Dessa forma, o conteúdo desse objeto não será mais sincronizado com o banco de dados.

Para tornar apenas um objeto detached, devemos utilizar o método **detach()**:

```
1  Pessoa p = manager.find(Pessoa.class, 1L);
2  manager.detach(p);
```

Para tornar detached todos os objetos administrados por um Entity Manager, devemos utilizar o método **clear()**.

```
1 manager.clear();
```

Na chamada do método **close()**, todos os objetos administrados por um Entity Manager também passam para o estado detached.

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

Detached → Managed

O estado de um objeto detached pode ser propagado para um objeto managed com a mesma identidade para que os dados sejam sincronizados com o banco de dados. Esse processo é realizado pelo método **merge()**.

```
1 Pessoa pessoaManaged = manager.merge(pessoaDetached);
```

Managed → Removed

Quando um objeto managed se torna detached, os dados correspondentes a esse objeto não são apagados do banco de dados. Agora, quando utilizamos o método **remove()**, marcamos um objeto para ser removido do banco de dados.

```
1 Pessoa p = manager.find(Pessoa.class, 1L);
2 manager.remove(p);
```

O conteúdo do objeto será removido no banco de dados quando o provedor realizar uma sincronização.

Managed → Managed

O conteúdo de um objeto no estado managed pode ficar desatualizado em relação ao banco de dados se alguém ou alguma aplicação alterar os dados na base de dados. Para atualizar um objeto managed com os dados do banco de dados, devemos utilizar o método **refresh()**.

```
1 Pessoa p = manager.find(Pessoa.class, 1L);
2 manager.refresh(p);
```

Exercícios de Fixação

1 Crie um projeto no eclipse chamado **K19-EntityManager**. Copie a pasta **lib** do projeto **K19-JPA2-Hibernate** para o projeto **K19-EnityManager**. Depois adicione os jars dessa pasta no classpath desse novo projeto.

2 Abra o **MySQL Workbench** e apague a base de dados **K21_entity_manager_bd** se ela existir. Depois crie a base de dados **K21_entity_manager_bd**.

3 Copie a pasta `META-INF` do projeto `K19-JPA2-Hibernate` para dentro da pasta `src` do projeto **K19-EntityManager**. Altere o arquivo `persistence.xml` do projeto **K19-EntityManager**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
<?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="K21_entity_manager_pu" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property
        name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5InnoDBDialect" />

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

      <property
        name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver" />

      <property
        name="javax.persistence.jdbc.user"
        value="root" />

      <property
        name="javax.persistence.jdbc.password"
        value="root" />

      <property
        name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/K21_entity_manager_bd"/>
    </properties>
  </persistence-unit>
</persistence>
```

Código XML 3.1: persistence.xml

4 Crie um pacote chamado `br.com.k19.modelo` no projeto **K19-EntityManager** e adicione a seguinte classe:

```
@Entity
public class Pessoa {

  @Id @GeneratedValue
  private Long id;

  private String nome;

  // GETTERS E SETTERS
}
```

Código Java 3.17: Pessoa.java

5 Persista objetos através de um Entity Manager. Crie uma classe chamada `TestePersist` dentro de um pacote chamado `br.com.k19.testes` no projeto **K19-EntityManager**.

```java
public class TestePersist {
  public static void main(String[] args) {
    EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
    EntityManager manager = factory.createEntityManager();

    // ABRINDO A TRASACAO
    manager.getTransaction().begin();

    // OBJETO NO ESTADO NEW
    Pessoa p = new Pessoa();
    p.setNome("Rafael Cosentino");

    // OBJETO NO ESTADO MANAGED
    manager.persist(p);

    // SINCRONIZANDO E CONFIRMANDO A TRANSACAO
    manager.getTransaction().commit();

    System.out.println("Pessoa id: " + p.getId());

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

Código Java 3.18: TestePersist.java

Execute e consulte o banco de dados através do MySQL Workbench!

6 Busque objetos através de um Entity Manager dado a identidade dos objetos. Crie uma classe chamada `TesteFind` dentro de um pacote `br.com.k19.testes` no projeto **K19-EntityManager**.

```java
public class TesteFind {
  public static void main(String[] args) {
    EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
    EntityManager manager = factory.createEntityManager();

    // OBJETO NO ESTADO MANAGED
    Pessoa p = manager.find(Pessoa.class, 1L);
    System.out.println("Id: " + p.getId());
    System.out.println("Nome: " + p.getNome());

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

Código Java 3.19: TesteFind.java

Execute e observe as mensagens no Console!

7 Altere objetos no estado managed e depois faça um sincronização com o banco de dados através de uma chamada ao método `commit()`. Crie uma classe chamada `TesteManaged` dentro de um pacote `br.com.k19.testes` no projeto **K19-EntityManager**.

```java
public class TesteManaged {
  public static void main(String[] args) {
    EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
```

```
5       EntityManager manager = factory.createEntityManager();
6
7       manager.getTransaction().begin();
8
9       // OBJETO NO ESTADO MANAGED
10      Pessoa p = manager.find(Pessoa.class, 1L);
11
12      // ALTERANDO O CONTEUDO DO OBJETO
13      p.setNome("Marcelo Martins");
14
15      // SINCRONIZANDO E CONFIRMANDO A TRANSACAO
16      manager.getTransaction().commit();
17
18      manager.close();
19      factory.close();
20    }
21  }
```

Código Java 3.20: TesteManaged.java

Execute e consulte o banco de dados através do MySQL Workbench!

8 Altere objetos no estado detached e depois faça um sincronização com o banco de dados através de uma chamada ao método `commit()`. Crie uma classe chamada `TesteDetached` dentro de um pacote `br.com.k19.testes` no projeto **K19-EntityManager**.

```
1   public class TesteDetached {
2     public static void main(String[] args) {
3       EntityManagerFactory factory =
4           Persistence.createEntityManagerFactory("K21_entity_manager_pu");
5       EntityManager manager = factory.createEntityManager();
6
7       manager.getTransaction().begin();
8
9       // OBJETO NO ESTADO MANAGED
10      Pessoa p = manager.find(Pessoa.class, 1L);
11
12      // OBJETO NO ESTADO DETACHED
13      manager.detach(p);
14
15      // ALTERANDO O CONTEUDO DO OBJETO
16      p.setNome("Jonas Hirata");
17
18      // SINCRONIZANDO E CONFIRMANDO A TRANSACAO
19      manager.getTransaction().commit();
20
21      manager.close();
22      factory.close();
23    }
24  }
```

Código Java 3.21: TesteDetached.java

Execute e consulte o banco de dados através do MySQL Workbench!

9 Busque um objeto no banco e então desvincule-o através do método `detach()`. Passe esse objeto como parâmetro para o método `merge()` e então altere uma propriedade do objeto devolvido. Faça um sincronização com o banco de dados através de uma chamada ao método `commit()`. Crie uma classe chamada `TesteMerge` dentro de um pacote `br.com.k19.testes` no projeto **K19-EntityManager**.

```java
public class TesteMerge {
  public static void main(String[] args) {
    EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    // OBJETO NO ESTADO MANAGED
    Pessoa p = manager.find(Pessoa.class, 1L);

    // OBJETO NO ESTADO DETACHED
    manager.detach(p);

    // OBJETO p2 NO ESTADO MANAGED
    Pessoa p2 = manager.merge(p);

    // ALTERANDO O CONTEUDO DO OBJETO
    p2.setNome("Jonas Hirata");

    // SINCRONIZANDO E CONFIRMANDO A TRANSACAO
    manager.getTransaction().commit();

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

Código Java 3.22: TesteMerge.java

Execute e consulte o banco de dados através do MySQL Workbench!

10 Busque por objetos no banco de dados e depois remova-os. Faça uma sincronização com o banco de dados através de uma chamada ao método `commit()`. Crie uma classe chamada `TesteRemoved` dentro de um pacote `br.com.k19.testes` no projeto **K19-EntityManager**.

```java
public class TesteRemoved {
  public static void main(String[] args) {
    EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    // OBJETO NO ESTADO MANAGED
    Pessoa p = manager.find(Pessoa.class, 1L);

    // OBJETO NO ESTADO REMOVED
    manager.remove(p);

    // SINCRONIZANDO E CONFIRMANDO A TRANSACAO
    manager.getTransaction().commit();

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

Código Java 3.23: TesteRemoved.java

Execute e consulte o banco de dados através do MySQL Workbench!

LAZY e EAGER

Como os Entity Managers administram as instâncias das entidades, eles são responsáveis pelo carregamento do estado dos objetos. Há dois modos de carregar um objeto com os dados obtidos de um banco de dados: LAZY e EAGER. No modo LAZY, o provedor posterga ao máximo a busca dos dados no banco de dados. Já no modo EAGER, o provedor busca imediatamente os dados no banco de dados.

find() VS getReference()

Tanto o método `find()` quanto o método `getReference()` permitem que a aplicação obtenha instâncias das entidades a partir das identidades dos objetos. A diferença entre eles é que o `find()` tem comportamento EAGER e o `getReference()` tem comportamento LAZY.

No exemplo abaixo, um objeto é buscado no banco de dados através do método `find()`. Dessa forma, os dados do objeto são carregados imediatamente.

```
1 Pessoa p = manager.find(Pessoa.class, 1L);
2 // o objeto já está carregado
```

Código Java 3.24: Buscando um objeto com o método `find()`

No exemplo abaixo, usamos o método `getReference()` para buscar os dados de um objeto no banco de dados. Como o comportamento desse método é LAZY, os dados do objeto são carregados apenas quando o estado desse objeto é acessado pela primeira vez. Por exemplo, na chamada ao método `getNome()`.

```
1 Pessoa p = manager.getReference(Pessoa.class, 1L);
2 // o objeto não está carregado ainda
3 String nome = p.getNome();
4 // agora o objeto está carregado
```

Código Java 3.25: Buscando um objeto com o método `getReference()`

Quando recuperamos o valor da propriedade `nome` através do método `getNome()`, os demais atributos de tipos básicos também são carregados.

Fetch Type - Tipos Básicos

Alguns dados de um determinado objeto podem ser muito grandes e nem sempre necessários. Por exemplo, considere um objeto do tipo Livro, que possua como atributos título, preço e resumo. Título e preço são dados pequenos, enquanto que o resumo de um livro pode ser um dado relativamente grande.

Imagine que o resumo será utilizado pela aplicação em situações bem esporádicas. Assim, seria interessante que o valor desse atributo fosse carregado apenas quando utilizado.

No entanto, não podemos exigir que o provedor comporte-se dessa forma. Mas, podemos indicar ao provedor que esse comportamento é desejável através do atributo `fetch` da anotação `@Basic`.

```
1 @Basic(fetch=FetchType.LAZY)
2 protected String getNome() {
```

```
3   return nome;
4 }
```

Código Java 3.26: Indicando o modo de carregamento desejável

Importante
O modo LAZY para atributos básicos só pode ser aceito pelos provedores se o modo de acesso for Property Access.

Fetch Type - Relacionamentos

Também podemos definir o modo de carregamento que desejamos utilizar para os relacionamentos das entidades. Por exemplo, considere um relacionamento unidirecional entre estados e governadores.

```
1 @Entity
2 class Estado {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @OneToOne
8   private Governador governador;
9 }
```

Código Java 3.27: Estado.java

```
1 @Entity
2 class Governador {
3   @Id
4   @GeneratedValue
5   private Long id;
6 }
```

Código Java 3.28: Governador.java

Por padrão, quando os dados de um estado são recuperados do banco de dados, os dados do governador associado a esse estado também são recuperados. Em outras palavras, o modo de carregamento padrão do atributo que estabelece o relacionamento entre estados e governadores é EAGER. Podemos alterar esse comportamento padrão aplicando a propriedade `fetch` na anotação `@OneToOne`.

```
1 @Entity
2 class Estado {
3   @Id
4   @GeneratedValue
5   private Long id;
6
7   @OneToOne(fetch=FetchType.LAZY)
8   private Governador governador;
9 }
```

Código Java 3.29: Estado.java

O modo de carregamento dos relacionamentos do tipo One To One e Many To One é EAGER por padrão. O modo de carregamento dos relacionamentos do tipo One To Many e Many To Many é, por padrão, LAZY. Lembrando que o modo de carregamento pode ser definido com a propriedade `fetch` das anotações de relacionamento.

```
@OneToOne(fetch=FetchType.LAZY)
@ManyToOne(fetch=FetchType.LAZY)
@OneToMany(fetch=FetchType.EAGER)
@ManyToMany(fetch=FetchType.EAGER)
```

Código Java 3.30: Definindo o modo de carregamento para relacionamentos

Importante

No modo de carregamento LAZY, o provedor JPA posterga ao máximo o carregamento de um objeto. Esse carregamento só poderá ser feito posteriormente se o Entity Manager que administra esse objeto estiver aberto. Caso a aplicação tente acessar o conteúdo não carregado de um objeto após o fechamento do Entity Manager que o administra, o provedor lançará uma exceção.

Exercícios de Fixação

11 Teste o comportamento do método de busca `find()`. Crie um classe chamada `TesteFindEager` no pacote `br.com.k19.testes` no projeto **K19-EntityManager**. Observe que o código abaixo supõe a existência de registro com identificador igual a 1 na tabela correspondente à classe `Pessoa`. Caso esse registro não exista, busque por um que exista, escolhendo um identificador adequado na chamada do método `find()`.

```
public class TesteFindEager {
  public static void main(String[] args) {
    EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
    EntityManager manager = factory.createEntityManager();

    System.out.println("-----------CHAMANDO O FIND------------");
    Pessoa p = manager.find(Pessoa.class, 1L);
    System.out.println("-----------FEZ O SELECT---------------");

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

Código Java 3.31: TesteFindEager.java

Execute e veja a saída!

12 Teste o comportamento do método de busca `getReference()`. Crie um classe chamada `TesteGetReferenceLazy` no pacote `br.com.k19.testes` do projeto **K19-EntityManager**.

```
public class TesteGetReferenceLazy {
  public static void main(String[] args) {
    EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
    EntityManager manager = factory.createEntityManager();

    System.out.println("-----------CHAMANDO O GETREFERENCE----");
    Pessoa p = manager.getReference(Pessoa.class, 1L);
```

```
9       System.out.println("-----------NAO FEZ O SELECT-----------");
10
11      manager.close();
12      factory.close();
13    }
14  }
```

Código Java 3.32: TesteGetReferenceLazy.java

Execute e veja a saída!

13 Teste o problema de Lazy Initialization. Crie um classe chamada `TesteLazyInitialization` no pacote `br.com.k19.testes` do projeto **K19-EntityManager**.

```
1  public class TesteLazyInitialization {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_entity_manager_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      // OBJETO CARREGADO EM MODO LAZY
8      Pessoa p = manager.getReference(Pessoa.class, 1L);
9
10     manager.close();
11     factory.close();
12
13     // TENTA USAR UM DADO DO OBJETO
14     System.out.println(p.getNome());
15   }
16 }
```

Código Java 3.33: TesteLazyInitialization.java

Execute e veja a saída!

14 Crie duas classes para modelar governadores e estados, estabelecendo um relacionamento One to One entre essas entidades. Essas classes devem ser adicionadas no pacote `br.com.k19.modelo` do projeto **K19-EntityManager**.

```
1  @Entity
2  public class Estado {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @OneToOne
10   private Governador governador;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 3.34: Estado.java

```
1  @Entity
2  public class Governador {
3
4    @Id @GeneratedValue
5    private Long id;
6
```

```
7   private String nome;
8
9   @OneToOne(mappedBy="governador")
10  private Estado estado;
11
12  // GETTERS E SETTERS
13 }
```

Código Java 3.35: Governador.java

15 Adicione um governador e um estado. Crie uma classe chamada `AdicionaGovernadorEstado` no pacote `br.com.k19.testes` do projeto **K19-EntityManager**.

```
1  public class AdicionaGovernadorEstado {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_entity_manager_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Governador governador = new Governador();
10     governador.setNome("Rafael Cosentino");
11
12     Estado estado = new Estado();
13     estado.setNome("São Paulo");
14
15     governador.setEstado(estado);
16     estado.setGovernador(governador);
17
18     manager.persist(estado);
19     manager.persist(governador);
20
21     manager.getTransaction().commit();
22
23     manager.close();
24     factory.close();
25   }
26 }
```

Código Java 3.36: AdicionaGovernadorEstado.java

16 Teste o carregamento EAGER no relacionamento One to One entre estados e governadores. Crie uma classe chamada `TesteCarregamentoRelacionamento` no pacote `br.com.k19.testes` do projeto **K19-EntityManager**.

```
1  public class TesteCarregamentoRelacionamento {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence.createEntityManagerFactory("↩
           K21_entity_manager_pu");
4      EntityManager manager = factory.createEntityManager();
5
6      Estado estado = manager.find(Estado.class, 1L);
7    }
8  }
```

Código Java 3.37: TesteCarregamentoRelacionamento.java

Observe a saída no console para verificar o carregamento tanto do estado quanto do governador.

17 Altere a política padrão do carregamento do governador adicionando a propriedade `fetch` na anotação `@OneToOne` na classe `Estado`.

```
@OneToOne(fetch=FetchType.LAZY)
private Governador governador;
```

18 Execute novamente a classe `TesteCarregamentoRelaciomento` e observe a saída do console para verificar que agora somente o estado é carregado.

19 Faça acontecer o problema de Lazy Initialization no exemplo de estados e governadores.

```
public class TesteCarregamentoRelacionamento {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence.createEntityManagerFactory("↩
        K21_entity_manager_pu");
    EntityManager manager = factory.createEntityManager();

    Estado estado = manager.find(Estado.class, 1L);

    manager.close();
    factory.close();

    System.out.println(estado.getGovernador().getNome());
  }
}
```

Código Java 3.39: TesteCarregamentoRelacionamento.java

Caching

Considere uma aplicação de cadastramento de pessoas. No formulário de cadastro de uma pessoa, o usuário deve selecionar o estado e a cidade onde a pessoa nasceu. A lista de cidades fica armazenada em uma tabela do banco de dados. As cidades são modeladas pela seguinte classe:

```
@Entity
public class Cidade {

  @Id @GeneratedValue
  private Long id;

  private String nomeDaCidade;

  private String nomeDoEstado;
}
```

Código Java 3.40: Cidade.java

O conteúdo da tabela de cidades raramente é alterado por motivos óbvios. Constantemente, a aplicação realizará consultas a essa tabela. Tendo em mente que o custo para trazer dados do banco de dados para a memória da aplicação é alto, poderíamos melhorar o desempenho da aplicação se mantivéssemos a lista das cidades em memória.

Em situações como essa, podemos aplicar o conceito de caching. O caching consiste em manter uma cópia dos dados para evitar o constante acesso ao dispositivo de armazenamento de dados.

Os provedores JPA disponibilizam alguns tipos de cache. Veremos o funcionamento de cada um deles a seguir.

Persistence Context ou Cache de Primeiro Nível

Um objeto já carregado por um Entity Manager é mantido no persistence context (cache de primeiro nível). Cada Entity Manager possui o seu próprio persistence context. Se a aplicação buscar um objeto através de um Entity Manager e ele já estiver carregado no Persistence Context correspondente, a busca não será realizado no banco de dados, evitando assim uma operação no banco de dados.

Exercícios de Fixação

20 Verifique o comportamento dos Entity Managers ao buscar duas vezes o mesmo objeto. Crie uma classe chamada `TestePersistenceContext` no pacote `br.com.k19.testes` do projeto **K19-EntityManager**.

```
1  public class TestePersistenceContext {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_entity_manager_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      System.out.println("------------------PRIMEIRO FIND-----------------");
8      Estado estado = manager.find(Estado.class, 1L);
9      System.out.println("------------------SEGUNDO FIND------------------");
10     estado = manager.find(Estado.class, 1L);
11   }
12 }
```

Código Java 3.41: TestePersistenceContext.java

Execute e observe a saída no console para verificar que o provedor só realiza uma busca

Shared Cache ou Cache de Segundo Nível

Diferentemente do cache de primeiro nível, o cache de segundo nível é compartilhado entre todos os Entity Managers da aplicação. Para habilitar o cache de segundo nível em uma aplicação JPA, devemos acrescentar o elemento `<shared-cache-mode>` na unidade de persistência configurada no arquivo `persistence.xml`. Esse elemento pode possuir os seguintes valores:

ALL: Habilita o cache de segunda nível para todas as entidades.

ENABLE_SELECTIVE: Habilita o cache de segunda nível apenas para as entidades anotadas com `@Cacheable(true)`.

DISABLE_SELECTIVE: Habilita o cache de segunda nível para todas as entidades, exceto aquelas anotadas com `@Cacheable(false)`.

NONE: Desabilita o cache de segundo nível para todas as entidades.

UNSPECIFIED: Nesse caso, o comportamento não está definido pela especificação e depende da implementação do provedor JPA.

O Hibernate 4.1.2 suporta a utilização de diversas implementações de caches de segundo nível. É necessário definir qual implementação queremos utilizar. Essa escolha deve ser informada através da propriedade `hibernate.cache.region.factory_class`.

Exercícios de Fixação

21 Entre na pasta **K19-Arquivos/hibernate-release-VERSAO.Final/lib** da Área de Trabalho e copie os jars da pasta **optional/ehcache**, da pasta **required** e da pasta **jpa** para a pasta **lib** do projeto **K19-EntityManager**. Depois adicione esses jars no build path do projeto.

Importante
Você também pode obter esses arquivos através do site da K19: www.k19.com.br/arquivos.

22 Altere o arquivo `persistence.xml` para habilitar a utilização do shared cache no Hibernate.

```
1  <persistence version="2.0"
2    xmlns="http://java.sun.com/xml/ns/persistence"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
6
7    <persistence-unit name="K21_entity_manager_pu" transaction-type="RESOURCE_LOCAL">
8      <provider>org.hibernate.ejb.HibernatePersistence</provider>
9
10     <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
11     <properties>
12       <property name="hibernate.dialect"
13         value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
14       <property name="hibernate.hbm2ddl.auto" value="update"/>
15       <property name="hibernate.show_sql" value="true"/>
16       <property name="javax.persistence.jdbc.driver"
17         value="com.mysql.jdbc.Driver"/>
18       <property name="javax.persistence.jdbc.user" value="root"/>
19       <property name="javax.persistence.jdbc.password" value="root"/>
20       <property name="javax.persistence.jdbc.url"
21         value="jdbc:mysql://localhost:3306/K21_entity_manager_bd"/>
22
23       <property name="hibernate.cache.region.factory_class"
24         value="org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory"/>
25
26     </properties>
27   </persistence-unit>
28 </persistence>
```

Código XML 3.2: persistence.xml

23 No pacote `br.com.k19.modelo` do projeto **K19-EntityManager**, crie uma entidade para representar cidades. Habilite o cache de segundo nível para essa entidade.

```
@Entity
@Cacheable(true)
public class Cidade {

  @Id @GeneratedValue
  private Long id;

  private String nomeDaCidade;

  private String nomeDoEstado;

  // GETTERS E SETTERS
}
```

Código Java 3.42: Cidade.java

24 No pacote `br.com.k19.testes` do projeto **K19-EntityManager**, crie uma classe chamada `InsereCidades` para inserir algumas cidades no banco de dados.

```
public class InsereCidades {

  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_entity_manager_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    Cidade saoPaulo = new Cidade();
    saoPaulo.setNomeDaCidade("São Paulo");
    saoPaulo.setNomeDoEstado("São Paulo");

    Cidade rioDeJaneiro = new Cidade();
    rioDeJaneiro.setNomeDaCidade("Rio de Janeiro");
    rioDeJaneiro.setNomeDoEstado("Rio de Janeiro");

    Cidade natal = new Cidade();
    natal.setNomeDaCidade("Natal");
    natal.setNomeDoEstado("Rio Grande do Norte");

    manager.persist(saoPaulo);
    manager.persist(rioDeJaneiro);
    manager.persist(natal);

    manager.getTransaction().commit();

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

Código Java 3.43: InsereCidades.java

25 No pacote `br.com.k19.testes` do projeto **K19-EntityManager**, crie uma classe para testar o funcionamento do cache de segundo nível.

```
public class TesteSharedCache {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
```

```
4           .createEntityManagerFactory("K21_entity_manager_pu");
5       EntityManager manager1 = factory.createEntityManager();
6
7       System.out.println("------------------PRIMEIRO FIND-----------------");
8       Cidade cidade = manager1.find(Cidade.class, 1L);
9
10      EntityManager manager2 = factory.createEntityManager();
11
12      System.out.println("------------------SEGUNDO FIND------------------");
13      cidade = manager2.find(Cidade.class, 1L);
14    }
15  }
```

Código Java 3.44: TesteSharedCache.java

Observe no console que apenas uma operação select foi realizada.

Cascade

Por padrão, as operações dos Entity Managers são aplicadas somente ao objeto passado como parâmetro para o método que implementa a operação, ou seja, essas operações não são aplicadas aos objetos relacionados ao objeto passado como parâmetro. Por exemplo, suponha um relacionamento entre estados e governadores.

```
1  @Entity
2  public class Estado {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @OneToOne
10   private Governador governador;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 3.45: Estado.java

```
1  @Entity
2  public class Governador {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @OneToOne(mappedBy="governador")
10   private Estado estado;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 3.46: Governador.java

Suponha que um objeto da classe `Estado` e outro da classe `Governador` sejam criados e associados. Se apenas um dos objetos for persistido um erro ocorrerá na sincronização com o banco de dados.

```
manager.getTransaction().begin();

Governador governador = new Governador();
governador.setNome("Rafael Cosentino");

Estado estado = new Estado();
estado.setNome("São Paulo");

governador.setEstado(estado);
estado.setGovernador(governador);

manager.persist(estado);

manager.getTransaction().commit();
```

Código Java 3.47: Persistindo apenas um objeto

Para evitar o erro, os dois objetos precisam ser persistidos.

```
manager.persist(estado);
manager.persist(governador);
```

Código Java 3.48: Persistindo todos os objetos relacionados

Ou então podemos configurar a operação `persist()` para que ela seja aplicada em cascata. Essa configuração pode ser realizada através do atributo **cascade** das anotações de relacionamento.

```
@OneToOne(cascade=CascadeType.PERSIST)
private Governador governador;
```

Código Java 3.49: Configurando a operação `persist()` em cascata

O atributo cascade das anotações de relacionamento pode ser utilizado para configurar o comportamento em cascata para as outras operações dos Entity Managers.

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

Importante
O atributo cascade é unidirecional. Dessa forma, nos relacionamentos bidirecionais para ter o comportamento do cascade nas duas direções é necessário utilizar a propriedade cacasde nas duas entidades.

Exercícios de Fixação

26 Tente persistir um governador e um estado. Crie uma classe chamada `TesteCascade` no pacote **br.com.k19.testes** do projeto **K19-EntityManager**.

```
1  public class TesteCascade {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      Governador governador = new Governador();
8      governador.setNome("Rafael Cosentino");
9
10     Estado estado = new Estado();
11     estado.setNome("São Paulo");
12
13     governador.setEstado(estado);
14     estado.setGovernador(governador);
15
16     manager.getTransaction().begin();
17     manager.persist(estado);
18     manager.getTransaction().commit();
19   }
20 }
```

Código Java 3.50: TesteCascade.java

Execute e observe o erro

27 Modifique a classe `Estado` para configurar a propriedade cascade no relacionamento com governadores.

```
1  @Entity
2  public class Estado {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @OneToOne(cascade=CascadeType.PERSIST,fetch=FetchType.LAZY)
10   private Governador governador;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 3.51: Estado.java

28 Execute a classe `TesteCascade` e observe que não ocorre o mesmo erro que aconteceu anteriormente.

Remoção de Objetos Órfãos

Em determinados relacionamentos, a existência das instâncias de uma entidade depende fortemente do vínculo com as instâncias da outra entidade. Por exemplo, considere as entidades `Topico` e `Comentario`. Um comentário só deve existir se estiver vinculado a um tópico. Nesse caso, podemos configurar o comportamento em cascata no relacionamento entre tópicos e comentários para a operação `remove()`. Dessa forma, quando um tópico for removido, os comentários correspondentes

também serão removidos.

```
@Entity
public class Topico {

  @Id @GeneratedValue
  private Long id;

  @OneToMany(cascade=CascadeType.REMOVE)
  private List<Comentario> comentarios;

  private String titulo;

  // GETTERS E SETTERS
}
```

Código Java 3.52: Topico.java

```
@Entity
public class Comentario {

  @Id @GeneratedValue
  private Long id;

  @Temporal(TemporalType.DATE)
  private Calendar data;

  // GETTERS E SETTERS
}
```

Código Java 3.53: Comentario.java

```
Topico topico = manager.find(Topico.class, 1L);

manager.getTransaction().begin();

// os comentários desse tópico serão removidos em cascata
manager.remove(topico);

manager.getTransaction().commit();
```

Código Java 3.54: Removendo um tópico e seus comentários

Contudo, se os vínculos entre um tópico e seus comentários forem desfeitos, os comentários não serão removidos automaticamente. No exemplo abaixo, os comentários serão mantidos no banco de dados.

```
Topico topico = manager.find(Topico.class, 1L);

manager.getTransaction().begin();

topico.getComentarios().clear();

manager.getTransaction().commit();
```

Código Java 3.55: Desvinculando os comentários de um tópico

Podemos determinar que todo comentário não vinculado a um tópico deve ser automaticamente removido. Essa configuração pode ser realizada através do atributo `orphanRemoval` das anotações `@OneToOne` e `@OneToMany`. Veja o exemplo a seguir.

```
@Entity
```

```
2  public class Topico {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    @OneToMany(orphanRemoval=true)
8    private List<Comentario> comentarios;
9
10   private String titulo;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 3.56: Topico.java

Pare para pensar...
Não é necessário definir `cascade=CascadeType.REMOVE` em um relacionamento com `orphanRemoval=true`.

Exercícios de Fixação

29 No pacote `br.com.k19.modelo` do projeto **K19-EntityManager**, crie as seguintes entidades:

```
1  @Entity
2  public class Comentario {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    @Temporal(TemporalType.DATE)
8    private Calendar data;
9
10   // GETTERS E SETTERS
11 }
```

Código Java 3.57: Comentario.java

```
1  @Entity
2  public class Topico {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    @OneToMany(cascade={CascadeType.PERSIST,CascadeType.REMOVE})
8    private List<Comentario> comentarios = new ArrayList<Comentario>();
9
10   private String titulo;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 3.58: Topico.java

30 Adicione um tópico e alguns comentários. Crie uma classe chamada `AdicionaTopicoComentarios` no pacote `br.com.k19.testes` do projeto **K19-EntityManager**.

```
1  public class AdicionaTopicoComentarios {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
5
6      EntityManager manager = factory.createEntityManager();
7
8      Topico topico = new Topico();
9      topico.setTitulo("K19 - Orphan Removal");
10
11     for(int i = 0; i < 10; i++){
12       Comentario comentario = new Comentario();
13       comentario.setData(Calendar.getInstance());
14       topico.getComentarios().add(comentario);
15     }
16
17     manager.getTransaction().begin();
18
19     manager.persist(topico);
20
21     manager.getTransaction().commit();
22
23     manager.close();
24     factory.close();
25   }
26 }
```

Código Java 3.59: AdicionaTopicoComentarios.java

Execute a classe `AdicionaTopicoComentarios`.

31 Consulte os dados das tabelas `Topico`, `Comentario` e `Topico_Comentario` através do MySQL Workbench.

32 Desfaça o vínculo entre o tópico e os comentários adicionados anteriormente. Para isso, crie uma classe chamada `TesteOrphanRemoval` no pacote `br.com.k19.testes` do projeto **K19-Entity-Manager**.

```
1  public class TesteOrphanRemoval {
2
3    public static void main(String[] args) {
4      EntityManagerFactory factory =
5        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
6
7      EntityManager manager = factory.createEntityManager();
8
9      manager.getTransaction().begin();
10
11     Topico topico = manager.find(Topico.class, 1L);
12     topico.getComentarios().clear();
13
14     manager.getTransaction().commit();
15
16     manager.close();
17     factory.close();
18   }
19 }
```

Código Java 3.60: TesteOrphanRemoval.java

33 Consulte os dados das tabelas `Topico`, `Comentario` e `Topico_Comentario` através do MySQL Workbench. Observe que os comentários ainda estão persistidos na tabela `Comentario`. Apenas os vínculos entre o tópico e os comentários foram desfeitos. Verifique a tabela `Topico_Comentario`.

34 Apague as tabelas `Topico`, `Comentario` e `Topico_Comentario` através do MySQL Workbench.

35 Execute novamente a classe `AdicionaTopicoComentarios` e consulte os dados das tabelas `Topico`, `Comentario` e `Topico_Comentario` através do MySQL Workbench.

36 Altere a classe `Topico` para aplicar o `orphanRemoval=true` no relacionamento entre tópicos e comentários.

```
1  @Entity
2  public class Topico {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    @OneToMany(cascade={CascadeType.PERSIST}, orphanRemoval=true)
8    private List<Comentario> comentarios = new ArrayList<Comentario>();
9
10   private String titulo;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 3.61: Topico.java

37 Execute novamente a classe `TesteOrphanRemoval` e consulte os dados das tabelas `Topico`, `Comentario` e `Topico_Comentario` através do MySQL Workbench. Observe que agora os comentários foram apagados da tabela `Comentario`.

Callbacks

Podemos monitorar o ciclo de vida dos objetos das entidades da nossa aplicação. Determinados eventos podem ser capturados e podemos associar métodos a esses eventos. Esses métodos são chamados de **callbacks**. Um método de callback é executado automaticamente quando o evento associado a ele é disparado. Veja os eventos que podem ser monitorados na listagem abaixo:

PrePersist: Disparado quando um objeto novo é persistido através da operação `persist()` ou `merge()` dos Entity Managers. Esse evento também é disparado para os objetos persistidos em cascata.

PostPersist: Disparado durante a execução de uma sincronização, após a operação `insert` correspondente ao objeto que foi persistido. Um rollback na trasanção corrente pode desfazer a operação `insert`, mas não o evento.

PreRemove: Disparado quando um objeto gerenciado é removido através da operação `remove()` dos Entity Managers. Esse evento também é disparado para os objetos removidos em cascata.

PostRemove: Disparado durante a execução de uma sincronização, após a operação `delete` correspondente ao objeto que foi removido. Um rollback na transação corrente pode desfazer a operação `delete`, mas não o evento.

PreUpdate: Disparado durante a execução de uma sincronização, antes da operação `update` correspondente ao objeto que foi alterado.

PostUpdate: Disparado durante a execução de uma sincronização, após a operação `update` correspondente ao objeto que foi alterado. Um rollback na transanção corrente pode desfazer a operação `update`, mas não o evento.

PostLoad: Disparado depois que uma instância de uma entidade foi carregada com os dados do banco de dados.

Os métodos de callback associados aos eventos acima listados podem ser definidos nas classes das entidades da nossa aplicação. Esses métodos devem ser anotados com `@PrePersist`, `@PostPersist`, `@PreRemove`, `@PostRemove`, `@PreUpdate`, `@PostUpdate` ou `@PostLoad` para associá-los aos eventos correspondentes. No exemplo abaixo, adicionamos um método de callback para cada evento definido pela especificação JPA.

```
@Entity
public class Produto {

  @Id @GeneratedValue
  private Long id;

  private String nome;

  private Double preco;


  @PrePersist
  public void prePersist(){
    System.out.println("Persistindo um novo objeto com persist() ou merge()...");
  }

  @PostPersist
  public void postPersist(){
    System.out.println("O comando insert foi executado no banco de dados...");
    System.out.println("Um rollback ainda pode desfazer o comando insert...");
  }

  @PreRemove
  public void preRemove(){
    System.out.println("Removendo um objeto gerenciado com remove()...");
  }

  @PostRemove
  public void postRemove(){
    System.out.println("O comando delete foi executado no banco de dados...");
    System.out.println("Um rollback ainda pode desfazer o comando delete...");
  }

  @PreUpdate
  public void preUpdate(){
    System.out.println("O comando update executará no banco de dados...");
  }

  @PostUpdate
  public void postUpdate(){
    System.out.println("O comando update foi executado no banco de dados...");
    System.out.println("Um rollback ainda pode desfazer o comando update...");
  }
```

```
44
45    @PostLoad
46    public void postLoad(){
47      System.out.println("Um objeto foi carregado com os dados do banco de dados.");
48    }
49  }
```

Código Java 3.62: Produto.java

Mais Sobre

Um mesmo método de callback pode estar associado a dois ou mais eventos. No exemplo abaixo, o método `callback()` foi associado a todos os eventos JPA.

```
1  @Entity
2  public class Produto {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    private Double preco;
10
11   @PrePersist
12   @PostPersist
13   @PreRemove
14   @PostRemove
15   @PreUpdate
16   @PostUpdate
17   @PostLoad
18   public void callback(){
19     System.out.println("tratando todos os eventos");
20   }
21 }
```

Código Java 3.63: Produto.java

Mais Sobre

Podemos reaproveitar métodos de callback para duas ou mais entidades. Para isso, devemos defini-los em uma classe separada e depois associar essa classe às entidades desejadas através da anotação `@EntityListeners`.

```
1  public class K19Listener {
2    @PrePersist
3    @PostPersist
4    @PreRemove
5    @PostRemove
6    @PreUpdate
7    @PostUpdate
8    @PostLoad
9    public void callback(){
10     System.out.println("tratando todos os eventos");
11   }
12 }
```

Código Java 3.64: K19Listener.java

```
1  @Entity
2  @EntityListeners(K19Listener.class)
```

```
3  public class Produto {
4
5    @Id @GeneratedValue
6    private Long id;
7
8    private String nome;
9
10   private Double preco;
11 }
```

Código Java 3.65: Produto.java

Exercícios de Fixação

38 Defina uma classe chamada `Produto` no pacote `br.com.k19.modelo` do projeto **K19-EntityManager**,

```
1  @Entity
2  public class Produto {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    private Double preco;
10
11   @PrePersist
12   public void prePersist(){
13     System.out.println("Persistindo um novo objeto com persist() ou merge()...");
14   }
15
16   @PostPersist
17   public void postPersist(){
18     System.out.println("O comando insert foi executado no banco de dados...");
19     System.out.println("Um rollback ainda pode desfazer o comando insert...");
20   }
21
22   // GETTERS E SETTERS
23 }
```

Código Java 3.66: Produto.java

39 Adicione alguns produtos no banco de dados. Crie uma classe chamada `AdicionaProduto` no pacote `br.com.k19.testes` do projeto **K19-EntityManager**.

```
1  public class AdicionaProduto {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_entity_manager_pu");
5
6      EntityManager manager = factory.createEntityManager();
7
8      Produto p = new Produto();
9      p.setNome("K19 - Caneta");
10     p.setPreco(4.56);
11
12     manager.getTransaction().begin();
```

```
13
14      manager.persist(p);
15
16      manager.getTransaction().commit();
17
18      factory.close();
19    }
20  }
```

Código Java 3.67: AdicionaProduto.java

Execute a classe `AdicionaProduto` e observe as mensagens no console.

Concorrência

Quando dois Entity Managers manipulam objetos da mesma entidade e com o mesmo identificador, um resultado incorreto pode ser obtido. Por exemplo, suponha que os seguintes trechos de código sejam executados em paralelo.

```
1 manager1.getTransaction().begin();
2
3 Conta x = manager1.find(Conta.class, 1L);
4
5 x.setSaldo(x.getSaldo() + 500);
6
7 manager1.getTransaction().commit();
```

Código Java 3.68: Depósito na conta

```
1 manager2.getTransaction().begin();
2
3 Conta y = manager2.find(Conta.class, 1L);
4
5 y.setSaldo(y.getSaldo() - 500);
6
7 manager2.getTransaction().commit();
```

Código Java 3.69: Saque na conta

O primeiro trecho acrescenta 500 reais ao saldo da conta com identificador 1. O segundo trecho retira 500 reais da mesma conta. Dessa forma, o saldo dessa conta deve possuir o mesmo valor antes e depois desses dois trechos de código serem executados. Contudo, dependendo da ordem na qual as linhas dos dois trechos são executadas, o resultado pode ser outro.

Por exemplo, suponha que o valor inicial do saldo da conta com identificador 1 seja 2000 reais e as linhas dos dois trechos são executadas na seguinte ordem:

```
1  manager1.getTransaction().begin();
2  manager2.getTransaction().begin();
3
4  Conta x = manager1.find(Conta.class, 1L); // x: saldo = 2000
5
6  x.setSaldo(x.getSaldo() + 500); // x: saldo = 2500
7
8  Conta y = manager2.find(Conta.class, 1L); // y: saldo = 2000
9
10 y.setSaldo(y.getSaldo() - 500); // y: saldo = 1500
11
12 manager1.getTransaction().commit(); // Conta 1: saldo = 2500
```

```
13 manager2.getTransaction().commit(); // Conta 1: saldo = 1500
```

Nesse caso, o saldo final seria 1500 reais.

Exercícios de Fixação

40 Acrescente no pacote `br.com.k19.modelo` do projeto **K19-EntityManager** uma classe para definir contas bancárias.

```
1  @Entity
2  public class Conta {
3
4    @Id
5    @GeneratedValue
6    private Long id;
7
8    private double saldo;
9
10   // GETTERS AND SETTERS
11 }
```

Código Java 3.71: Conta.java

41 Adicione uma classe no pacote `br.com.k19.testes` para cadastrar uma conta no banco de dados.

```
1  public class AdicionaConta {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_entity_manager_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Conta c = new Conta();
10     c.setSaldo(2000);
11     manager.persist(c);
12
13     manager.getTransaction().commit();
14
15     manager.close();
16     factory.close();
17   }
18 }
```

Código Java 3.72: AdicionaConta.java

Execute e verifique a tabela Conta.

42 Simule o problema de concorrência entre Entity Managers adicionando a seguinte classe no pacote **br.com.k19.testes**.

```
1  public class TestaAcessoConcorrente {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
```

```
4         .createEntityManagerFactory("K21_entity_manager_pu");
5
6     EntityManager manager1 = factory.createEntityManager();
7     EntityManager manager2 = factory.createEntityManager();
8
9     manager1.getTransaction().begin();
10    manager2.getTransaction().begin();
11
12    Conta conta1 = manager1.find(Conta.class, 1L);
13
14    conta1.setSaldo(conta1.getSaldo() + 500);
15
16    Conta conta2 = manager2.find(Conta.class, 1L);
17
18    conta2.setSaldo(conta2.getSaldo() - 500);
19
20    manager1.getTransaction().commit();
21    manager2.getTransaction().commit();
22
23    manager1.close();
24    manager2.close();
25    factory.close();
26  }
27 }
```

Código Java 3.73: TestaAcessoConcorrente.java

Execute e verifique que o saldo final da conta com identificador 1 é 1500, mas o correto seria 2000.

Locking Otimista

Para solucionar o problema da concorrência entre Entity Managers, podemos aplicar a idéia de **Locking Otimista**. Nessa abordagem, um atributo para determinar a versão dos registros é acrescentado na entidade. Esse atributo deve ser anotado com `@Version` e seu tipo deve ser `short`, `int`, `long`, `Short`, `Integer`, `Long` ou `java.sql.Timestamp`.

```
1  @Entity
2  public class Conta {
3
4    @Id
5    @GeneratedValue
6    private Long id;
7
8    private double saldo;
9
10   @Version
11   private Long versao;
12
13   // GETTERS AND SETTERS
14 }
```

Código Java 3.74: Conta.java

Toda vez que um Entity Manager modifica um registro da tabela correspondente à classe `Conta`, o campo referente ao atributo anotado com `@Version` é atualizado.

Agora, antes de modificar um registro da tabela referente à classe `Conta`, os Entity Managers comparam a versão do registro no banco de dados com a do objeto que eles possuem.

Se as versões forem a mesma, significa que nenhum outro Entity Manager modificou o registro e então as modificações podem ser executadas sem problemas. Caso contrário, se as versões forem diferentes, significa que algum outro Entity Manager modificou o registro e então as modificações são abortadas e uma exceção é lançada. Em geral, as aplicações devem capturar essa exceção e tentar refazer a operação.

Exercícios de Fixação

43 Acrescente um atributo na classe `Conta` anotado com `@Version`.

```
@Entity
public class Conta {

  @Id
  @GeneratedValue
  private Long id;

  private double saldo;

  @Version
  private Long versao;

  // GETTERS AND SETTERS
}
```

Código Java 3.75: Conta.java

44 Apague a tabela `Conta` através do MySQL Workbench.

45 Execute a classe `AdicionaConta` e verifique a tabela `Conta` através do MySQL Workbench.

46 Execute a classe `TestaAcessoConcorrente` e observe a exceção gerada pelo segundo Entity Manager.

Locking Pessimista

Outra abordagem para lidar com o problema da concorrência entre Entity Managers é o **Locking Pessimista**. Nessa abordagem, um Entity Manager pode "travar" os registros, fazendo com que os outros Entity Managers que desejem manipular os mesmos registros tenham que aguardar.

Há várias maneiras de utilizar o locking pessimista. Uma delas é passar mais um parâmetro quando um objeto é buscado através do método `find()`.

```
Conta x = manager.find(Conta.class, 1L, LockModeType.PESSIMISTIC_WRITE);
```

Uma grande dificuldade em utilizar locking pessimista é que podemos gerar um `deadlock`. Suponha que dois Entity Managers busquem o mesmo objeto na mesma thread utilizando o locking

pessimista como mostra o código a seguir.

```
1 Conta x = manager1.find(Conta.class, 1L, LockModeType.PESSIMISTIC_WRITE);
2 Conta y = manager2.find(Conta.class, 1L, LockModeType.PESSIMISTIC_WRITE);
3 manager1.commit();// NUNCA VAI EXECUTAR ESSA LINHA
```

Na linha 1, o primeiro Entity Manager "trava" a conta com identificador 1 e esse objeto só será liberado na linha 3. Na linha 2, o segundo Entity Manager vai esperar o primeiro liberar o objeto, impedindo que a linha 3 seja executada. Dessa forma, a linha 3 nunca será executada. Depois de um certo tempo esperando na linha 2, o segundo Entity Manager lança uma exceção.

Exercícios de Fixação

47 Teste o problema de deadlock quando o locking pessimista é utilizado. Adicione a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-EntityManager**.

```
1  public class TestaDeadLock {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_entity_manager_pu");
5
6      EntityManager manager1 = factory.createEntityManager();
7      EntityManager manager2 = factory.createEntityManager();
8
9      manager1.getTransaction().begin();
10     manager2.getTransaction().begin();
11
12     manager1.find(Produto.class, 1L, LockModeType.PESSIMISTIC_WRITE);
13     System.out.println("Produto 1 travado");
14     manager2.find(Produto.class, 1L, LockModeType.PESSIMISTIC_WRITE);
15     System.out.println("Essa mensagem não será impressa!!!");
16
17     manager1.getTransaction().commit();
18     manager2.getTransaction().commit();
19
20     manager1.close();
21     manager2.close();
22
23     factory.close();
24   }
25 }
```

Código Java 3.78: TestaDeadLock.java

Execute e aguarde até ocorrer uma exceção.

JPQL

CAPÍTULO 4

A capacidade que os bancos de dados possuem para realizar consultas de maneira eficiente é um forte argumento para utilizá-los. A definição e os resultados das consultas nos banco de dados são fortemente baseados no modelo relacional. Por outro lado, é natural que as aplicações baseadas no modelo orientado a objetos desejem que a definição e os resultados das consultas sejam baseados no paradigma orientado a objetos.

Por isso, os provedores de JPA oferecem mecanismos para realizar consultas de uma maneira orientada a objetos. Para ser mais exato, a especificação JPA 2 define dois mecanismos para realizar consultas orientadas a objetos: o primeiro utiliza uma linguagem específica para consultas chamada JPQL (Java Persistence Query Language) e o segundo é basicamente uma biblioteca Java para consultas.

Outro fator fundamental para justificar a utilização dos mecanismos de consulta do JPA 2 é que eles são independentes dos mecanismos específicos de consulta do banco de dados. Podemos definir uma consulta em JPQL ou Criteria e executá-la em qualquer banco de dados suportado pelo provedor JPA. Neste capítulo, mostraremos o funcionamento da JPQL.

Consultas Dinâmicas

Consultas em JPQL podem ser definidas em qualquer classe Java, dentro de um método por exemplo. Para criar uma consulta, devemos utilizar o método **createQuery()** passando uma string com o código JPQL. Consultas criadas dessa maneira são chamadas de consultas dinâmicas.

```
1 public void umMetodoQualquer() {
2   String jpql = "SELECT p FROM Pessoa p";
3   Query query = manager.createQuery(jpql);
4 }
```

Apesar da flexibilidade, criar consultas dinâmicas pode prejudicar a performance da aplicação. Por exemplo, se uma consulta dinâmica é criada dentro de um método toda vez que esse método for chamado, o código JPQL dessa consulta será processado pelo provedor. Uma alternativa às consultas dinâmicas são as **Named Queries**, que são menos flexíveis porém mais eficientes.

Named Query

Diferentemente de uma consulta dinâmica, uma Named Query é processada apenas no momento da inicialização da unidade de persistência. Além disso, os provedores JPA podem mapear as Named Queries para Stored Procedures pre-compiladas no banco de dados melhorando a performance das consultas.

As Named Queries são definidas através de anotações nas classes que implementam as entidades. Podemos aplicar a anotação **@NamedQuery** quando queremos definir apenas uma consulta ou a anotação **@NamedQueries** quando queremos definir várias consultas.

```
1 @NamedQuery(name="Pessoa.findAll", query="SELECT p FROM Pessoa p")
2 class Pessoa {
3   ...
4 }
```

Código Java 4.2: Pessoa.java

```
1 @NamedQueries({
2   @NamedQuery(name="Pessoa.findAll", query="SELECT p FROM Pessoa p"),
3   @NamedQuery(name="Pessoa.count", query="SELECT COUNT(p) FROM Pessoa p")
4 })
5 class Pessoa {
6   ...
7 }
```

Código Java 4.3: Pessoa.java

Mais Sobre

O nome de uma Named Query deve ser único na unidade de persistência. Para evitar nomes repetidos, uma boa prática é utilizar o nome de uma entidade como prefixo para o nome da consulta. Em geral, a entidade escolhida é a que tem maior relação com a consulta. Veja o exemplo abaixo.

```
1 @NamedQuery(name="Pessoa.findAll", query="SELECT p FROM Pessoa p")
```

Para executar uma Named Query, devemos utilizar o método **createNamedQuery()**. Apesar do nome, esse método não cria uma Named Query, pois as Named Queries são criadas na inicialização da unidade de persistência. Esse método apenas recupera uma Named Query existente para ser utilizada.

```
1 public void listaPessoas() {
2   Query query = manager.createNamedQuery("Pessoa.findAll");
3   List<Pessoa> pessoas = query.getResultList();
4 }
```

Código Java 4.5: Recuperando uma Named Query

Parâmetros

Para tornar as consultas em JPQL mais genéricas e evitar problemas com SQL Injection, devemos parametrizá-las. Adicionar um parâmetro em uma consulta é simples. Para isso, basta utilizar caractere “:” seguido do nome do argumento.

```
1 @NamedQuery(name="Pessoa.findByIdade",
2       query="SELECT p FROM Pessoa p WHERE p.idade > :idade")
```

Código Java 4.6: Parametrizando uma consulta

Antes de executar uma consulta com parâmetros, devemos definir os valores dos argumentos.

```
public void listaPessoas() {
  Query query = manager.createNamedQuery("Pessoa.findByIdade");
  query.setParameter("idade", 18);
  List<Pessoa> pessoasComMaisDe18 = query.getResultList();
}
```

É possível também adicionar parâmetros em uma consulta de maneira ordinal com o uso do caractere "?" seguido de um número.

```
@NamedQuery(name="Pessoa.findByIdade",
      query="SELECT p FROM Pessoa p WHERE p.idade > ?1")
```

```
public void listaPessoas() {
  Query query = manager.createNamedQuery("Pessoa.findByIdade");
  query.setParameter(1, 18);
  List<Pessoa> pessoasComMaisDe18 = query.getResultList();
}
```

Exercícios de Fixação

1 Crie um projeto no eclipse chamado **K19-JPQL**. Copie a pasta `lib` do projeto `K19-JPA2-Hibernate` para o projeto `K19-JPQL`. Depois adicione os jars dessa pasta no classpath desse novo projeto.

2 Abra o MySQL Workbench e apague a base de dados **K21_jpql_bd** se existir. Depois crie a base de dados **K21_jpql_bd**.

3 Copie a pasta `META-INF` do projeto `K19-JPA2-Hibernate` para dentro da pasta `src` do projeto `K19-JPQL`. Altere o arquivo `persistence.xml` do projeto **K19-JPQL**, modificando os nomes da unidade de persistência e da base da dados. Veja como o código deve ficar:

```
<?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="K21_jpql_pu" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property
        name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5InnoDBDialect" />

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

      <property
        name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver" />
```

```
23
24      <property
25        name="javax.persistence.jdbc.user"
26        value="root" />
27
28      <property
29        name="javax.persistence.jdbc.password"
30        value="root" />
31
32      <property
33        name="javax.persistence.jdbc.url"
34        value="jdbc:mysql://localhost:3306/K21_jpql_bd"/>
35    </properties>
36  </persistence-unit>
37 </persistence>
```

Código XML 4.1: persistence.xml

4 Crie um pacote chamado `br.com.k19.modelo` no projeto `K19-JPQL` e adicione as seguintes classes:

```
1  @Entity
2  public class Livro {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    private Double preco;
10
11   // GETTERS E SETTERS
12 }
```

Código Java 4.10: Livro.java

```
1  @Entity
2  public class Autor {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @ManyToMany
10   private Collection<Livro> livros = new ArrayList<Livro>();
11
12   // GETTERS E SETTERS
13 }
```

Código Java 4.11: Autor.java

5 Carregue o banco de dados com as informações de alguns livros e autores. Adicione a seguinte classe em um novo pacote chamado `br.com.k19.testes` dentro do projeto **K19-JPQL**.

```
1  public class PopulaBanco {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
```

```
8
9       Livro livro1 = new Livro();
10      livro1.setNome("The Battle for Your Mind");
11      livro1.setPreco(20.6);
12      manager.persist(livro1);
13
14      Livro livro2 = new Livro();
15      livro2.setNome("Differentiate or Die");
16      livro2.setPreco(15.8);
17      manager.persist(livro2);
18
19      Livro livro3 = new Livro();
20      livro3.setNome("How to Transform Your Ideas");
21      livro3.setPreco(32.7);
22      manager.persist(livro3);
23
24      Livro livro4 = new Livro();
25      livro4.setNome("Digital Fortress");
26      livro4.setPreco(12.9);
27      manager.persist(livro4);
28
29      Livro livro5 = new Livro();
30      livro5.setNome("Marketing in an Era of Competition, Change and Crisis");
31      livro5.setPreco(26.8);
32      manager.persist(livro5);
33
34      Autor autor1 = new Autor();
35      autor1.setNome("Patrick Cullen");
36      autor1.getLivros().add(livro2);
37      autor1.getLivros().add(livro4);
38      manager.persist(autor1);
39
40      Autor autor2 = new Autor();
41      autor2.setNome("Fraser Seitel");
42      autor2.getLivros().add(livro3);
43      manager.persist(autor2);
44
45      Autor autor3 = new Autor();
46      autor3.setNome("Al Ries");
47      autor3.getLivros().add(livro1);
48      manager.persist(autor3);
49
50      Autor autor4 = new Autor();
51      autor4.setNome("Jack Trout");
52      autor4.getLivros().add(livro1);
53      autor4.getLivros().add(livro2);
54      autor4.getLivros().add(livro5);
55      manager.persist(autor4);
56
57      Autor autor5 = new Autor();
58      autor5.setNome("Steve Rivkin");
59      autor5.getLivros().add(livro2);
60      autor5.getLivros().add(livro3);
61      autor5.getLivros().add(livro5);
62      manager.persist(autor5);
63
64      manager.getTransaction().commit();
65
66      manager.close();
67      factory.close();
68    }
69  }
```

Código Java 4.12: PopulaBanco.java

Execute e consulte os dados no banco de dados.

6 Teste as consultas dinâmicas com JPQL. Crie a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-JPQL**.

```
1  public class TesteConsultaDinamicas {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      Query query = manager.createQuery("select a from Autor a");
8      List<Autor> autores = query.getResultList();
9
10     for (Autor autor : autores) {
11       System.out.println("Autor: " + autor.getNome());
12       Collection<Livro> livros = autor.getLivros();
13
14       for (Livro livro : livros) {
15         System.out.println("Livro: " + livro.getNome());
16         System.out.println("Preço: " + livro.getPreco());
17         System.out.println();
18       }
19       System.out.println();
20     }
21
22     manager.close();
23     factory.close();
24   }
25 }
```

Código Java 4.13: TesteConsultaDinamicas.java

Execute e observe que as consultas são realizadas aos poucos devido ao carregamento em modo LAZY.

7 Para testar as Named Queries, acrescente a anotação `@NamedQuery` na classe `Autor`.

```
1  @Entity
2  @NamedQuery(name="Autor.findAll", query="select a from Autor a")
3  public class Autor {
4    ...
5  }
```

Código Java 4.14: Autor.java

8 Em seguida, crie um teste para Named Query definida no exercício anterior. Adicione no pacote `br.com.k19.testes` do projeto **K19-JPQL** a seguinte classe:

```
1  public class TesteNamedQuery {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      Query query = manager.createNamedQuery("Autor.findAll");
8      List<Autor> autores = query.getResultList();
9
10     for (Autor autor : autores) {
11       System.out.println("Autor: " + autor.getNome());
12       Collection<Livro> livros = autor.getLivros();
13
14       for (Livro livro : livros) {
15         System.out.println("Livro: " + livro.getNome());
```

```
16          System.out.println("Preço: " + livro.getPreco());
17          System.out.println();
18        }
19        System.out.println();
20      }
21
22      manager.close();
23      factory.close();
24    }
25  }
```

Código Java 4.15: TesteNamedQuery

Execute e observe que as consultas são realizadas aos poucos devido ao carregamento em modo LAZY.

9 Acrescente a anotação `@NamedQuery` na classe `Livro` para definir uma consulta por preço mínimo utilizando parâmetros.

```
1  @Entity
2   @NamedQuery(name="Livro.findByPrecoMinimo",
3         query="select livro from Livro livro where livro.preco >= :preco")
4  public class Livro {
5    ...
6  }
```

Código Java 4.16: Livro.java

10 Em seguida, crie um teste para Named Query definida no exercício anterior. Adicione no pacote `br.com.k19.testes` do projeto **K19-JPQL** a seguinte classe:

```
1  public class TesteParametros {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      Query query = manager.createNamedQuery("Livro.findByPrecoMinimo");
8      query.setParameter("preco", 20.0);
9      List<Livro> livros = query.getResultList();
10
11     for (Livro livro : livros) {
12       System.out.println("Nome: " + livro.getNome());
13       System.out.println("Preço: " + livro.getPreco());
14     }
15
16     manager.close();
17     factory.close();
18   }
19 }
```

Código Java 4.17: TesteParametros.java

Tipos de Resultado

Lista de Entidades

Uma consulta em JPQL pode devolver uma lista com os objetos de uma entidade que são compatíveis com os filtros da pesquisa. Por exemplo, considere a seguinte consulta:

```
1 String query = "SELECT p FROM Pessoa p";
```

O resultado dessa pesquisa é uma lista com todas as instâncias da entidade `Pessoa` que foram persistidas. Esse resultado pode ser obtido através do método **getResultList()**.

```
1 String query = "SELECT p FROM Pessoa p";
2 Query query = manager.createQuery(query);
3 List<Pessoa> pessoas = query.getResultList();
```

Nesse caso, os objetos da listagem devolvida pela consulta estão no estado managed, ou seja, alterações realizadas no conteúdo desses objetos são sincronizadas com o banco de dados de acordo com as regras de sincronização.

Typed Query

O compilador da linguagem Java não verifica a compatibilidade entre a variável e o resultado da consulta. Na consulta abaixo, o compilador não sabe se o método `getResultList()` devolverá de fato uma lista de pessoas, pois ele não processa a string que define a consulta.

```
1 String query = "SELECT p FROM Pessoa p";
2 Query query = manager.createQuery(query);
3 List<Pessoa> pessoas = query.getResultList();
```

Sem a ajuda do compilador, há mais chances de ocorrerem erros de execução. Por exemplo, a consulta abaixo provocaria uma exceção.

```
1 String query = "SELECT p FROM Pessoa p";
2 Query query = manager.createQuery(query);
3 List<Departamento> departamentos = query.getResultList();
```

Para diminuir a chance de erro, podemos utilizar as **Typed Queries**. Nesse tipo de consulta, o compilador também não verifica o código JPQL. No entanto, devemos especificar o tipo de resultado esperado para que o compilador verifique o restante do código e garanta que a utilização do resultado da consulta seja compatível com o tipo especificado.

```
1 String query = "SELECT p FROM Pessoa p";
2 TypedQuery<Pessoa> query = manager.createQuery(query, Pessoa.class);
3 List<Pessoa> pessoas = query.getResultList();
```

Lista de Objetos Comuns

A consulta abaixo devolve uma lista de pessoas.

```
1 String query = "SELECT p FROM Pessoa p";
2 TypedQuery<Pessoa> query = manager.createQuery(query, Pessoa.class);
3 List<Pessoa> pessoas = query.getResultList();
```

Código Java 4.23: Recuperando a lista de pessoas

Dessa forma, teríamos acesso a todos os dados das pessoas dessa listagem. Contudo, muitas vezes, não desejamos todas as informações. Por exemplo, se a nossa aplicação precisa apresentar uma lista dos nomes das pessoas cadastradas, não é necessário recuperar nada além dos nomes.

Quando definimos as consultas, podemos determinar o que elas devem trazer de fato do banco de dados. Por exemplo, a consulta abaixo recupera apenas os nomes das pessoas.

```
1 String query = "SELECT p.nome FROM Pessoa p";
2 TypedQuery<String> query = manager.createQuery(query, String.class);
3 List<String> nomes = query.getResultList();
```

Código Java 4.24: Recuperando apenas os nomes das pessoas

Valores únicos

Suponha que desejamos saber quantas pessoas possuem mais do que 18 anos. Nesse caso, não é necessário trazer mais do que um número do banco de dados. Em outras palavras, o resultado dessa consulta não deve ser uma lista, mas sim um valor numérico. Para isso, podemos aplicar as funções de agregação:

AVG	Calcula a média de um conjunto de números
COUNT	Contabiliza o número de resultados
MAX	Recupera o maior elemento um conjunto de números
MIN	Recupera o menor elemento um conjunto de números
SUM	Calcula a soma de um conjunto de números

Tabela 4.1: Funções de agregação

A consulta abaixo devolve a quantidade de pessoas persistidas. Observe que isso é feito utilizando-se o método **getSingleResult()** ao invés do **getResultList()**, pois o resultado não é uma lista.

```
1 String query = "SELECT COUNT(p) FROM Pessoa p";
2 TypedQuery<Long> query = manager.createQuery(query, Long.class);
3 Long numeroDePessoas = query.getSingleResult();
```

Código Java 4.25: Recuperando o número de pessoas persistidas

A consulta abaixo devolve a maior idade entre as pessoas persistidas.

```
1 String query = "SELECT MAX(p.idade) FROM Pessoa p";
2 TypedQuery<Integer> query = manager.createQuery(query, Integer.class);
3 Integer maiorIdade = query.getSingleResult();
```

Código Java 4.26: Idade máxima entre as pessoas persistidas

Resultados Especiais

Algumas consultas possuem resultados complexos. Por exemplo, suponha que desejamos obter uma listagem com os nomes dos funcionários e o nome do departamento em que o funcionário trabalha.

```
1 "SELECT f.nome, f.departamento.nome FROM Funcionario f";
```

Nesse caso, o resultado será uma lista de array de Object. Para manipular essa lista, devemos lidar com o posicionamento dos dados nos arrays.

```
String query = "SELECT f.nome, f.departamento.nome FROM Funcionario f";
Query query = manager.createQuery(query);
List<Object[]> lista = query.getResultList();

for(Object[] tupla : lista) {
  System.out.println("Funcionário: " + tupla[0]);
  System.out.println("Departamento: " + tupla[1]);
}
```

Operador NEW

Para contornar a dificuldade de lidar com o posicionamento dos dados nos arrays, podemos criar uma classe para modelar o resultado da nossa consulta e aplicar o operador **NEW** no código JPQL.

```
package resultado;

class FuncionarioDepartamento {
  private String funcionarioNome;
  private String departamentoNome;

  public FuncionarioDepartamento(String funcionarioNome, String departamentoNome) {
    this.funcionarioNome = funcionarioNome;
    this.departamentoNome = departamentoNome;
  }

  // GETTERS E SETTERS
}
```

Código Java 4.29: FuncionarioDepartamento.java

```
String query = "SELECT NEW resultado.FuncionarioDepartamento(f.nome,
          f.departamento.nome) FROM Funcionario f";

Query query = manager.createQuery(query);

List<FuncionarioDepartamento> resultados = query.getResultList();

for(FuncionarioDepartamento resultado : resultados) {
  System.out.println("Funcionário: " + resultado.getFuncionarioNome());
  System.out.println("Departamento: " + resultado.getDepartamentoNome());
}
```

A vantagem da utilização de uma classe para modelar o resultado de uma consulta complexa ao invés de um array de Object é que a aplicação não precisa lidar com o posicionamento dos itens do resultado.

Importante
A classe que modela o resultado complexo esperado de uma consulta deve possuir um construtor para receber os dados do resultado.

Importante
Para utilizar uma classe que modela o resultado de uma consulta complexa dentro do

código JPQL com o operador NEW, devemos indicar o nome completo da classe (fully qualified name).

Exercícios de Fixação

11 Teste o recurso de Typed Query utilizando a Named Query `Autor.findAll`. Adicione no pacote `br.com.k19.testes` do projeto **K19-JPQL** a seguinte classe:

```
1  public class TesteTypedQuery {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      TypedQuery<Autor> query = manager.createNamedQuery("Autor.findAll", Autor.class);
8      List<Autor> autores = query.getResultList();
9
10     for (Autor autor : autores) {
11       System.out.println("Autor: " + autor.getNome());
12     }
13
14     manager.close();
15     factory.close();
16   }
17 }
```

Código Java 4.31: TesteTypedQuery.java

Observe que não há mais warnings.

12 Crie um teste para recuperar somente os nomes dos livros cadastrados no banco de dados. Adicione a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-JPQL**.

```
1  public class TesteConsultaObjetosComuns {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      TypedQuery<String> query = manager.createQuery("select livro.nome from Livro ↩
           livro", String.class);
8      List<String> nomes = query.getResultList();
9
10     for (String nome : nomes) {
11       System.out.println(nome);
12     }
13
14     manager.close();
15     factory.close();
16   }
17 }
```

Código Java 4.32: TesteConsultaObjetosComuns.java

13 Crie um teste para recuperar o valor da média dos preços dos livros. Adicione a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-JPQL**.

```
1  public class TesteConsultaLivroPrecoMedio {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      TypedQuery<Double> query = manager.createQuery("select avg(livro.preco) from ↩
           Livro livro", Double.class);
8      Double precoMedio = query.getSingleResult();
9
10     System.out.println("Preço médio: " + precoMedio);
11
12     manager.close();
13     factory.close();
14   }
15 }
```

Código Java 4.33: TesteConsultaLivroPrecoMedio.java

14 No pacote `br.com.k19.modelo` do projeto **K19-JPQL**, crie duas entidades para modelar departamentos e funcionários.

```
1  @Entity
2  public class Departamento {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    // GETTERS E SETTERS
10 }
```

Código Java 4.34: Departamento.java

```
1  @Entity
2  public class Funcionario {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @ManyToOne
10   private Departamento departamento;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 4.35: Funcionario.java

15 Adicione alguns funcionários e departamentos. No pacote `br.com.k19.testes` do projeto **K19-JPQL**, adicione a seguinte classe:

```
1  public class AdicionaFuncionarioDepartamento {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
```

```
5     EntityManager manager = factory.createEntityManager();
6
7     manager.getTransaction().begin();
8
9     Departamento d = new Departamento();
10    d.setNome("Treinamentos");
11
12    Funcionario f = new Funcionario();
13    f.setNome("Rafael Cosentino");
14
15    f.setDepartamento(d);
16
17    manager.persist(f);
18    manager.persist(d);
19
20    manager.getTransaction().commit();
21
22    manager.close();
23    factory.close();
24  }
25 }
```

Código Java 4.36: AdicionaFuncionarioDepartamento.java

16 Crie um teste para recuperar os nomes dos funcionários e os nomes dos seus respectivos departamentos. Adicione a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-JPQL**.

```
1  public class TesteBuscaFuncionarioDepartamento {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      Query query = manager
8          .createQuery("select f.nome, f.departamento.nome from Funcionario f");
9      List<Object[]> lista = query.getResultList();
10
11     for (Object[] tupla : lista) {
12       System.out.println("Funcionário: " + tupla[0]);
13       System.out.println("Departamento: " + tupla[1]);
14     }
15
16     manager.close();
17     factory.close();
18   }
19 }
```

Código Java 4.37: TesteBuscaFuncionarioDepartamento.java

17 No pacote `br.com.k19.modelo` do projeto **K19-JPQL**, crie uma classe para melhorar a manipulação da consulta dos nomes dos funcionários e nomes dos seus respectivos departamentos.

```
1  public class FuncionarioDepartamento {
2    private String funcionario;
3    private String departamento;
4
5    public FuncionarioDepartamento(String funcionario, String departamento) {
6      this.funcionario = funcionario;
7      this.departamento = departamento;
8    }
9
10   // GETTERS
11 }
```

Código Java 4.38: FuncionarioDepartamento.java

18 Altere a classe `TesteBuscaFuncionarioDepartamento` para que ela utilize o operador `NEW` da JPQL.

```
1  public class TesteBuscaFuncionarioDepartamento {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      Query query = manager
8        .createQuery("select new br.com.k19.modelo.FuncionarioDepartamento(f.nome, f.↩
       departamento.nome) from Funcionario f");
9
10     List<FuncionarioDepartamento> lista = query.getResultList();
11
12     for (FuncionarioDepartamento fd : lista) {
13       System.out.println("Funcionário: " + fd.getFuncionario());
14       System.out.println("Departamento: " + fd.getDepartamento());
15     }
16
17     manager.close();
18     factory.close();
19   }
20 }
```

Código Java 4.39: TesteBuscaFuncionarioDepartamento.java

Paginação

Supondo que exista uma grande quantidade de livros cadastrados no banco de dados, buscar todos os livros sem nenhum filtro vai sobrecarregar o tráfego da rede e a memória utilizada pela aplicação. Nesses casos, podemos aplicar o conceito de **paginação** para obter os livros aos poucos. A paginação do resultado de uma consulta é realizada através dos métodos **setFirstResult()** e **setMaxResults()**.

```
1 TypedQuery<Livro> query =
2   manager.createQuery("select livro from Livro livro", Livro.class);
3 query.setFirstResult(10);
4 query.setMaxResults(20);
5 List<Livro> livros = query.getResultList();
```

Código Java 4.40: Definindo a paginação do resultado da consulta

Exercícios de Fixação

19 Teste o recurso de paginação das consultas. Adicione a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-JPQL**.

```
1 public class TesteBuscaPaginada {
```

```
2   public static void main(String[] args) {
3     EntityManagerFactory factory = Persistence
4         .createEntityManagerFactory("K21_jpql_pu");
5     EntityManager manager = factory.createEntityManager();
6
7     TypedQuery<Livro> query = manager.createQuery("select livro from Livro livro",
8         Livro.class);
9
10    query.setFirstResult(2);
11    query.setMaxResults(3);
12    List<Livro> livros = query.getResultList();
13
14    for (Livro livro : livros) {
15      System.out.println("Livro: " + livro.getNome());
16    }
17
18    manager.close();
19    factory.close();
20  }
21 }
```

Código Java 4.41: TesteBuscaPaginada.java

O Problema do $N+1$

Em alguns casos, o comportamento LAZY pode gerar um número excessivo de consultas, comprometendo o desempenho da aplicação. Por exemplo, considere as entidades `Departamento` e `Funcionario`.

```
1  @Entity
2  public class Funcionario {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    // GETTERS E SETTERS
10 }
```

Código Java 4.42: Funcionario.java

```
1  @Entity
2  public class Departamento {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @OneToMany
10   private List<Funcionario> funcionarios;
11
12   // GETTERS E SETTERS
13 }
```

Código Java 4.43: Departamento.java

Em muitos dos casos, quando buscamos uma lista de departamentos, não precisamos dos dados dos seus respectivos funcionários. Assim, optamos pelo comportamento LAZY para o relacionamento entre departamentos e funcionários.

No entanto, em alguns casos, estaremos sim interessados nos dados dos departamentos e dos seus funcionários. Podemos realizar uma consulta para recuperar a lista de departamentos. Além disso, como o comportamento escolhido foi LAZY, uma consulta adicional será realizada pelo provedor para cada departamento a fim de recuperar os seus funcionários.

No total, teremos $N+1$ consultas, onde N é o número de departamentos obtidos na primeira consulta. Note que se o relacionamento entre departamentos e funcionários fosse EAGER então apenas uma consulta seria necessária nesses casos.

Para solucionar esse problema, podemos utilizar o comando `left join fetch` na consulta que buscaria os departamentos.

```
1 SELECT DISTINCT(d) FROM Departamento d LEFT JOIN FETCH d.funcionarios
```

Aplicando o comando `left join fetch`, os departamentos e seus respectivos funcionários são recuperados em apenas uma consulta.

Exercícios de Fixação

20 Faça uma consulta para obter todos os autores e seus respectivos livros. No pacote `br.com.k19.testes` do projeto **K19-JPQL**, crie a classe `ListaAutoresELivros`.

```
1  public class ListaAutoresELivros {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      Query query = manager.createQuery("select a from Autor a");
8      List<Autor> autores = query.getResultList();
9      System.out.println();
10
11     for (Autor autor : autores) {
12       System.out.println("Autor: " + autor.getNome());
13       Collection<Livro> livros = autor.getLivros();
14
15       for (Livro livro : livros) {
16         System.out.println("Livro: " + livro.getNome());
17       }
18       System.out.println();
19     }
20
21     manager.close();
22     factory.close();
23   }
24 }
```

Código Java 4.45: ListaAutoresELivros.java

Execute e observe, no console, a quantidade de consultas realizadas.

21 Altere a classe `ListaAutoresELivros` para reduzir a quantidade de consultas ao banco de dados. Aplique o comando `fetch join` na consulta de autores.

```java
public class ListaAutoresELivros {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_jpql_pu");
    EntityManager manager = factory.createEntityManager();

    Query query = manager.createQuery(
                "select distinct(a) from Autor a left join fetch a.livros");
    List<Autor> autores = query.getResultList();
    System.out.println();

    for (Autor autor : autores) {
      System.out.println("Autor: " + autor.getNome());
      Collection<Livro> livros = autor.getLivros();

      for (Livro livro : livros) {
        System.out.println("Livro: " + livro.getNome());
      }
      System.out.println();
    }

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

Código Java 4.46: ListaAutoresELivros.java

Execute e observe, no console, que apenas uma consulta é realizada.

Operações em Lote (Bulk Operations)

Para atualizar ou remover registros das tabelas do banco de dados, a abordagem mais comum é realizar uma consulta, trazendo para a memória os objetos referentes aos registros que devem ser modificados ou removidos.

```java
Pessoa p = manager.find(Pessoa.class, 1L);
p.setNome("Rafael Cosentino");
```

```java
Pessoa p = manager.find(Pessoa.class, 1L);
manager.remove(p);
```

Em alguns casos, essa abordagem não é a mais eficiente. Por exemplo, suponha que uma aplicação que controla os produtos de uma loja virtual necessite atualizar os preços de todos os produtos com uma taxa fixa.

```java
TypedQuery<Produto> query = manager.createNamedQuery("Produto.findAll");
List<Produto> produtos = query.getResultList();
for(Produto p : produtos) {
  double preco = p.getPreco();
  p.setPreco(preco * 1.1); // 10% de aumento
}
```

Todos os produtos seriam carregados e isso sobrecarregaria a rede, pois uma grande quantidade de dados seria transferida do banco de dados para a aplicação, e a memória, pois muitos objetos seriam criados pelo provedor JPA.

A abordagem mais eficiente nesse caso é realizar uma operação em lote (bulk operation). Uma operação em lote é executada no banco de dados sem transferir os dados dos registros para a memória da aplicação.

```
Query query = manager.createQuery("UPDATE Produto p SET p.preco = p.preco * 1.1");
query.executeUpdate();
```

A mesma estratégia pode ser adotada quando diversos registros devem ser removidos. Não é necessário carregar os objetos na memória para removê-los, basta realizar uma operação em lote.

```
Query query = manager.createQuery("DELETE Produto p WHERE p.preco < 50");
query.executeUpdate();
```

Exercícios de Fixação

22 No pacote `br.com.k19.modelo` do projeto **K19-JPQL**, adicione a seguinte classe:

```
@Entity
public class Produto {

  @Id @GeneratedValue
  private Long id;

  private String nome;

  private Double preco;

  // GETTERS E SETTERS
}
```

Código Java 4.52: Produto.java

23 Adicione alguns produtos no banco de dados. Crie a seguinte classe em um pacote chamado `br.com.k19.testes` do projeto **K19-JPQL**.

```
public class AdicionaProdutos {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_jpql_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    for (int i = 0; i < 100; i++) {
      Produto p = new Produto();
      p.setNome("produto " + i);
      p.setPreco(i * 10.0);
      manager.persist(p);
    }

    manager.getTransaction().commit();

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

Código Java 4.53: AdicionaProdutos.java

Execute e verifique a tabela produto na base de dados K21_jpql.

24 Faça uma operação em lote para atualizar o preço de todos os produtos de acordo com uma taxa fixa. Crie a seguinte classe em um pacote chamado `br.com.k19.testes` do projeto **K19-JPQL**.

```
public class AumentaPreco {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_jpql_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    Query query = manager
        .createQuery("UPDATE Produto p SET p.preco = p.preco * 1.1");
    query.executeUpdate();

    manager.getTransaction().commit();

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

Código Java 4.54: AumentaPreco.java

Execute e verifique a tabela produto na base de dados K21_jpql.

Observe também o console do Eclipse. Nenhum select é realizado.

25 Faça uma operação em lote para remover todos os produtos com preço menor do que um valor fixo. Crie a seguinte classe em um pacote chamado `br.com.k19.testes` do projeto **K19-JPQL**.

```
public class RemoveProdutos {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_jpql_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    Query query = manager
        .createQuery("DELETE Produto p WHERE p.preco < 50");
    query.executeUpdate();

    manager.getTransaction().commit();

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

Código Java 4.55: RemoveProdutos.java

Execute e verifique a tabela produto na base de dados K21_jpql.

Observe também o console do Eclipse. Nenhum select é realizado.

Operadores

As consultas em JPQL utilizam alguns tipos de operadores, que veremos a seguir.

Condicionais

- Menor (<)

```
1 String query = "SELECT p FROM Pessoa p WHERE p.idade < :idade";
```

Código Java 4.56: Seleciona as pessoas com idade menor do que a idade especificada

- Maior (>)

```
1 String query = "SELECT p FROM Pessoa p WHERE p.idade > :idade";
```

Código Java 4.57: Seleciona as pessoas com idade maior do que a idade especificada

- Menor Igual (<=)

```
1 String query = "SELECT p FROM Pessoa p WHERE p.idade <= :idade";
```

Código Java 4.58: Seleciona as pessoas com idade menor do que ou igual a idade especificada

- Maior Igual (>=)

```
1 String query = "SELECT p FROM Pessoa p WHERE p.idade >= :idade";
```

Código Java 4.59: Seleciona as pessoas com idade maior do que ou igual a idade especificada

- Igual (=)

```
1 String query = "SELECT p FROM Pessoa p WHERE p.idade = :idade";
```

Código Java 4.60: Seleciona as pessoas com idade igual a idade especificada

- Diferente (<>)

```
1 String query = "SELECT p FROM Pessoa p WHERE p.idade <> :idade";
```

Código Java 4.61: Seleciona as pessoas com idade diferente da idade especificada

- IS NULL

```
1 String query = "SELECT p FROM Pessoa p WHERE p.nome IS NULL";
```

Código Java 4.62: Seleciona as pessoas cujos nomes são nulos

- IS NOT NULL

```
1 String query = "SELECT p FROM Pessoa p WHERE p.nome IS NOT NULL";
```

Código Java 4.63: Seleciona as pessoas cujos nomes são não nulos

- BETWEEN

```
1 String query = "SELECT p FROM Pessoa p WHERE p.idade BETWEEN :minimo AND :maximo";
```

Código Java 4.64: Seleciona as pessoas cujas idades estão no intervalo especificado

- NOT BETWEEN

```
1 String query = "SELECT p FROM Pessoa p WHERE p.idade NOT BETWEEN :minimo AND :maximo"↩
      ;
```

Código Java 4.65: Seleciona as pessoas cujas idades estão fora do intervalo especificado

- AND

```
1 String query = "SELECT p FROM Pessoa p WHERE p.nome IS NOT NULL AND p.idade >= :idade↩
      ";
```

*Código Java 4.66: Seleciona as pessoas cujas nomes são não nulos **e** cujas idades são maiores do que a idade especificada*

- OR

```
1 String query = "SELECT p FROM Pessoa p WHERE p.nome IS NOT NULL OR p.idade >= :idade"↩
      ;
```

*Código Java 4.67: Seleciona as pessoas cujas nomes são não nulos **ou** cujas idades são maiores do que a idade especificada*

- NOT

```
1 String query = "SELECT p FROM Pessoa p WHERE NOT (p.idade >= :idade)";
```

Código Java 4.68: Seleciona as pessoas com idade menor do que a idade especificada

- MEMBER OF

```
1 String query = "SELECT f FROM Funcionario f WHERE f MEMBER OF f.empresa.diretoria";
```

Código Java 4.69: Seleciona os funcionários que fazem parte da diretoria da empresa

- NOT MEMBER OF

```
1 String query = "SELECT f FROM Funcionario f WHERE f NOT MEMBER OF f.empresa.diretoria↩
      ";
```

Código Java 4.70: Seleciona os funcionários que não fazem parte da diretoria da empresa

- IS EMPTY

```
1 String query = "SELECT a FROM Autor a WHERE a.livros IS EMPTY";
```

Código Java 4.71: Seleciona os autores que possuem não possuem nenhum livro

- IS NOT EMPTY

```
1 String query = "SELECT a FROM Autor a WHERE a.livros IS NOT EMPTY";
```

Código Java 4.72: Seleciona os autores que possuem possuem ao menos um livro

- EXISTS

```
String query = "SELECT d FROM Departamento d WHERE EXISTS
  (SELECT f FROM FUNCIONARIO f WHERE f.departamento = d)";
```

Código Java 4.73: Seleciona os departamentos que possuem ao menos um funcionário

- NOT EXISTS

```
String query = "SELECT d FROM Departamento d WHERE NOT EXISTS
  (SELECT f FROM FUNCIONARIO f WHERE f.departamento = d)";
```

Código Java 4.74: Seleciona os departamentos que não possuem nenhum um funcionário

- LIKE

```
String query = "SELECT a FROM Autor a WHERE a.nome LIKE Patrick%";
```

Código Java 4.75: Seleciona os autores com nome dentro do padrão especificado

- NOT LIKE

```
String query = "SELECT a FROM Autor a WHERE a.nome NOT LIKE Patrick%";
```

Código Java 4.76: Seleciona os autores com nome fora do padrão especificado

- IN

```
String query = "SELECT a FROM Autor a WHERE a.nome IN ('Patrick Cullen', 'Fraser ↩
    Seitel')";
```

Código Java 4.77: Seleciona os autores com nome igual a um dos nomes especificados

- NOT IN

```
String query = "SELECT a FROM Autor a WHERE
  a.nome NOT IN ('Patrick Cullen', 'Fraser Seitel')";
```

Código Java 4.78: Seleciona os autores com nome diferente dos nomes especificados

Escalares

- ALL

```
String query = "SELECT livro FROM Livro livro WHERE
  livro.preco >= ALL(SELECT livro.preco FROM Livro livro)";
```

Código Java 4.79: Seleciona os livros mais caros

- ANY

```
String query = "SELECT livro FROM Livro livro WHERE
  livro.titulo = ANY(SELECT livro2.titulo FROM Livro livro2 WHERE livro <> livro2)";
```

Código Java 4.80: Seleciona os livros que possuem títulos em comum

- SOME

```
1 String query = "SELECT livro FROM Livro livro WHERE
2   livro.titulo = SOME(SELECT livro2.titulo FROM Livro livro2 WHERE livro <> livro2)";
```

Código Java 4.81: Seleciona os livros que possuem títulos em comum

Agregadores

- AVG

```
1 String query = "SELECT AVG(livro.preco) FROM Livro livro";
```

Código Java 4.82: Seleciona a média dos preços dos livros

- SUM

```
1 String query = "SELECT SUM(livro.preco) FROM Livro livro";
```

Código Java 4.83: Seleciona a soma dos preços dos livros

- MIN

```
1 String query = "SELECT MIN(livro.preco) FROM Livro livro";
```

Código Java 4.84: Seleciona o preço do livro mais barato

- MAX

```
1 String query = "SELECT MAX(livro.preco) FROM Livro livro";
```

Código Java 4.85: Seleciona o preço do livro mais caro

- COUNT

```
1 String query = "SELECT COUNT(livro) FROM Livro livro";
```

Código Java 4.86: Seleciona a quantidade de livros cadastrados

Funções

ABS: Calcula o valor absoluto de um número.

CONCAT: Concatena strings.

CURRENT_DATE: Recupera a data atual.

CURRENT_TIME: Recupera o horário atual.

CURRENT_TIMESTAMP: Recupera a data e o horário atuais.

LENGTH: Calcula o número de caracteres de uma string.

LOCATE: Localiza uma string dentro de outra.

LOWER: Deixa as letras de uma string minúsculas.

MOD: Calcula o resto da divisão entre dois números.

SIZE: Calcula o número de elementos de uma coleção.

SQRT: Calcula a raiz quadrada de um número.

SUBSTRING: Recupera um trecho de uma string.

UPPER: Deixa as letras de uma string maiúsculas.

TRIM: Elimina eventuais espaços no início e no fim de uma string.

ORDER BY

Podemos ordenar o resultado de uma consulta através do operador **ORDER BY** com as opções **ASC** ou **DESC**.

```
1 String query = "SELECT livro FROM Livro livro ORDER BY livro.preco ASC";
```

Exemplos

1. Suponha que seja necessário selecionar os livros mais baratos. Em outras palavras, devemos selecionar os livros tais que não exista nenhum outro livro com preço menor.

```
1 SELECT livro1
2 FROM Livro livro1
3 WHERE NOT EXISTS (
4   SELECT livro2
5   FROM Livro livro2
6   WHERE livro1.preco > livro2.preco
7 )
```

Código Java 4.88: Seleciona os livros mais baratos

2. Suponha que seja necessário selecionar os livros mais baratos de determinado autor.

```
1  SELECT livro1
2  FROM Livro livro1, Autor autor
3  WHERE autor1.id = :id and
4  livro1 MEMBER OF autor.livros and
5  NOT EXISTS (
6    SELECT livro2
7    FROM Livro livro2
8    WHERE livro2 MEMBER OF autor.livros and
9    livro1.preco > livro2.preco
10 )
```

Código Java 4.89: Seleciona os livros mais baratos de um determinado autor

3. Suponha que seja necessário listar os livros em ordem decrescente em relação aos preços.

```
1 SELECT livro FROM Livro livro ORDER BY livro.preco DESC
```

Código Java 4.90: Seleciona os livros em ordem decrescente de preço

4. Suponha que seja necessário selecionar os autores com mais livros.

```
SELECT autor1
FROM Autor autor1
WHERE NOT EXISTS (
  SELECT autor2
  FROM Autor autor2
  WHERE SIZE(autor2.livros) > SIZE(autor1.livros)
)
```

Código Java 4.91: Seleciona os autores com mais livros

Referências

Para conhecer mais sobre a sintaxe do JPQL, consulte:

http://docs.oracle.com/cd/E28613_01/apirefs.1211/e24396/ejb3_langref.html

Consultas Nativas

Os provedores JPA também devem oferecer o suporte à consultas nativas, ou seja, consultas definidas em SQL. Contudo, devemos lembrar que consultas definidas em SQL são específicas de um determinado banco de dados e eventualmente podem não funcionar em bancos de dados diferentes.

```
String sql = "SELECT * FROM Produto";
Query nativeQuery = manager.createNativeQuery(sql, Produto.class);
List<Produto> produtos = nativeQuery.getResultList();
```

Código Java 4.92: Selecionando os produtos com uma consulta nativa

Exercícios de Fixação

26 Testes as consultas nativas. Acrescente a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-JPQL**.

```
public class TesteConsultaNativas {
  public static void main(String[] args) {
    EntityManagerFactory factory =
      Persistence.createEntityManagerFactory("K21_jpql_pu");

    EntityManager manager = factory.createEntityManager();

    String sql = "SELECT * FROM Produto";
    Query nativeQuery = manager.createNativeQuery(sql, Produto.class);
    List<Produto> produtos = nativeQuery.getResultList();

    for (Produto p : produtos) {
      System.out.println(p.getNome());
    }

    manager.close();
```

```
17      factory.close();
18    }
19  }
```

Código Java 4.93: TesteConsultaNativas.java

Stored Procedures

A partir da versão 2.1 da especificação JPA, as stored procedures dos bancos de dados podem ser utilizadas mais facilmente. No exemplo abaixo, uma stored procedure foi mapeada com a anotação `@NamedStoredProcedureQuery`. O atributo `name` define o nome da stored procedure dentro da aplicação. O atributo `resultClasses` define a classe que modela o resultado da stored procedure. O atributo `procedureName` define o nome da stored procedure no banco de dados. O atributo `parameters` define os parâmetros da stored procedure.

```
1  @NamedStoredProcedureQuery(
2      name = "BuscaProdutos",
3      resultClasses = Produto.class,
4      procedureName = "BUSCA_PRODUTOS",
5      parameters = {
6          @StoredProcedureParameter(
7            mode=ParameterMode.IN,
8            name="PRECO_MINIMO",
9            type=Double.class)
10     }
11 )
12 @Entity
13 public class Produto {
14   ...
15 }
```

Código Java 4.94: Produto.java

A stored procedure mapeada anteriormente pode ser chamada com o código a seguir.

```
1 StoredProcedureQuery query =
2   manager.createNamedStoredProcedureQuery("BuscaProdutos");
3 query.setParameter("PRECO_MINIMO", 1000.0);
4 List<Produto> produtos = query.getResultList();
```

Exercícios de Fixação

27 Através do MySQL Workbench, defina uma stored procedure na base dados **K21_jpql** com o seguinte código.

```
1  DROP PROCEDURE IF EXISTS BUSCA_PRODUTOS;
2
3  DELIMITER //
4
5  CREATE PROCEDURE BUSCA_PRODUTOS(IN PRECO_MINIMO DOUBLE)
6  BEGIN
7    SELECT * FROM Produto WHERE preco > PRECO_MINIMO;
8  END //
9
```

```
10 DELIMITER ;
```

28 Altere a classe **Produto** do projeto **K19-JPQL**. Adicione o mapeamento da stored procedure criada no exercício anterior.

```
1  @NamedStoredProcedureQuery(
2      name = "BuscaProdutos",
3      resultClasses = Produto.class,
4      procedureName = "BUSCA_PRODUTOS",
5      parameters = {
6          @StoredProcedureParameter(
7            mode=ParameterMode.IN,
8            name="PRECO_MINIMO",
9            type=Double.class)
10     }
11 )
12 @Entity
13 public class Produto {
14   ...
15 }
```

Código Java 4.96: Produto.java

29 Execute a stored procedure mapeada no exercício anterior. Crie a seguinte classe em um pacote chamado `br.com.k19.testes` do projeto **K19-JPQL**.

```
1  public class TestaStoredProcedure {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_jpql_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      StoredProcedureQuery query =
8        manager.createNamedStoredProcedureQuery("BuscaProdutos");
9      query.setParameter("PRECO_MINIMO", 1000.0);
10     List<Produto> produtos = query.getResultList();
11
12     for (Produto produto : produtos) {
13       System.out.println("ID: " + produto.getId());
14       System.out.println("NOME: " + produto.getNome());
15       System.out.println("PREÇO: " + produto.getPreco());
16       System.out.println("----------------------------------");
17     }
18
19     manager.close();
20     factory.close();
21   }
22 }
```

Código Java 4.97: TestaStoredProcedure.java

Execute a classe `TestaStoredProcedure`.

CRITERIA

CAPÍTULO 5

O primeiro mecanismo definido pela especificação JPA 2 para realizar consultas orientadas a objetos utiliza a linguagem JPQL. O segundo mecanismo utiliza um conjunto de classes e interfaces e funciona basicamente como uma biblioteca. O nome dessa segunda abordagem é Criteria.

Necessidade

A primeira questão a ser discutida nesse momento é a necessidade de dois mecanismos para definição de consultas. Um mecanismo só não seria suficiente?

Teoricamente, qualquer consulta definida com JPQL pode ser definida com Criteria e vice-e-versa. Contudo, algumas consultas são mais facilmente definidas em JPQL enquanto outras mais facilmente definidas em Criteria.

As consultas que não dependem de informações externas (por exemplo, buscar todos os livros cadastrados no banco de dados sem nenhum filtro) são mais facilmente definidas em JPQL.

As consultas que dependem de informações externas fixas (por exemplo, buscar todos os livros cadastrados no banco de dados que possuam preço maior do que um valor definido pelos usuários) também são mais facilmente criadas em JPQL utilizando os parâmetros de consulta.

Agora, as consultas que dependem de informações externas não fixas são mais facilmente definidas em Criteria. Por exemplo, considere uma busca avançada por livros na qual há muitos campos opcionais. Essa consulta depende de informações externas não fixas já que nem todo campo fará parte da consulta todas as vezes que ela for executada.

Estrutura Geral

As duas interfaces principais de Criteria são: **CriteriaBuilder** e **CriteriaQuery**. As consultas em Criteria são construídas por um Criteria Builder que é obtido através de um Entity Manager.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
```

Código Java 5.1: Obtendo um Criteria Builder a partir de um Entity Manager

A definição de uma consulta em Criteria começa na chamada do método `createQuery()` de um Criteria Builder. O parâmetro desse método indica o tipo esperado do resultado da consulta.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
2 CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
```

Código Java 5.2: Iniciando a construção de uma consulta em Criteria

Para definir quais dados devem ser considerados na consulta, devemos utilizar o método **from()** da Criteria Query. Este método devolve uma raiz do espaço de dados considerado pela pesquisa.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
2 CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
3 // DEFININDO O ESPAÇO DE DADOS DA CONSULTA
4 Root<Autor> a = c.from(Autor.class);
```

Código Java 5.3: Definindo o espaço de dadas da consulta

Para definir o que queremos selecionar do espaço de dados da consulta, devemos utilizar o método **select()** da Criteria Query.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
2 CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
3 // DEFININDO O ESPAÇO DE DADOS DA CONSULTA
4 Root<Autor> a = c.from(Autor.class);
5 // SELECIONANDO UMA RAIZ DO ESPAÇO DE DADOS
6 c.select(a);
```

Código Java 5.4: Definindo o que deve ser selecionado na consulta

Com a consulta em Criteria definida, devemos invocar um Entity Manager para poder executá-la da seguinte forma.

```
1 TypedQuery<Autor> query = manager.createQuery(c);
2 List<Autor> autores = query.getResultList();
```

Código Java 5.5: Executando a consulta

Exercícios de Fixação

1 Crie um projeto no eclipse chamado **K19-Criteria**. Copie a pasta **lib** do projeto **K19-JPA2-Hibernate** para o projeto **K19-Criteria**. Depois adicione os jars dessa pasta no classpath desse novo projeto.

2 Abra o MySQL Workbench e apague a base de dados **K21_criteria_bd** se existir. Depois crie a base de dados **K21_criteria_bd**.

3 Copie a pasta **META-INF** do projeto **K19-K19-JPA2-Hibernate** para dentro da pasta **src** do projeto **K19-Criteria**. Altere o arquivo **persistence.xml** do projeto **K19-Criteria**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence
3    version="2.1"
4    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
7      http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd ">
8
9    <persistence-unit name="K21_criteria_pu" transaction-type="RESOURCE_LOCAL">
10     <provider>org.hibernate.ejb.HibernatePersistence</provider>
```

```
11      <properties>
12        <property
13          name="hibernate.dialect"
14          value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15
16        <property
17          name="hibernate.hbm2ddl.auto"
18          value="update" />
19
20        <property
21          name="javax.persistence.jdbc.driver"
22          value="com.mysql.jdbc.Driver" />
23
24        <property
25          name="javax.persistence.jdbc.user"
26          value="root" />
27
28        <property
29          name="javax.persistence.jdbc.password"
30          value="root" />
31
32        <property
33          name="javax.persistence.jdbc.url"
34          value="jdbc:mysql://localhost:3306/K21_criteria_bd"/>
35      </properties>
36    </persistence-unit>
37  </persistence>
```

Código XML 5.1: persistence.xml

4 Crie um pacote chamado `br.com.k19.modelo` no projeto **K19-Criteria** e adicione as seguintes classes:

```
1  @Entity
2  public class Autor {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    @ManyToMany
10   private Collection<Livro> livros = new ArrayList<Livro>();
11
12   // GETTERS E SETTERS
13 }
```

Código Java 5.6: Autor.java

```
1  @Entity
2  public class Livro {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    private Double preco;
10
11   // GETTERS E SETTERS
12 }
```

Código Java 5.7: Livro.java

5 Carregue o banco de dados com as informações de alguns livros e autores. Adicione a seguinte classe em um novo pacote chamado `br.com.k19.testes` no projeto **K19-Criteria**. **Você pode copiar a classe PopulaBanco criada no projeto K19-JPQL e alterar a unidade de persistência**.

```
1  public class PopulaBanco {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Livro livro1 = new Livro();
10     livro1.setNome("The Battle for Your Mind");
11     livro1.setPreco(20.6);
12     manager.persist(livro1);
13
14     Livro livro2 = new Livro();
15     livro2.setNome("Differentiate or Die");
16     livro2.setPreco(15.8);
17     manager.persist(livro2);
18
19     Livro livro3 = new Livro();
20     livro3.setNome("How to Transform Your Ideas");
21     livro3.setPreco(32.7);
22     manager.persist(livro3);
23
24     Livro livro4 = new Livro();
25     livro4.setNome("Digital Fortress");
26     livro4.setPreco(12.9);
27     manager.persist(livro4);
28
29     Livro livro5 = new Livro();
30     livro5.setNome("Marketing in an Era of Competition, Change and Crisis");
31     livro5.setPreco(26.8);
32     manager.persist(livro5);
33
34     Autor autor1 = new Autor();
35     autor1.setNome("Patrick Cullen");
36     autor1.getLivros().add(livro2);
37     autor1.getLivros().add(livro4);
38     manager.persist(autor1);
39
40     Autor autor2 = new Autor();
41     autor2.setNome("Fraser Seitel");
42     autor2.getLivros().add(livro3);
43     manager.persist(autor2);
44
45     Autor autor3 = new Autor();
46     autor3.setNome("Al Ries");
47     autor3.getLivros().add(livro1);
48     manager.persist(autor3);
49
50     Autor autor4 = new Autor();
51     autor4.setNome("Jack Trout");
52     autor4.getLivros().add(livro1);
53     autor4.getLivros().add(livro2);
54     autor4.getLivros().add(livro5);
55     manager.persist(autor4);
56
57     Autor autor5 = new Autor();
58     autor5.setNome("Steve Rivkin");
59     autor5.getLivros().add(livro2);
60     autor5.getLivros().add(livro3);
61     autor5.getLivros().add(livro5);
62     manager.persist(autor5);
63
64     manager.getTransaction().commit();
65
```

```
66      manager.close();
67      factory.close();
68    }
69  }
```

Código Java 5.8: PopulaBanco.java

6 Teste a os recursos de Criteria criando uma consulta para recuperar todos os livros cadastrados no banco de dados. Crie a seguinte classe no pacote `br.com.k19.testes`.

```
1  public class TesteCriteria {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      CriteriaBuilder cb = manager.getCriteriaBuilder();
8      CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
9      Root<Livro> l = c.from(Livro.class);
10     c.select(l);
11
12     TypedQuery<Livro> query = manager.createQuery(c);
13     List<Livro> livros = query.getResultList();
14
15     for (Livro livro : livros) {
16       System.out.println(livro.getNome());
17     }
18   }
19 }
```

Código Java 5.9: TesteCriteria.java

Exercícios Complementares

1 Crie uma consulta utilizando Criteria para recuperar todos os autores. Adicione uma classe para essa tarefa no pacote `br.com.k19.testes` do projeto **K19-Criteria**.

Tipos de Resultados

Lista de Entidades

O resultado de uma consulta em Criteria pode ser uma lista com os objetos de uma entidade que são compatíveis com os filtros da pesquisa.

```
1 CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2 Root<Livro> livro = c.from(Livro.class);
3 c.select(livro);
```

Código Java 5.11: Uma consulta em Criteria

O resultado da consulta acima é uma lista com todos os objetos da classe `Livro`.

`avg()`	Calcula a média de um conjunto de números
`count()`	Contabiliza o número de resultados
`max()` e `greatest()`	Recupera o maior elemento um conjunto de números
`min()` e `least()`	Recupera o menor elemento um conjunto de números
`sum()`, `sumAsLong()` e `sumAsDouble()`	Calcula a soma de um conjunto de números

Tabela 5.1: Funções de agregação.

Lista de Objetos Comuns

Muitas vezes, não queremos selecionar todas as informações dos objetos pesquisados. Por exemplo, suponha que seja necessário gerar uma lista com os nomes dos livros cadastrados no banco de dados. Através do método `select()` podemos definir o que deve ser recuperado do banco de dados.

```
1 CriteriaQuery<String> c = cb.createQuery(String.class);
2 Root<Livro> livro = c.from(Livro.class);
3 c.select(livro.<String>get("nome"));
```

Código Java 5.12: Selecionando apenas os nomes dos livros

A consulta acima recupera apenas os nomes dos livros. O resultado dessa pesquisa é uma lista de strings.

Valores Únicos

Algumas consultas possuem como resultado valores únicos. Por exemplo, suponha que queremos obter a média dos preços dos livros cadastrados no banco de dados ou a quantidade de livros de uma determinada editora. Nesse tipo de consulta, devemos apenas trazer do banco de dados um valor único calculado no próprio banco.

```
1 CriteriaQuery<Double> c = cb.createQuery(Double.class);
2 Root<Livro> l = c.from(Livro.class);
3 c.select(cb.avg(l.<Double>get("preco")));
```

Código Java 5.13: Obtendo a média dos preços dos livros

A consulta acima devolve a média dos preços dos livros cadastrados no banco de dados. Nessa consulta, foi utilizada uma função de agregação. Veja a lista dessas funções:

Resultados Especias

Podemos selecionar múltiplos atributos de uma entidade em uma consulta em Criteria. Por exemplo, podemos montar uma listagem com os nomes e preços dos livros cadastrados no banco de dados. Para selecionar múltiplos atributos, devemos utilizar o método `multiselect()`.

```
1 CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
2 Root<Livro> l = c.from(Livro.class);
3 c.multiselect(l.<String>get("nome"), l.<Double>get("preco"));
```

Código Java 5.14: Selecionando os nomes e os preços dos livros

O resultado da consulta acima é uma lista de array de Object. Para manipular esse resultado, temos que utilizar a posição dos dados dos arrays da lista.

```
TypedQuery<Object[]> query = manager.createQuery(c);
List<Object[]> resultado = query.getResultList();

for (Object[] registro : resultado) {
  System.out.println("Livro: " + registro[0]);
  System.out.println("Preço: " + registro[1]);
}
```

Código Java 5.15: Manipulando os resultados da consulta

Também podemos utilizar a interface `Tuple` para não trabalhar com posicionamento de dados em arrays. Nessa abordagem, devemos aplicar "apelidos" para os itens selecionados através do método `alias()`.

```
CriteriaQuery<Tuple> c = cb.createQuery(Tuple.class);
Root<Livro> l = c.from(Livro.class);
c.multiselect(l.<String>get("nome").alias("livro.nome"), l.<Double>get("preco").alias↩
    ("livro.preco"));

TypedQuery<Tuple> query = manager.createQuery(c);
List<Tuple> resultado = query.getResultList();

for (Tuple registro : resultado) {
  System.out.println("Livro: " + registro.get("livro.nome"));
  System.out.println("Preço: " + registro.get("livro.preco"));
}
```

Código Java 5.16: Usando a interface `Tuple` *para manipular os dados da consulta*

Exercícios de Fixação

7 Recupere uma listagem com os nomes dos livros cadastrados no banco de dados. Adicione a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-Criteria**.

```
public class ListaNomesDosLivros {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_criteria_pu");
    EntityManager manager = factory.createEntityManager();

    CriteriaBuilder cb = manager.getCriteriaBuilder();
    CriteriaQuery<String> c = cb.createQuery(String.class);
    Root<Livro> livro = c.from(Livro.class);
    c.select(livro.<String>get("nome"));

    TypedQuery<String> query = manager.createQuery(c);
    List<String> nomes = query.getResultList();

    for (String nome : nomes) {
      System.out.println("Livro: " + nome);
    }

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

Código Java 5.17: ListaNomesDosLivros.java

8 Recupere a média dos valores dos livros cadastrados no banco de dados. Adicione a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-Criteria**.

```
public class CalculaMediaDosPrecosDosLivros {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_criteria_pu");
    EntityManager manager = factory.createEntityManager();

    CriteriaBuilder cb = manager.getCriteriaBuilder();
    CriteriaQuery<Double> c = cb.createQuery(Double.class);
    Root<Livro> l = c.from(Livro.class);
    c.select(cb.avg(l.<Double>get("preco")));

    TypedQuery<Double> query = manager.createQuery(c);
    Double media = query.getSingleResult();

    System.out.println("Média: " + media);

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

Código Java 5.18: CalculaMediaDosPrecosDosLivros.java

9 Recupere os nomes e os preços dos livros cadastrados no banco de dados. Adicione a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-Criteria**.

```
public class ConsultaNomePrecoDosLivros {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_criteria_pu");
    EntityManager manager = factory.createEntityManager();

    CriteriaBuilder cb = manager.getCriteriaBuilder();
    CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
    Root<Livro> l = c.from(Livro.class);
    c.multiselect(l.<String>get("nome"), l.<Double>get("preco"));

    TypedQuery<Object[]> query = manager.createQuery(c);
    List<Object[]> resultado = query.getResultList();

    for (Object[] registro : resultado) {
      System.out.println("Livro: " + registro[0]);
      System.out.println("Preço: " + registro[1]);
    }

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

Código Java 5.19: ConsultaNomePrecoDosLivros.java

10 Altere a classe do exercício anterior para que ela utilize a interface `Tuple`.

```
public class ConsultaNomePrecoDosLivros {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_criteria_pu");
    EntityManager manager = factory.createEntityManager();

```

```
7       CriteriaBuilder cb = manager.getCriteriaBuilder();
8       CriteriaQuery<Tuple> c = cb.createQuery(Tuple.class);
9       Root<Livro> l = c.from(Livro.class);
10      c.multiselect(l.<String>get("nome").alias("livro.nome"), l.<Double>get("preco").↩
        alias("livro.preco"));
11
12      TypedQuery<Tuple> query = manager.createQuery(c);
13      List<Tuple> resultado = query.getResultList();
14
15      for (Tuple registro : resultado) {
16        System.out.println("Livro: " +
17        registro.get("livro.nome")); System.out.println("Preço: " + registro.get("livro↩
        .preco"));
18      }
19
20      manager.close();
21      factory.close();
22    }
23  }
```

Código Java 5.20: ConsultaNomePrecoDosLivros.java

Filtros e Predicados

Podemos definir filtros para as consultas através da criação de **Predicates** e do método **where()** de uma Criteria Query. Os predicados são condições que devem ser satisfeitas para que uma informação seja adicionada no resultado de uma consulta. Eles são criados por um Criteria Builder.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
2 CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
3 Root<Autor> a = c.from(Autor.class);
4 c.select(a);
5
6 // CRIANDO UM PREDICATE
7 Predicate predicate = cb.equal(a.get("nome"), "Patrick Cullen");
8 // ASSOCIANDO O PREDICATE A CONSULTA
9 c.where(predicate);
```

Código Java 5.21: Criando um predicado e associando-o a uma consulta

Exercícios de Fixação

11 Teste uma consulta com Criteria criando uma classe para recuperar todos os livros cadastrados no banco de dados filtrando os pelo nome. Crie a seguinte classe no pacote `br.com.k19.testes` do projeto **K19-Criteria**.

```
1  public class TestePredicate {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      CriteriaBuilder cb = manager.getCriteriaBuilder();
8      CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
9      Root<Livro> l = c.from(Livro.class);
10     c.select(l);
11
```

```
12      Predicate predicate = cb.equal(l.get("nome"), "The Battle for Your Mind");
13      c.where(predicate);
14
15      TypedQuery<Livro> query = manager.createQuery(c);
16      List<Livro> livros = query.getResultList();
17
18      for (Livro livro : livros) {
19        System.out.println(livro.getId());
20        System.out.println(livro.getNome());
21        System.out.println(livro.getPreco());
22      }
23    }
24  }
```

Código Java 5.22: TestePredicate.java

Lista de Predicados

- equal()

```
1  cb.equal(livro.get("nome"), "The Battle for Your Mind");
```

Código Java 5.23: Verificando se o nome do livro é igual a "The Battle for Your Mind"

- and()

```
1  cb.and(cb.equal(livro.get("nome"), "Noites"), cb.equal(livro.get("editora"), "Saraiva↩
       "));
```

*Código Java 5.24: Verificando se o livro tem o nome "Noites" **e** sua editora é "Saraiva".*

- or()

```
1  cb.or(cb.equal(livro.get("nome"), "Noites"), cb.equal(livro.get("editora"), "Saraiva"↩
       ));
```

*Código Java 5.25: Verificando se o livro possui o nome "Noites" **ou** se sua editora é "Saraiva".*

- notEqual()

```
1  cb.notEqual(livro.get("nome"), "The Battle for Your Mind");
```

Código Java 5.26: Verificando se o nome do livro é diferente de "The Battle for Your Mind"

- not()

```
1  cb.not(cb.equal(livro.get("nome"), "The Battle for Your Mind"));
```

Código Java 5.27: Verificando se o nome do livro é diferente de "The Battle for Your Mind"

- greaterThan(), gt()

```
1  cb.gt(livro.<Double>get("preco"), 20.0);
```

Código Java 5.28: Verificando se o preço do livro é maior do que 20.0

ou

```
1 cb.greaterThan(livro.<Double>get("preco"), 20.0);
```

Código Java 5.29: Verificando se o preço do livro é maior do que 20.0

- `greaterThanOrEqualTo()`, `ge()`

```
1 cb.ge(livro.<Double>get("preco"), 20.0);
```

Código Java 5.30: Verificando se o preço do livro é maior do que ou igual a 20.0

ou

```
1 cb.greaterThanOrEqualTo(livro.<Double>get("preco"), 20.0);
```

Código Java 5.31: Verificando se o preço do livro é maior do que ou igual a 20.0

- `lessThan()`, `lt()`

```
1 cb.lt(livro.<Double>get("preco"), 20.0);
```

Código Java 5.32: Verificando se o preço do livro é menor do que 20.0

ou

```
1 cb.lessThan(livro.<Double>get("preco"), 20.0);
```

Código Java 5.33: Verificando se o preço do livro é menor do que 20.0

- `lessThanOrEqualTo()`, `le()`

```
1 cb.le(livro.<Double>get("preco"), 20.0);
```

Código Java 5.34: Verificando se o preço do livro é menor do que ou igual a 20.0

ou

```
1 cb.lessThanOrEqualTo(livro.<Double>get("preco"), 20.0);
```

Código Java 5.35: Verificando se o preço do livro é menor do que ou igual a 20.0

- `between()`

```
1 cb.between(livro.<Double>get("preco"), 20.0, 30.0);
```

Código Java 5.36: Verificando se o preço do livro está entre 20.0 e 30.0 inclusive

- `isNull()`

```
1 cb.isNull(livro.get("nome"));
```

Código Java 5.37: Verificando se o nome dos livros são nulos

- `isNotNull()`

```
1 cb.isNotNull(livro.get("nome"));
```

Código Java 5.38: Verificando se o nome dos livros não são nulos

- isEmpty()

```
1 cb.isEmpty(autor.<Collection<Livro>>get("livros"));
```

Código Java 5.39: Verificando se a lista de livros dos autores está vazia

- isNotEmpty()

```
1 cb.isNotEmpty(autor.<Collection<Livro>>get("livros"));
```

Código Java 5.40: Verificando se a lista de livros dos autores não está vazia

- isMember()

```
1 cb.isMember(livro, livro.<Editora>get("editora").<Collection<Livro>>get("maisVendidos↩
      "));
```

Código Java 5.41: Verificando se os livros estão na coleção dos livros mais vendidos de suas respectivas editoras

- isNotMember()

```
1 cb.isNotMember(livro, livro.<Editora>get("editora").<Collection<Livro>>get("↩
      maisVendidos"));
```

Código Java 5.42: Verificando se os livros não estão na coleção dos livros mais vendidos de suas respectivas editoras

- like()

```
1 cb.like(livro.<String>get("nome"), "%Battle%");
```

Código Java 5.43: Determinando se o nome dos livros seguem o padrão especificado

- notLike()

```
1 cb.notLike(livro.<String>get("nome"), "%Battle%");
```

Código Java 5.44: Determinando se o nome dos livros não seguem o padrão especificado

- in()

```
1 cb.in(livro.<String>get("editora")).value("Saraiva").value("Moderna");
```

Código Java 5.45: Verificando se a editora dos livros é igual a uma das editoras especificadas

ou

```
1 livro.<String>get("editora").in("Saraiva","Moderna");
```

Código Java 5.46: Verificando se a editora dos livros é igual a uma das editoras especificadas

Funções

- abs()

```
1 cb.abs(livro.<Double>get("preco"));
```

Código Java 5.47: Obtendo o valor absoluto do preço dos produtos

- concat()

```
1 cb.concat(livro1.<String>get("nome"), livro2.<String>get("nome"));
```

Código Java 5.48: Concatenando o nome de dois livros

- currentDate()

```
1 cb.currentDate();
```

Código Java 5.49: Recuperando a data atual

- currentTime()

```
1 cb.currentTime();
```

Código Java 5.50: Recuperando o horário atual

- currentTimestamp()

```
1 cb.currentTimestamp();
```

Código Java 5.51: Recuperando a data e o horário atual

- length()

```
1 cb.length(livro.<String>get("nome"));
```

Código Java 5.52: Obtendo a quantidade de caracteres dos nomes dos livros

- locate()

```
1 cb.locate(livro1.<String>get("nome"), livro2.<String>get("nome"));
```

Código Java 5.53: Verificando a posição do nome de um autor dentro do nome de outro autor

- lower()

```
1 cb.lower(livro.<String>get("nome"));
```

Código Java 5.54: Colocando o nome dos livros em caixa baixa

- mod()

```
1 cb.mod(autor1.<Integer>get("idade"), autor2.<Integer>get("idade"));
```

Código Java 5.55: Calculando o resto da divisão entre as idades de dois autores

- `size()`

```
1 cb.size(autor.<Collection<Livro>>get("livros"));
```

Código Java 5.56: Recuperando a quantidade de livros dos autores

- `sqrt()`

```
1 cb.sqrt(autor.<Integer>get("idade"));
```

Código Java 5.57: Calculando a raiz quadrada das idades dos autores

- `substring()`

```
1 cb.substring(livro.<String>get("nome"), 3, 5);
```

Código Java 5.58: Obtendo um determinado trecho dos nomes dos livros

- `upper()`

```
1 cb.upper(livro.<String>get("nome"));
```

Código Java 5.59: Colocando os nomes do livros em caixa alta

- `trim()`

```
1 cb.trim(livro.<String>get("nome"));
```

Código Java 5.60: Removendo eventuais espaços do início e do final do nome dos livros

Ordenação

Os resultados das consultas em Criteria também podem ser ordenados. Para isso, basta utilizarmos o método `orderBy()`

```
1 CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2 Root<Livro> livro = c.from(Livro.class);
3 c.select(livro);
4
5 c.orderBy(cb.desc(livro.<Double>get("preco")));
```

Código Java 5.61: Ordenando o resultado de uma consulta em Criteria

Subqueries

Algumas consultas necessitam de consultas auxiliares. Por exemplo, para encontrar o livro mais caro cadastrado no banco de dados, devemos criar uma consulta auxiliar para poder comparar, dois a dois, os preços dos livros.

Uma consulta auxiliar ou uma **subquery** é criada através do método `subquery()`. No exemplo abaixo, criamos uma consulta e uma sub-consulta para encontrar o livro mais caro.

```
CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
Root<Livro> livro1 = c.from(Livro.class);
c.select(livro1);

Subquery<Livro> subquery = c.subquery(Livro.class);
Root<Livro> livro2 = subquery.from(Livro.class);
subquery.select(a2);

Predicate predicate = cb.greaterThan(livro2.<Double>get("preco"),
  livro1.<Double>get("preco"));
subquery.where(predicate);

Predicate predicate2 = cb.not(cb.exists(subquery));
c.where(predicate2);
```

Código Java 5.62: Consulta e sub-consulta para encontrar o livro mais caro

Exemplos

1. Suponha que seja necessário selecionar os livros mais baratos. Em outras palavras, devemos selecionar os livros tais que não exista nenhum outro livro com preço menor.

```
CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
Root<Livro> livro1 = c.from(Livro.class);
c.select(livro1);

Subquery<Livro> subquery = c.subquery(Livro.class);
Root<Livro> livro2 = subquery.from(Livro.class);
subquery.select(livro2);
Predicate gt = cb.gt(livro1.<Double>get("preco"), livro2.<Double>get("preco"));
subquery.where(gt);

Predicate notExists = cb.not(cb.exists(subquery));
c.where(notExists);
```

Código Java 5.63: Selecionando os livros mais baratos

2. Suponha que seja necessário selecionar os livros mais baratos de um determinado autor.

```
CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
Root<Livro> livro1 = c.from(Livro.class);
Root<Autor> autor1 = c.from(Autor.class);
c.select(livro1);

Subquery<Livro> subquery = c.subquery(Livro.class);
Root<Livro> livro2 = subquery.from(Livro.class);
Root<Autor> autor2 = subquery.from(Autor.class);
subquery.select(livro2);
Predicate isMember1 = cb.isMember(livro2, autor2.<Collection<Livro>>get("livros"));
Predicate equal1 = cb.equal(autor2, autor1);
Predicate gt = cb.gt(livro1.<Double>get("preco"), livro2.<Double>get("preco"));
Predicate predicate = cb.and(isMember1,equal1,gt);
subquery.where(predicate);
```

```
15
16 Predicate notExists = cb.not(cb.exists(subquery));
17 Predicate equal2 = cb.equal(autor1.<String>get("nome"), "Jack Trout");
18 Predicate isMember2 = cb.isMember(livro1, autor1.<Collection<Livro>>get("livros"));
19 Predicate predicate2 = cb.and(isMember2,equal2,notExists);
20 c.where(predicate2);
```

Código Java 5.64: Selecionando os livros mais barato do autor Jack Trout

3. Suponha que seja necessário listar os livros em ordem decrescente em relação aos preços.

```
1 CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2 Root<Livro> livro = c.from(Livro.class);
3 c.select(livro);
4 c.orderBy(cb.desc(livro.<Double>get("preco")));
```

Código Java 5.65: Ordenando o resultado da consulta por ordem decrescente de preço

4. Suponha que seja necessário selecionar os autores com mais livros.

```
1 CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
2 Root<Autor> autor1 = c.from(Autor.class);
3 c.select(autor1);
4
5 Subquery<Autor> subquery = c.subquery(Autor.class);
6 Root<Autor> autor2 = subquery.from(Autor.class);
7 subquery.select(autor2);
8 Predicate gt = cb.gt(cb.size(autor2.<Collection<Livro>>get("livros")), cb.size(autor1↩
      .<Collection<Livro>>get("livros")));
9 subquery.where(gt);
10
11 Predicate notExists = cb.not(cb.exists(subquery));
12 c.where(notExists);
```

Código Java 5.66: Selecionando os autores com mais livros

O Problema do $N+1$

No Capítulo 4, vimos que o uso do modo LAZY em relacionamentos pode provocar um número excessivo de consultas ao banco de dados em alguns casos. Assim como em JPQL, também podemos evitar esse problema usando fetch join em Criteria. Por exemplo, considere as entidades `Departamento` e `Funcionario`.

```
1  @Entity
2  public class Funcionario {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    private String nome;
8
9    // GETTERS E SETTERS
10 }
```

Código Java 5.67: Funcionario.java

```
1 @Entity
2 public class Departamento {
```

```
3
4   @Id @GeneratedValue
5   private Long id;
6
7   private String nome;
8
9   @OneToMany
10  private List<Funcionario> funcionarios;
11
12  // GETTERS E SETTERS
13 }
```

Código Java 5.68: Departamento.java

Se optarmos pelo modo LAZY no relacionamento entre departamentos e funcionários, então uma consulta para selecionar os departamentos não trará de imediato os seus respectivos funcionários. Para que os funcionários sejam recuperados na mesma consulta em que os departamentos são selecionados, podemos usar o método `fetch()`. Veja o exemplo abaixo.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
2 CriteriaQuery<Departamento> c = cb.createQuery(Departamento.class);
3 Root<Departamento> d = c.from(Departamento.class);
4 d.fetch("funcionarios", JoinType.LEFT);
5 c.select(d).distinct(true);
```

Código Java 5.69: Consulta para obter os departamentos e seus respectivos funcionários

Nesse exemplo, o método `distinct()` é usado para garantir que não haja elementos repetidos na lista obtida pela consulta.

Exercícios de Fixação

12 Faça uma consulta para obter todos os autores e seus respectivos livros. No pacote `br.com.k19.testes` do projeto **K19-Criteria**, crie a classe `ListaAutoresELivros`.

```
1  public class ListaAutoresELivros {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      CriteriaBuilder cb = manager.getCriteriaBuilder();
8      CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
9      Root<Autor> a = c.from(Autor.class);
10     c.select(a);
11
12     TypedQuery<Autor> query = manager.createQuery(c);
13     List<Autor> autores = query.getResultList();
14
15     System.out.println();
16
17     for (Autor autor : autores) {
18       System.out.println("Autor: " + autor.getNome());
19       Collection<Livro> livros = autor.getLivros();
20
21       for (Livro livro : livros) {
22         System.out.println("Livro: " + livro.getNome());
23       }
24       System.out.println();
```

```
25      }
26
27      manager.close();
28      factory.close();
29    }
30  }
```

Código Java 5.70: ListaAutoresELivros.java

Execute e observe, no console, a quantidade de consultas realizadas.

13 Altere a classe `ListaAutoresELivros` para reduzir a quantidade de consultas ao banco de dados. Aplique o comando `fetch join` na consulta de autores.

```
1  public class ListaAutoresELivros {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      CriteriaBuilder cb = manager.getCriteriaBuilder();
8      CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
9      Root<Autor> a = c.from(Autor.class);
10     a.fetch("livros", JoinType.LEFT);
11     c.select(a).distinct(true);
12
13     TypedQuery<Autor> query = manager.createQuery(c);
14     List<Autor> autores = query.getResultList();
15
16     System.out.println();
17
18     for (Autor autor : autores) {
19       System.out.println("Autor: " + autor.getNome());
20       Collection<Livro> livros = autor.getLivros();
21
22       for (Livro livro : livros) {
23         System.out.println("Livro: " + livro.getNome());
24       }
25       System.out.println();
26     }
27
28     manager.close();
29     factory.close();
30   }
31 }
```

Código Java 5.71: ListaAutoresELivros.java

Execute e observe, no console, que apenas uma consulta é realizada.

Operações em Lote (Bulk Operations)

No Capítulo 4, vimos que operações que atualizam ou removem uma grande quantidade de registros podem afetar consideravelmente a performance de uma aplicação. Nesses casos, as operações em lote (bulk operations) podem melhorar o desempenho evitando que dados sejam transferidos do banco de dados para a memória.

Por exemplo, considere que uma aplicação que controla os produtos de uma loja virtual necessite atualizar os preços de todos os produtos com uma taxa fixa.

```
CriteriaBuilder cb = manager.getCriteriaBuilder();

CriteriaUpdate<Produto> update = cb.createCriteriaUpdate(Produto.class);
Root<Produto> produto = update.from(Produto.class);
update.set(
  produto.<Double> get("preco"),
  cb.prod(produto.<Double> get("preco"), 1.1));

Query query = manager.createQuery(update);
query.executeUpdate();
```

Analogamente, operações em lote podem ser utilizadas para remover registros.

```
CriteriaBuilder cb = manager.getCriteriaBuilder();

CriteriaDelete<Produto> delete = cb.createCriteriaDelete(Produto.class);
Root<Produto> produto = delete.from(Produto.class);
delete.where(cb.lessThan(produto.get("preco"), 50));

Query query = manager.createQuery(delete);
query.executeUpdate();
```

Exercícios de Fixação

14 No pacote `br.com.k19.modelo` do projeto **K19-Criteria**, adicione a seguinte classe:

```
@Entity
public class Produto {

  @Id @GeneratedValue
  private Long id;

  private String nome;

  private Double preco;

  // GETTERS E SETTERS
}
```

Código Java 5.74: Produto.java

15 Adicione alguns produtos no banco de dados. Crie a seguinte classe em um pacote chamado `br.com.k19.testes` do projeto **K19-Criteria**.

```
public class AdicionaProdutos {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_criteria_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    for (int i = 0; i < 100; i++) {
      Produto p = new Produto();
      p.setNome("produto " + i);
      p.setPreco(i * 10.0);
      manager.persist(p);
    }
```

```
15
16      manager.getTransaction().commit();
17
18      manager.close();
19      factory.close();
20    }
21  }
```

Código Java 5.75: AdicionaProdutos.java

Execute e verifique a tabela produto na base de dados K21_criteria_bd.

16 Faça uma operação em lote para atualizar o preço de todos os produtos de acordo com uma taxa fixa. Crie a seguinte classe em um pacote chamado `br.com.k19.testes` do projeto **K19-Criteria**.

```
1  public class AumentaPreco {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      CriteriaBuilder cb = manager.getCriteriaBuilder();
10
11     CriteriaUpdate<Produto> update =
12       cb.createCriteriaUpdate(Produto.class);
13     Root<Produto> produto =
14       update.from(Produto.class);
15     update.set(
16       produto.<Double> get("preco"),
17       cb.prod(produto.<Double> get("preco"), 1.1));
18
19     Query query = manager.createQuery(update);
20     query.executeUpdate();
21
22     manager.getTransaction().commit();
23
24     manager.close();
25     factory.close();
26   }
27 }
```

Código Java 5.76: AumentaPreco.java

Execute e verifique a tabela produto na base de dados K21_criteria_bd.

Observe também o console do Eclipse. Nenhum select é realizado.

17 Faça uma operação em lote para remover todos os produtos com preço menor do que um valor fixo. Crie a seguinte classe em um pacote chamado `br.com.k19.testes` do projeto **K19-Criteria**.

```
1  public class RemoveProdutos {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      CriteriaBuilder cb = manager.getCriteriaBuilder();
10
```

```
11      CriteriaDelete<Produto> delete =
12        cb.createCriteriaDelete(Produto.class);
13      Root<Produto> produto =
14        delete.from(Produto.class);
15      delete.where(cb.lessThan(produto.get("preco"), 50));
16
17      Query query = manager.createQuery(delete);
18      query.executeUpdate();
19
20      manager.getTransaction().commit();
21
22      manager.close();
23      factory.close();
24    }
25  }
```

Código Java 5.77: RemoveProdutos.java

Execute e verifique a tabela produto na base de dados K21_criteria_bd.

Observe também o console do Eclipse. Nenhum select é realizado.

Metamodel

Quando definimos uma consulta em Criteria, estamos propensos a confundir os nomes ou os tipos dos atributos das entidades, já que o compilador não realiza a verificação dessas informações. Dessa forma, os erros só seriam percebidos em tempo de execução. Considere o exemplo abaixo.

```
1  public class CalculaMedia {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_entity_manager");
5      EntityManager manager = factory.createEntityManager();
6
7      CriteriaBuilder cb = manager.getCriteriaBuilder();
8      CriteriaQuery<Float> c = cb.createQuery(Float.class);
9      Root<Livro> l = c.from(Livro.class);
10     c.select(cb.max(l.<Float> get("preço")));
11
12     TypedQuery<Float> query = manager.createQuery(c);
13     Float maiorPreco = query.getSingleResult();
14
15     System.out.println("Maior preço: " + maiorPreco);
16
17     manager.close();
18     factory.close();
19   }
20 }
```

Código Java 5.78: CalculaMedia.java

Nesse código, cometemos dois erros. O primeiro deles foi supor que o atributo que armazena o preço dos livros é do tipo `Float`. Na verdade, esse atributo é do tipo `Double`. O segundo erro foi um simples erro de digitação, pois utilizamos a string `"preço"` ao invés de `"preco"`. Esses erros não são identificados pelo compilador, mas durante a execução uma exceção será lançada.

Para evitar que o desenvolvedor cometa erros desse tipo, podemos aplicar o recurso conhecido como Metamodel, definido pela especificação JPA 2.

Note que os erros cometidos no exemplo acima estão relacionados ao tipo e ao nome do atributo `preco` da entidade `Livro`. A função do Metamodel é justamente armazenar as informações referentes aos nomes e tipos dos atributos das entidades. Por exemplo, para a entidade `Livro`, teríamos uma classe semelhante à classe do código abaixo.

```
@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Livro.class)
public abstract class Livro_ {
  public static volatile SingularAttribute<Livro, Long> id;
  public static volatile SingularAttribute<Livro, Double> preco;
  public static volatile SingularAttribute<Livro, String> nome;
}
```

Código Java 5.79: Livro_.java

A classe `Livro_` foi gerada automaticamente pelo Hibernate a partir da classe `Livro`. Ela segue as regras do Canonical Metamodel definidas pela especificação JPA 2.

O próximo passo é utilizar o Canonical Metamodel da entidade `Livro` para definir a consulta anterior. Veja o código abaixo.

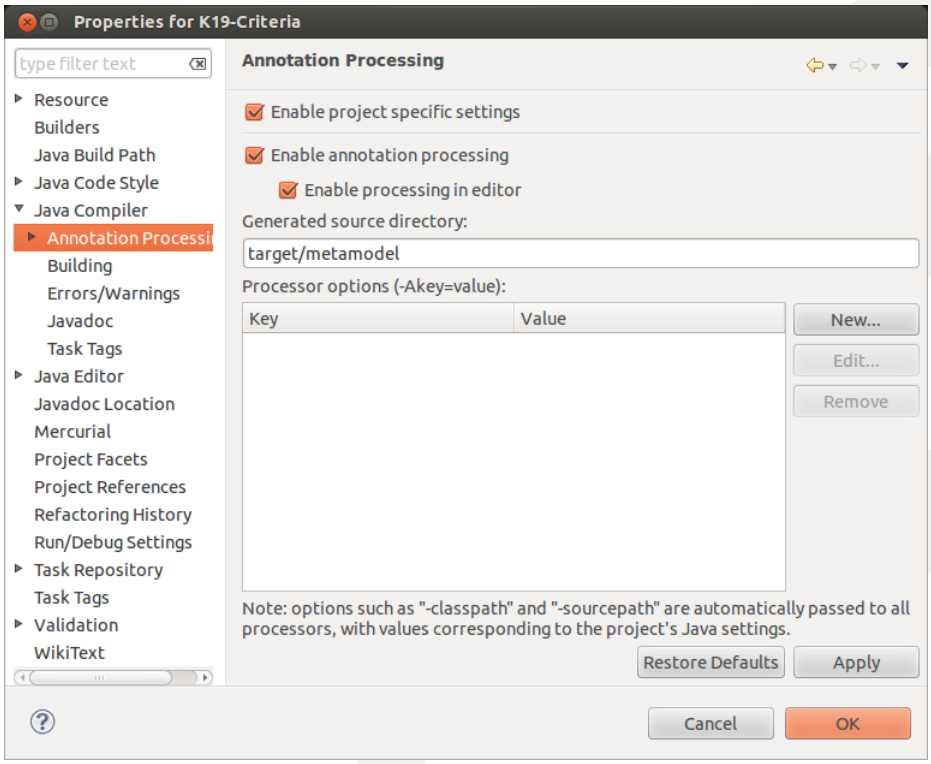
```
public class CalculaMedia {
  public static void main(String[] args) {
    EntityManagerFactory factory = Persistence
        .createEntityManagerFactory("K21_entity_manager");
    EntityManager manager = factory.createEntityManager();

    CriteriaBuilder cb = manager.getCriteriaBuilder();
    CriteriaQuery<Double> c = cb.createQuery(Double.class);
    Root<Livro> l = c.from(Livro.class);
    c.select(cb.max(l.get(Livro_.preco)));

    TypedQuery<Double> query = manager.createQuery(c);
    Double maiorPreco = query.getSingleResult();

    System.out.println("Maior preço: " + maiorPreco);

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

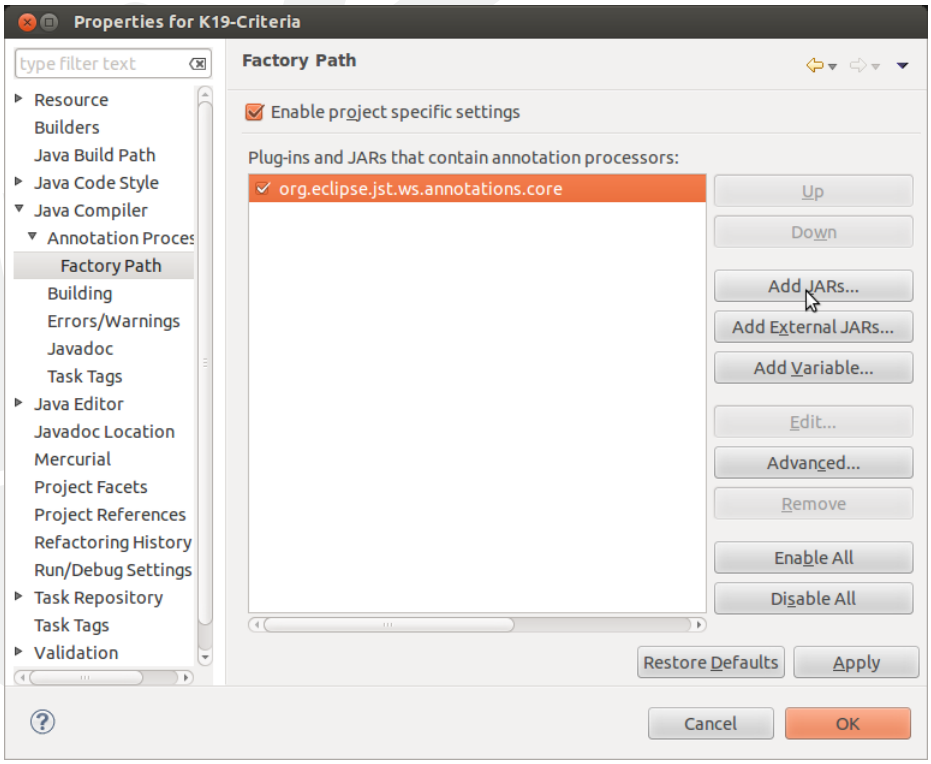
Código Java 5.80: CalculaMedia.java

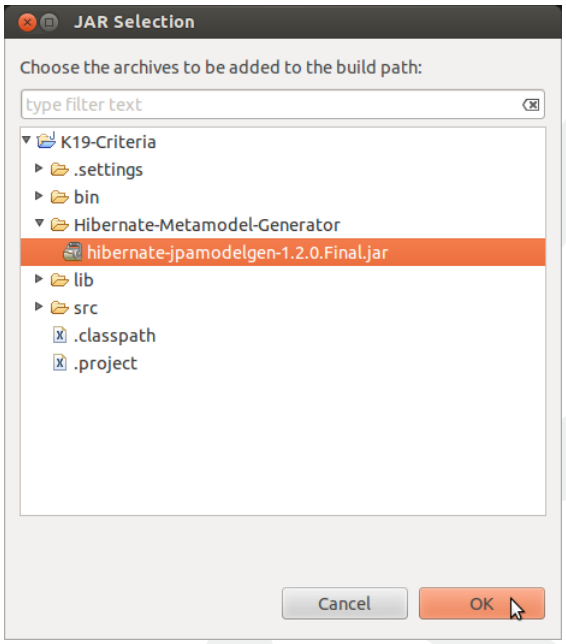
Uma vez que utilizamos o Canonical Metamodel (`Livro_`) da entidade `Livro`, não é possível parametrizar a consulta com outro tipo a não ser o `Double`, pois um erro de compilação ocorrerá. Além disso, também não podemos mais cometer o erro relacionado ao nome do atributo da entidade, sem que haja uma falha na compilação.

Exercícios de Fixação

18 Copie a pasta **Hibernate-Metamodel-Generator** localizada na Área de Trabalho dentro do diretório **K19-Arquivos** para a pasta raiz do projeto **K19-Criteria**. Em seguida, clique com o botão direito do mouse sobre o projeto **K19-Criteria**, selecione a opção "Properties" e configure o Hibernate Metamodel Generator de acordo com as figuras abaixo.


Properties for K19-Criteria
type filter text
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Annotation Processi
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
Mercurial
Project Facets
Project References
Refactoring History
Run/Debug Settings
Task Repository
Task Tags
Validation
WikiText
Annotation Processing
Enable project specific settings
Enable annotation processing
Enable processing in editor
Generated source directory:
target/metamodel
Processor options (-Akey=value):
Key
Value
New...
Edit...
Remove
Note: options such as "-classpath" and "-sourcepath" are automatically passed to all processors, with values corresponding to the project's Java settings.
Restore Defaults
Apply
Cancel
OK


Properties for K19-Criteria
type filter text
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Annotation Proces
Factory Path
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
Mercurial
Project Facets
Project References
Refactoring History
Run/Debug Settings
Task Repository
Task Tags
Validation
Factory Path
Enable project specific settings
Plug-ins and JARs that contain annotation processors:
org.eclipse.jst.ws.annotations.core
Up
Down
Add JARs...
Add External JARs...
Add Variable...
Edit...
Advanced...
Remove
Enable All
Disable All
Restore Defaults
Apply
Cancel
OK



Importante
Você também pode obter a pasta **Hibernate-Metamodel-Generator** através do site da K19: www.k19.com.br/arquivos.

19 Abra o diretório `target/metamodel` e verifique se as classes que formam o Canonical Metamodel foram geradas.

20 No pacote `br.com.k19.testes` do projeto **K19-Criteria**, crie uma classe chamada `ConsultaLivroMaisCaro`. Usando os recursos do Metamodel, faça uma consulta à base de dados `K21_criteria_bd` para saber o preço do livro mais caro.

```
1  public class ConsultaLivroMaisCaro {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      CriteriaBuilder cb = manager.getCriteriaBuilder();
8      CriteriaQuery<Double> c = cb.createQuery(Double.class);
9      Root<Livro> l = c.from(Livro.class);
10     c.select(cb.max(l.get(Livro_.preco)));
11
12     TypedQuery<Double> query = manager.createQuery(c);
13     Double maiorPreco = query.getSingleResult();
14
15     System.out.println("Maior preço: " + maiorPreco);
16
17     manager.close();
18     factory.close();
19   }
20 }
```

Código Java 5.81: ConsultaLivroMaisCaro.java

HIBERNATE SEARCH

APÊNDICE A

Eventualmente, uma aplicação Java necessita realizar constantemente buscas em uma grande quantidade de textos. Existem bibliotecas Java especializadas nesse tipo de tarefa. A mais utilizada delas é Apache Lucene (http://lucene.apache.org) da Apache Software Foundation.

A integração entre os recursos oferecidos pelo Lucene e uma aplicação orientada a objetos é complexa. O projeto Hibernate Search (http://www.hibernate.org/subprojects/search) tem como principal objetivo facilitar essa integração. O Hibernate Search é um subprojeto do Hibernate. Neste capítulo, mostraremos o funcionamento básico do Hibernate Search.

Configuração

Para utilizar os recursos do Hibernate Search em uma aplicação Java, devemos adicionar as bibliotecas do Hibernate Search e das suas dependências. Podemos obter essas bibliotecas em http://www.hibernate.org/subprojects/search/download. Neste curso, utilizaremos a versão 4.1.1 do Hibernate Search.

Após adicionar as bibliotecas necessárias, devemos acrescentar ao arquivo `persistence.xml` de uma aplicação JPA algumas propriedades para que o Hibernate Search possa ser utilizado. Veja as modificações em destaque no código abaixo.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence
3    version="2.1"
4    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
7      http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd ">
8
9    <persistence-unit name="K19-PU" transaction-type="RESOURCE_LOCAL">
10     <provider>org.hibernate.ejb.HibernatePersistence</provider>
11     <properties>
12       <property
13         name="hibernate.dialect"
14         value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15
16       <property
17         name="hibernate.hbm2ddl.auto"
18         value="update" />
19
20       <property
21         name="javax.persistence.jdbc.driver"
22         value="com.mysql.jdbc.Driver" />
23
24       <property
25         name="javax.persistence.jdbc.user"
26         value="root" />
27
28       <property
29         name="javax.persistence.jdbc.password"
```

```
30          value="root" />
31
32        <property
33          name="javax.persistence.jdbc.url"
34          value="jdbc:mysql://localhost:3306/k19_db"/>
35        <property
36          name="hibernate.search.default.directory_provider"
37          value="filesystem"/>
38        <property
39          name="hibernate.search.default.indexBase"
40          value="/k19/indexes"/>
41      </properties>
42    </persistence-unit>
43  </persistence>
```

Código XML A.1: persistence.xml

No código acima, a propriedade `hibernate.search.default.directory_provider` define o DirectoryProvider. Além disso, a propriedade `hibernate.search.default.indexBase` define o diretório os índices do Lucene serão gerados. Mais sobre a configuração do DirectoryProvider pode ser encontrada em http://www.hibernate.org/subprojects/search/docs.

Mapeamento

Podemos realizar buscas apenas nos dados das entidades indexadas. Para indicar quais entidades devem ser indexadas pelo Lucene, devemos utilizar a anotação `@Indexed` do Hibernate Search. Além disso, apenas o conteúdo dos atributos anotados com `@Field` será considerado nas buscas do Lucene.

```
1  @Entity
2  @Indexed
3  public class Pessoa {
4
5    @Id @GeneratedValue
6    private Long id;
7
8    @Field
9    private String nome;
10
11   @Lob
12   @Field
13   private String biografia;
14 }
```

Código Java A.1: Pessoa.java

Indexação

A partir do momento que as entidades estão mapeadas com as anotações do Hibernate Search, podemos indexar os dados armazenados no banco de dados referentes às entidades anotadas com `@Indexed`. Para realizar essa tarefa, devemos invocar o método `createIndexer()` da interface `FullTextEntityManager`. Veja o código abaixo.

```
1 EntityManager manager = ...
2 FullTextEntityManager fullTextManager = Search.getFullTextEntityManager(manager);
3 fullTextManager.createIndexer().startAndWait();
```

Código Java A.2: Indexando o conteúdo do banco de dados

Busca

Com o conteúdo indexado, podemos realizar buscas através do Hibernate Search. Para isso, devemos recuperar a fábrica de buscas por meio do método `getSearchFactory()` da `FullTextEntityManager`. Com a fábrica em mãos, devemos criar um montador de consultas para uma entidade em particular. Observe o código a seguir.

```
1 SearchFactory searchFactory = fullTextManager.getSearchFactory();
2 QueryBuilder pessoaQueryBuilder =
3     searchFactory.buildQueryBuilder().forEntity(Pessoa.class).get();
```

Código Java A.3: Obtendo um montador de consultas para a entidade `Pessoa`

Agora, através do `QueryBuilder`, podemos criar as consultas desejadas. Por exemplo, suponha que desejemos buscar as pessoas que possuam a palavra "programador" em sua biografia. Então, devemos indicar que a busca será realizada por palavra chave através do método `keyword()`. Além disso, devemos indicar qual atributo da entidade `Pessoa` será considerado na busca com o método `onField()`. A palavra procurada deve ser definida através do método `matching()`. Veja o código abaixo.

```
1 org.apache.lucene.search.Query luceneQuery
2     = pessoaQueryBuilder.keyword().onField("biografia").matching("programador");
```

Código Java A.4: Criando uma consulta

O próximo passo é transformar essa consulta do Lucene em uma consulta JPA. Essa tarefa é realizada com o método `createFullTextQuery()`.

```
1 javax.persistence.Query jpaQuery =
2     fullTextManager.createFullTextQuery(luceneQuery, Pessoa.class);
```

Código Java A.5: Transformando uma Lucene Query em uma JPA Query

Finalmente, podemos executar a JPA Query.

```
1 List<Pessoa> pessoas = jpaQuery.getResultList();
```

Código Java A.6: Transformando uma Lucene Query em uma JPA Query

Exercícios de Fixação

1 Crie um projeto no eclipse chamado **K19-Hibernate-Search**. Nesse projeto, adicione uma pasta chamada **lib**.

2 Entre na pasta **K19-Arquivos/hibernate-search-VERSAO.Final/dist** da Área de Trabalho e copie

os jars dessa pasta, os da pasta `lib/required`, os da pasta `lib/provided` e os da pasta `lib/optional` para a pasta **lib** do projeto **K19-Hibernate-Search**.

Importante
Você também pode esses arquivos através do site da K19: www.k19.com.br/arquivos.

3 Entre na pasta **K19-Arquivos/mysql-connector-java-VERSAO** da Área de Trabalho e copie o arquivo `mysql-connector-java-VERSAO-bin.jar` para pasta **lib** do projeto **K19-Hibernate-Search**.

Importante
Você também pode obter o arquivo **mysql-connector-java-VERSAO-bin.jar** através do site da K19: www.k19.com.br/arquivos.

4 Depois selecione todos os jars da pasta `lib` do projeto `K19-Hibernate-Search` e adicione-os no classpath.

5 Copie a pasta `META-INF` do projeto `K19-JPA2-Hibernate` para dentro da pasta `src` do projeto `K19-Hibernate-Search`. Altere o arquivo `persistence.xml` do projeto `K19-Hibernate-Search`, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence
3    version="2.1"
4    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
7      http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd ">
8
9    <persistence-unit name="K21_hibernate_search" transaction-type="RESOURCE_LOCAL">
10     <provider>org.hibernate.ejb.HibernatePersistence</provider>
11     <properties>
12       <property
13         name="hibernate.dialect"
14         value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15
16       <property
17         name="hibernate.hbm2ddl.auto"
18         value="update" />
19
20       <property
21         name="javax.persistence.jdbc.driver"
22         value="com.mysql.jdbc.Driver" />
23
24       <property
25         name="javax.persistence.jdbc.user"
26         value="root" />
27
28       <property
29         name="javax.persistence.jdbc.password"
30         value="root" />
31
32       <property
33         name="javax.persistence.jdbc.url"
```

```
34          value="jdbc:mysql://localhost:3306/K21_hibernate_search"/>
35        <property
36          name="hibernate.search.default.directory_provider"
37          value="filesystem"/>
38        <property
39          name="hibernate.search.default.indexBase"
40          value="/k19/indexes"/>
41      </properties>
42    </persistence-unit>
43  </persistence>
```

Código XML A.2: persistence.xml

6 Crie um pacote chamado **modelo** no projeto `K19-Hibernate-Search` e adicione a seguinte classe:

```
1  @Entity
2  @Indexed
3  public class Pessoa {
4
5    @Id @GeneratedValue
6    private Long id;
7
8    @Field
9    private String nome;
10
11   // GETTERS E SETTERS
12 }
```

Código Java A.7: Pessoa.java

7 Persista alguns objeto da classe `Pessoa`. Crie uma classe chamada `AdicionaPessoa` dentro de um pacote chamado `testes`.

```
1  public class AdicionaPessoa {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_hibernate_search");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Pessoa p1 = new Pessoa();
10     p1.setNome("Rafael Cosentino");
11
12     Pessoa p2 = new Pessoa();
13     p2.setNome("Jonas Hirata");
14
15     Pessoa p3 = new Pessoa();
16     p3.setNome("Marcelo Martins");
17
18     Pessoa p4 = new Pessoa();
19     p4.setNome("Christopher Hirata");
20
21     manager.persist(p1);
22     manager.persist(p2);
23     manager.persist(p3);
24     manager.persist(p4);
25
26     manager.getTransaction().commit();
27     manager.close();
28     factory.close();
29   }
```

```
30 }
```

Código Java A.8: AdicionaPessoa.java

8 Execute a indexação dos dados armazenados no banco de dados. Crie uma classe chamada `Indexacao` dentro de um pacote chamado `testes`.

```
1  public class Indexacao {
2    public static void main(String[] args) throws InterruptedException {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_hibernate_search");
5      EntityManager manager = factory.createEntityManager();
6
7      FullTextEntityManager fullTextManager = Search.getFullTextEntityManager(manager);
8      fullTextManager.createIndexer().startAndWait();
9
10     manager.close();
11     factory.close();
12   }
13 }
```

Código Java A.9: Indexacao.java

9 Busque as pessoas que possuam a palavra Hirata em seus nomes. Crie uma classe chamada `Busca` dentro de um pacote chamado `testes`.

```
1  public class Busca {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_hibernate_search");
5      EntityManager manager = factory.createEntityManager();
6
7      FullTextEntityManager fullTextManager = Search.getFullTextEntityManager(manager);
8      SearchFactory searchFactory = fullTextManager.getSearchFactory();
9      QueryBuilder pessoaQueryBuilder =
10       searchFactory.buildQueryBuilder().forEntity(Pessoa.class).get();
11
12     org.apache.lucene.search.Query luceneQuery
13       = pessoaQueryBuilder.keyword().onField("nome").matching("Hirata").createQuery()↩
           ;
14
15     javax.persistence.Query jpaQuery =
16       fullTextManager.createFullTextQuery(luceneQuery, Pessoa.class);
17
18     List<Pessoa> pessoas = jpaQuery.getResultList();
19
20     for(Pessoa p : pessoas) {
21       System.out.println(p.getNome());
22     }
23
24     manager.close();
25     factory.close();
26   }
27 }
```

Código Java A.10: Busca.java

HIBERNATE ENVERS

APÊNDICE B

Eventualmente, é necessário registrar todas as alterações realizadas nos registros correspondentes às instâncias de determinadas entidades de uma aplicação JPA. Por exemplo, considere um sistema bancário que necessita registrar todas as modificações nos saldos das contas para uma eventual auditoria.

Manter o histórico das alterações dos registros do banco de dados pode ser útil por diversos motivos. Por exemplo, recuperar dados que não deveriam ter sido modificados ou identificar uso indevido da aplicação por parte dos usuários.

O projeto Hibernate ORM possui recursos específicos para esse tipo de problema. Mais especificamente, o **Hibernate Envers** que faz parte do Hibernate ORM oferece mecanismos para controlar as modificações realizadas nos dados do banco de dados.

Neste capítulo, mostraremos como utilizar as principais funcionalidades do **Hibernate Envers**. Utilizaremos a versão 4.1.2 do Hibernate ORM que já possui o Hibernate Envers. As bibliotecas necessárias podem ser obtidas em http://www.hibernate.org/downloads.

Mapeamento

As entidades que devem ser monitoradas pelo Hibernate Envers devem ser anotadas com `@Audited`.

```
1  @Entity
2  @Audited
3  public Pessoa {
4
5    @Id
6    @GeneratedValue
7    private Long id;
8
9    private String nome;
10
11 }
```

Código Java B.1: Pessoa.java

Para cada entidade anotada com `@Audited`, uma tabela extra será criada no banco de dados para armazenar o histórico das modificações nas instâncias da entidade. Por padrão, o nome dessa tabela é a concatenação do nome da entidade com o sufixo `_AUD`.

Por exemplo, ao anotar a entidade `Pessoa` com `@Audited`, uma tabela chamada `Pessoa_AUD` é criada no banco de dados para armazenar as alterações na tabela `Pessoa`. Todo registro da tabela `Pessoa_AUD` corresponde a um único registro da tabela `Pessoa`. Os registros da tabela `Pessoa_AUD` possuem basicamente os mesmos campos dos registros da tabela `Pessoa`.

Toda vez que um registro é inserido, modificado ou removido da tabela `Pessoa` um registro com

basicamente os mesmos dados é inserido na tabela Pessoa_AUD. Os registros inseridos na tabela Pessoa_AUD possuem também um campo chamado REV e outro chamado REVTYPE. O campo REV armazena a versão do registro correspondente na tabela Pessoa. O campo REVTYPE indica se o registro da tabela Pessoa_AUD foi gerado a partir de uma inserção, modificação ou remoção na tabela Pessoa. Nas inserções o campo REVTYPE armazenará o valor 0, nas modificações armazenará o valor 1 e nas remoções o valor 2.

Consultas

O Hibernate Envers permite que os dados das instâncias de uma entidade sejam recuperados de acordo com versão do registro. Por exemplo, podemos recuperar a primeira versão de um determinado registro.

```
1 EntityManager manager = ...
2 AuditReader reader = AuditReaderFactory.get(manager);
3
4 AuditQuery query = reader.createQuery().forEntitiesAtRevision(Pessoa.class, 1);
5 query.add(AuditEntity.property("id").eq(1L));
6
7 Pessoa p = (Pessoa)query.getSingleResult();
```

Código Java B.2: Recuperando a primeira versão de um determinado registro

No código acima, recuperamos a primeira versão registrada da pessoa com identificador 1.

Exercícios de Fixação

1 Crie um projeto no eclipse chamado **K19-Hibernate-Envers**. Copie a pasta **lib** do projeto **K19-JPA2-Hibernate** para o projeto **K19-Hibernate-Envers**.

2 Entre na pasta **K19-Arquivos/hibernate-release-VERSAO.Final/lib** da Área de Trabalho e copie o jar da pasta envers para a pasta **lib** do projeto **K19-Hibernate-Envers**.

Importante
Você também pode obter esse arquivo através do site da K19: www.k19.com.br/arquivos.

3 Depois selecione todos os jars da pasta lib do projeto K19-Hibernate-Envers e adicione-os no classpath.

4 Copie a pasta META-INF do projeto K19-JPA2-Hibernate para dentro da pasta src do projeto K19-Hibernate-Envers. Altere o arquivo persistence.xml do projeto K19-Hibernate-Envers, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```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="K21_hibernate_envers" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property
        name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL5InnoDBDialect" />

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

      <property
        name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver" />

      <property
        name="javax.persistence.jdbc.user"
        value="root" />

      <property
        name="javax.persistence.jdbc.password"
        value="root" />

      <property
        name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/K21_hibernate_envers"/>
    </properties>
  </persistence-unit>
</persistence>
```

Código XML B.1: persistence.xml

5 Crie um pacote chamado **modelo** no projeto `K19-Hibernate-Envers` e adicione a seguinte classe:

```java
@Entity
@Audited
public class Pessoa {

  @Id @GeneratedValue
  private Long id;

  private String nome;

  // GETTERS E SETTERS
}
```

Código Java B.3: Pessoa.java

6 Persista um objeto da classe `Pessoa`. Crie uma classe chamada `AdicionaPessoa` dentro de um pacote chamado `testes`.

```java
public class AdicionaPessoa {
  public static void main(String[] args) {
    EntityManagerFactory factory =
```

```
4         Persistence.createEntityManagerFactory("K21_hibernate_envers");
5     EntityManager manager = factory.createEntityManager();
6
7     manager.getTransaction().begin();
8
9     Pessoa p = new Pessoa();
10    p.setNome("Rafael Cosentino");
11
12    manager.persist(p);
13
14    manager.getTransaction().commit();
15    manager.close();
16    factory.close();
17  }
18 }
```

Código Java B.4: AdicionaPessoa.java

Execute e consulte o banco de dados através do MySQL Workbench!

7 Altere o objeto da classe `Pessoa` persistido anteriormente. Crie uma classe chamada `AlteraPessoa` dentro de um pacote chamado `testes`.

```
1  public class AlteraPessoa {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_hibernate_envers");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Pessoa p = manager.find(Pessoa.class, 1L);
10
11     p.setNome("Marcelo Martins");
12
13     manager.getTransaction().commit();
14     manager.close();
15     factory.close();
16   }
17 }
```

Código Java B.5: AlteraPessoa.java

Execute e consulte o banco de dados através do MySQL Workbench!

8 Remova o objeto da classe `Pessoa` persistido anteriormente. Crie uma classe chamada `RemovePessoa` dentro de um pacote chamado `testes`.

```
1  public class RemovePessoa {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_hibernate_envers");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Pessoa p = manager.find(Pessoa.class, 1L);
10
11     manager.remove(p);
12
13     manager.getTransaction().commit();
14     manager.close();
```

```
15      factory.close();
16    }
17  }
```

Código Java B.6: RemovePessoa.java

Execute e consulte o banco de dados através do MySQL Workbench!

9 Consulte a primeira versão do objeto da classe `Pessoa` persistido anteriormente. Crie uma classe chamada `ConsultaPessoa` dentro de um pacote chamado `testes`.

```
1  public class ConsultaPessoa {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_hibernate_envers");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      AuditReader reader = AuditReaderFactory.get(manager);
10
11     AuditQuery query = reader.createQuery().forEntitiesAtRevision(Pessoa.class, 1);
12     query.add(AuditEntity.id().eq(1L));
13
14     Pessoa p = (Pessoa)query.getSingleResult();
15
16     System.out.println(p.getNome());
17
18     manager.getTransaction().commit();
19     manager.close();
20     factory.close();
21   }
22 }
```

Código Java B.7: ConsultaPessoa.java

Bean Validation e Hibernate Validator

APÊNDICE C

Uma tarefa extremamente comum no desenvolvimento de aplicações JPA é a validação dos dados das instâncias da entidades. Por exemplo, considere a entidade a seguir.

```
1  @Entity
2  public class Pessoa {
3
4    @Id
5    @GeneratedValue
6    private Long id;
7
8    private String nome;
9
10 }
```

Código Java C.1: Pessoa.java

Antes de persistir ou atualizar os dados de um objeto de uma classe `Pessoa` no banco de dados, pode ser necessário verificar se o atributo `nome` desse objeto não é nulo.

Na plataforma Java, uma especificação foi definida para padronizar a tarefa de validação dos objetos do domínio de uma aplicação. O nome dessa especificação é **Bean Validation** e ela pode ser obtida em `http://jcp.org/en/jsr/detail?id=303`. O projeto **Hibernate Validator** é a implementação de referência da Bean Validation e pode ser obtido em `http://www.hibernate.org/subprojects/validator/download`. utilizaremos a versão 4.3.0 do projeto Hibernate Validator.

Neste capítulo, mostraremos como utilizar os recursos básicos definidos pela especificação Bean Validation e implementados pelo projeto Hibernate Validator.

Regras de Validação

As regras de validação podem ser definidas através de anotações inseridas nos atributos das entidades. A especificação Bean Validation define um conjunto padrão de anotações para definir as regras de validação.

Veja abaixo as anotações disponíveis e suas respectivas funções.

- `@AssertFalse`
  Verifica se uma propriedade booleana possui valor `false`.
- `@AssertTrue`
  Verifica se uma propriedade booleana possui valor `true`.
- `@DecimalMax`
  Define o valor real máximo que uma propriedade pode armazenar.

- `@DecimalMin`
  Define o valor real mínimo que uma propriedade pode assumir.
- `@Digits`
  Define a quantidade máxima de dígitos da parte inteira (através do atributo `integer`) ou da parte fracionária (através do atributo `fraction`) de um número.
- `@Future`
  Verifica se uma data é posterior ao instante atual.
- `@Max`
  Define o valor inteiro máximo que uma propriedade pode assumir.
- `@Min`
  Define o valor inteiro mínimo que uma propriedade pode armazenar.
- `@NotNull`
  Verifica se o valor de uma propriedade não é `null`.
- `@Null`
  Verifica se o valor de uma propriedade é `null`.
- `@Past`
  Verifica se uma data é anterior ao instante atual.
- `@Pattern`
  Verifica se o valor de uma propriedade respeita uma expressão regular.
- `@Size`
  Define os tamanhos mínimo (através do atributo `min`) e máximo (através do atributo `max`) para uma `Collection`, array ou `String`.

Processando as Validações

As validações podem ser processadas manualmente pela aplicação através do método `validate()`. Esse método recebe o objeto que desejamos validar e devolve um conjunto com os erros encontrados ou um conjunto vazio caso nenhum erro seja encontrado. Veja o código abaixo.

```
1 Pessoa p = ...
2 ValidatorFactory validatorFactory = Validation.buildDefaultValidatorFactory();
3 Validator validator = validatorFactory.getValidator();
4
5 Set<ConstraintViolation<Topico>> errors = validator.validate(p);
6
7 for (ConstraintViolation<Topico> error : errors) {
8   System.out.println(error);
9 }
```

Código Java C.2: Validando um objeto

As validações podem ser realizadas automaticamente em determinados eventos. Esse comportamento pode ser configurado no arquivo `persistence.xml` através das propriedades:

- javax.persistence.validation.group.pre-persist
- javax.persistence.validation.group.pre-update

- javax.persistence.validation.group.pre-remove

Considere que a propriedade `javax.persistence.validation.group.pre-persist` esteja configurada no `persistence.xml`. Nesse caso, os objetos são validados no evento `PrePersist`, ou seja, quando o método `persist()` for chamado.

Exercícios de Fixação

1 Crie um projeto no eclipse chamado **K19-Bean-Validation**. Copie a pasta **lib** do projeto **K19-JPA2-Hibernate** para o projeto **K19-Bean-Validation**.

2 Entre na pasta **K19-Arquivos/hibernate-validator-VERSAO.Final/dist** da Área de Trabalho e copie os jars dessa pasta, os da pasta `lib/required` e os da pasta `lib/optional` para a pasta **lib** do projeto **K19-Bean-Validation**.

Importante
Você também pode obter esses arquivos através do site da K19: www.k19.com.br/arquivos.

3 Depois selecione todos os jars da pasta `lib` do projeto `K19-Bean-Validation` e adicione-os no classpath.

4 Copie a pasta `META-INF` do projeto `K19-JPA2-Hibernate` para dentro da pasta `src` do projeto `K19-Bean-Validation`. Altere o arquivo `persistence.xml` do projeto `K19-Bean-Validation`, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence
3    version="2.1"
4    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
7      http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd ">
8
9    <persistence-unit name="K21_bean_validation" transaction-type="RESOURCE_LOCAL">
10     <provider>org.hibernate.ejb.HibernatePersistence</provider>
11     <properties>
12       <property
13         name="hibernate.dialect"
14         value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15
16       <property
17         name="hibernate.hbm2ddl.auto"
18         value="update" />
19
20       <property
21         name="javax.persistence.jdbc.driver"
22         value="com.mysql.jdbc.Driver" />
23
24       <property
```

```
25          name="javax.persistence.jdbc.user"
26          value="root" />
27
28        <property
29          name="javax.persistence.jdbc.password"
30          value="root" />
31
32        <property
33          name="javax.persistence.jdbc.url"
34          value="jdbc:mysql://localhost:3306/K21_bean_validation"/>
35      </properties>
36    </persistence-unit>
37  </persistence>
```

Código XML C.1: persistence.xml

5 Crie um pacote chamado **modelo** no projeto `K19-Bean-Validation` e adicione a seguinte classe:

```
1  @Entity
2  public class Pessoa {
3
4    @Id @GeneratedValue
5    private Long id;
6
7    @NotNull
8    private String nome;
9
10   // GETTERS E SETTERS
11 }
```

Código Java C.3: Pessoa.java

6 Valide um objeto da classe `Pessoa`. Crie uma classe chamada `ValidaPessoa` dentro de um pacote chamado `testes`.

```
1  public class ValidaPessoa {
2    public static void main(String[] args) {
3      Pessoa p = new Pessoa();
4
5      ValidatorFactory validatorFactory = Validation.buildDefaultValidatorFactory();
6      Validator validator = validatorFactory.getValidator();
7
8      Set<ConstraintViolation<Pessoa>> errors = validator.validate(p);
9
10     for (ConstraintViolation<Pessoa> error : errors) {
11       System.out.println(error);
12     }
13   }
14 }
```

Código Java C.4: ValidaPessoa.java

Execute e observe as mensagens no Console

7 Configure o Hibernate Validation para que as validações sejam processadas automaticamente no evento `PrePersist`. Para isso, altere o arquivo `persistence.xml` do projeto `K19-Bean-Validation`. Veja como o código deve ficar:

```
1  <persistence version="2.0"
```

```
2   xmlns="http://java.sun.com/xml/ns/persistence"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/↩
        ns/persistence/persistence_2_0.xsd">
5
6   <persistence-unit name="K21_bean_validation" transaction-type="RESOURCE_LOCAL">
7     <provider>org.hibernate.ejb.HibernatePersistence</provider>
8     <properties>
9       <property name="hibernate.dialect"
10        value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
11      <property name="hibernate.hbm2ddl.auto" value="update"/>
12      <property name="hibernate.show_sql" value="true"/>
13      <property name="javax.persistence.jdbc.driver"
14        value="com.mysql.jdbc.Driver"/>
15      <property name="javax.persistence.jdbc.user" value="root"/>
16      <property name="javax.persistence.jdbc.password" value="root"/>
17      <property name="javax.persistence.jdbc.url"
18        value="jdbc:mysql://localhost:3306/K21_bean_validation"/>
19
20      <property name="javax.persistence.validation.group.pre-persist"
21        value="javax.validation.groups.Default"/>
22    </properties>
23  </persistence-unit>
24 </persistence>
```

Código XML C.2: persistence.xml

8 Persista um objeto da classe `Pessoa`. Crie uma classe chamada `AdicionaPessoa` dentro de um pacote chamado `testes`.

```
1  public class AdicionaPessoa {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_bean_validation");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Pessoa p = new Pessoa();
10
11     manager.persist(p);
12
13     manager.getTransaction().commit();
14     manager.close();
15     factory.close();
16   }
17 }
```

Código Java C.5: AdicionaPessoa.java

Execute e observe as mensagens no Console

MAPEAMENTO COM XML

APÊNDICE D

No Capítulo 2, vimos como o mapeamento objeto-relacional pode ser feito com o uso de anotações. Esse mesmo mapeamento também pode ser definido por meio de um arquivo XML. Neste capítulo, veremos como definir o mapeamento no arquivo `orm.xml`, que deve ficar no diretório `META-INF` no classpath da aplicação. O arquivo `orm.xml` deve conter a seguinte estrutura.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <entity-mappings
4    xmlns="http://java.sun.com/xml/ns/persistence/orm"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
7                        http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
8    version="2.0">
9
10 </entity-mappings>
```

Código XML D.1: orm.xml

Todas as regras de mapeamento devem ser especificadas dentro da tag `<entity-mappings>`. Para simplificar, omitiremos parte do cabeçalho do arquivo `orm.xml` nos exemplos apresentados neste capítulo.

Entidades

As classes da nossa aplicação que serão mapeadas para tabelas do banco de dados devem ser especificadas no arquivo `orm.xml` por meio da tag `<entity>`. Cada instância de uma entidade deve possuir um identificador único. Em geral, esse identificador é um atributo numérico. No exemplo abaixo, a classe `Pessoa` possui um atributo chamado `id`, que será esse identificador. Esse atributo é definido como identificador através da tag `<id>`. Veja o exemplo abaixo.

```
1  package br.com.k19;
2
3  class Pessoa {
4    private Long id;
5
6    // GETTERS E SETTERS
7  }
```

Código Java D.1: Pessoa.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pessoa">
3      <attributes>
4        <id name="id" />
5      </attributes>
6    </entity>
7  </entity-mappings>
```

Código XML D.2: orm.xml

Dessa forma, a coluna correspondente ao atributo `id` será definida como chave primária da tabela correspondente à classe `Pessoa`.

Importante

Por padrão, quando o mapeamento é definido pelo arquivo `orm.xml`, o modo de acesso aos dados das entidades é Property Access. Ou seja, o provedor JPA acessa os dados dos objetos por meio dos métodos getters e setters dos objetos. Dessa forma, os métodos getters e setters devem estar implementados. Para alterar esse comportamento, podemos usar o atributo `access` da tag `entity`, como mostrado no exemplo abaixo.

```
1 <entity-mappings>
2   <entity class="br.com.k19.Pessoa" access="FIELD">
3     <attributes>
4       <id name="id" />
5     </attributes>
6   </entity>
7 </entity-mappings>
```

Código XML D.3: orm.xml

Para definir o modo de acesso como sendo Field Access para todas as classes da unidade de persistência, podemos usar a tag `<access>`.

```
1 <entity-mappings>
2   <persistence-unit-metadata>
3     <persistence-unit-defaults>
4       <access>FIELD</access>
5     </persistence-unit-defaults>
6   </persistence-unit-metadata>
7   ...
8 </entity-mappings>
```

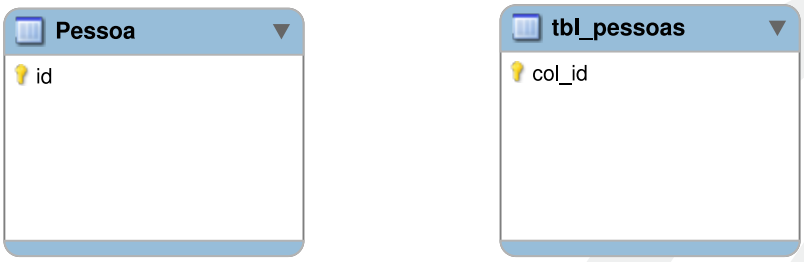
Código XML D.4: orm.xml

Mais detalhes sobre os modos Property Access e Field Access podem ser encontrados adiante.

Por convenção, a classe `Pessoa` será mapeada para uma tabela com o mesmo nome (`Pessoa`). O atributo `id` será mapeado para uma coluna com o mesmo nome (`id`) na tabela `Pessoa`. As tags `<table>` e `column` podem ser usadas para personalizar os nomes das tabelas e das colunas, respectivamente.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pessoa">
3      <table name="tbl_pessoas"/>
4      <attributes>
5        <id name="id">
6          <column name="col_id"/>
7        </id>
8      </attributes>
9    </entity>
10 </entity-mappings>
```

Código XML D.5: orm.xml



(a) (b)

Figura D.1: Tabelas correspondentes à classe `Pessoa`*. Em (a), os nomes da tabela e da coluna são padrões. Em (b), esses nomes são personalizados.*

Definindo Restrições

Podemos definir algumas restrições para os atributos das nossas entidades através das propriedades da tag `<column>`. Veja as principais propriedades abaixo:

`length`	Limita a quantidade de caracteres de uma string
`nullable`	Determina se o campo pode possuir valores `null` ou não
`unique`	Determina se uma coluna pode ter valores repetidos ou não
`precision`	Determina a quantidade de dígitos de um número decimal a serem armazenadas
`scale`	Determina a quantidade de casas decimais de um número decimal

Tabela D.1: Algumas propriedades da anotação `@Column`

No exemplo a seguir, associamos três restrições ao atributo `nome` da classe `Pessoa`. O nome deve possuir no máximo 30 caracteres, não pode ser nulo e duas pessoas não podem ter o mesmo nome. Além disso, definimos que a altura das pessoas será representada por um número de três dígitos, sendo dois deles referentes às casas decimais.

```
1  package br.com.k19;
2
3  class Pessoa {
4    private Long id;
5    private String nome;
6    private BigDecimal altura;
7  }
```

Código Java D.2: Pessoa.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pessoa">
3      <attributes>
4        <id name="id" />
5        <basic name="nome">
6          <column length="30" nullable="false" unique="true"/>
7        </basic>
8        <basic name="altura">
9          <column precision="3" scale="2"/>
10       </basic>
11     </attributes>
12   </entity>
13 </entity-mappings>
```

Código XML D.6: orm.xml

Gerando chaves primárias automaticamente

Em geral, os bancos de dados oferecem algum mecanismo para gerar os valores de uma chave primária simples e numérica. Do ponto de vista do desenvolvedor JPA, basta usar a tag **<generated-value/>** para que o banco gere os valores de uma chave primária simples e numérica automaticamente.

```
1 <entity-mappings>
2   <entity class="br.com.k19.Pessoa">
3     <attributes>
4       <id name="id">
5         <generated-value/>
6       </id>
7     </attributes>
8   </entity>
9 </entity-mappings>
```

Código XML D.7: orm.xml

Mapeamento Automático

Cada banco possui o seu próprio conjunto de tipos de dados. Para que as informações possam navegar da aplicação para o banco e vice-e-versa, os tipos do Java devem ser mapeados para tipos apropriados do banco de dados.

Alguns tipos do Java são mapeados automaticamente para tipos correspondentes do banco de dados. Eis uma lista dos tipos que são mapeados automaticamente:

- Tipos primitivos (byte, short, char, int, long, float, double e boolean)
- Classes Wrappers (Byte, Short, Character, Integer, Long, Float, Double e Boolean)
- String
- BigInteger e BigDecimal
- java.util.Date e java.util.Calendar
- java.sql.Date, java.sql.Time e java.sql.Timestamp
- Array de byte ou char
- Enums
- Serializables

Esses tipos são chamados de tipos básicos.

Objetos Grandes (LOB)

Eventualmente, dados maiores do que o comum devem ser armazenados no banco de dados. Por exemplo, uma imagem, uma música ou um texto com muitas palavras. Para esses casos, os bancos de dados oferecem tipos de dados específicos. Do ponto de vista do desenvolvedor JPA, basta usar a tag **<lob>** (*Large Objects*) para atributos do tipo `String`, `byte[]`, `Byte[]`, `char[]` ou `Character[]` para que o provedor (Hibernate, EclipseLink ou outra implementação de JPA) utilize os procedimentos adequados para manipular esses dados.

```
1 package br.com.k19;
2
3 class Pessoa {
4   private Long id;
5   private byte[] avatar;
6 }
```

Código Java D.3: Pessoa.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pessoa">
3      <attributes>
4        <id name="id">
5          <generated-value/>
6        </id>
7        <basic name="avatar">
8          <lob/>
9        </basic>
10     </attributes>
11   </entity>
12 </entity-mappings>
```

Código XML D.8: orm.xml

Data e Hora

Comumente, as aplicações Java utilizam as classes `java.util.Date` e `java.util.Calendar` para trabalhar com datas e horas. Essas classes são mapeadas automaticamente para tipos adequados no banco de dados. Portanto, basta declarar os atributos utilizando um desses dois tipos nas classes que serão mapeadas para tabelas.

```
1 package br.com.k19;
2
3 class Pessoa {
4   private Long id;
5   private Calendar nascimento;
6 }
```

Código Java D.4: Pessoa.java

Por padrão, quando aplicamos o tipo `java.util.Date` ou `java.util.Calendar`, tanto a data quanto a hora serão armazenadas no banco de dados. Para mudar esse comportamento, devemos usar a tag **<temporal>** escolhendo uma das três opções abaixo:

DATE: Armazena apenas a data (dia, mês e ano).

TIME: Armazena apenas o horário (hora, minuto e segundo).

TIMESTAMP (Padrão): Armazena a data e o horário.

```
<entity-mappings>
  <entity class="br.com.k19.Pessoa">
    <attributes>
      <id name="id">
        <generated-value/>
      </id>
      <basic name="nascimento">
        <temporal>DATE</temporal>
      </basic>
    </attributes>
  </entity>
</entity-mappings>
```

Código XML D.9: orm.xml

Dados Transientes

Eventualmente, não desejamos que alguns atributos de um determinado grupo de objetos sejam persistidos no banco de dados. Nesse caso, devemos usar a tag **<transient>**.

No exemplo abaixo, usamos a tag `<transient>` para indicar que o atributo `idade` não deve ser armazenado no banco de dados. A idade de uma pessoa pode ser deduzida a partir de sua data de nascimento, que já está armazenada no banco.

```
package br.com.k19;

class Pessoa {
  private Long id;
  private Calendar nascimento;
  private int idade;
}
```

Código Java D.5: Pessoa.java

```
<entity-mappings>
  <entity class="br.com.k19.Pessoa">
    <attributes>
      <id name="id">
        <generated-value/>
      </id>
      <basic name="nascimento">
        <temporal>DATE</temporal>
      </basic>
      <transient name="idade"/>
    </attributes>
  </entity>
</entity-mappings>
```

Código XML D.10: orm.xml

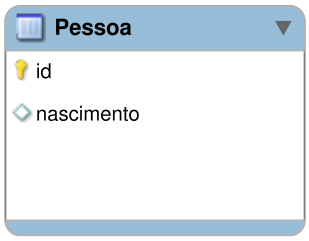


Figura D.2: Tabela correspondente à classe `Pessoa`. *Note que essa tabela não possui nenhuma coluna associada ao atributo* `idade` *da classe* `Pessoa`

Field Access e Property Access

Os provedores de JPA precisam ter acesso ao estado das entidades para poder administrá-las. Por exemplo, quando persistimos uma instância de uma entidade, o provedor deve "pegar" os dados desse objeto e armazená-los no banco. Quando buscamos uma instância de uma entidade, o provedor recupera as informações correspondentes do banco de dados e "guarda" em um objeto.

O JPA 2 define dois modos de acesso ao estado das instâncias das entidades: **Field Access** e **Property Access**. Quando colocamos as anotações de mapeamento nos atributos, estamos optando pelo modo Field Access. Por outro lado, também podemos colocar essas mesmas anotações nos métodos getters. Nesse caso, estamos optando pelo modo Property Accesss.

No modo Field Access, os atributos dos objetos são acessados diretamente através de **reflection** e não é necessário implementar métodos getters e setters. Nesse modo de acesso, se os métodos getters e setters estiverem implementados, eles não serão utilizados pelo provedor JPA.

No modo Property Access, os métodos getters e setters devem necessariamente ser implementados pelo desenvolvedor. Esses métodos serão utilizados pelo provedor para que ele possa acessar e modificar o estado dos objetos.

Exercícios de Fixação

1 Crie um projeto no Eclipse chamado **K19-Mapeamento-XML**. Copie a pasta **lib** do projeto **K19-JPA2-Hibernate** para o projeto **K19-Mapeamento-XML**. Depois adicione os jars dessa pasta no classpath desse novo projeto.

2 Abra o **MySQL Workbench** e apague a base de dados **K21_mapeamento_xml_bd** se existir. Depois crie a base de dados **K21_mapeamento_xml_bd**.

3 Copie a pasta **META-INF** do projeto **K19-JPA2-Hibernate** para dentro da pasta **src** do projeto **K19-Mapeamento-XML**. Altere o arquivo **persistence.xml** do projeto **K19-Mapeamento-XML**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

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

```
3   version="2.1"
4   xmlns="http://xmlns.jcp.org/xml/ns/persistence"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
7     http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd ">
8
9   <persistence-unit name="K21_mapeamento_xml_pu" transaction-type="RESOURCE_LOCAL">
10    <provider>org.hibernate.ejb.HibernatePersistence</provider>
11    <properties>
12      <property
13        name="hibernate.dialect"
14        value="org.hibernate.dialect.MySQL5InnoDBDialect" />
15
16      <property
17        name="hibernate.hbm2ddl.auto"
18        value="update" />
19
20      <property
21        name="javax.persistence.jdbc.driver"
22        value="com.mysql.jdbc.Driver" />
23
24      <property
25        name="javax.persistence.jdbc.user"
26        value="root" />
27
28      <property
29        name="javax.persistence.jdbc.password"
30        value="root" />
31
32      <property
33        name="javax.persistence.jdbc.url"
34        value="jdbc:mysql://localhost:3306/K21_mapeamento_xml_bd"/>
35    </properties>
36  </persistence-unit>
37 </persistence>
```

Código XML D.11: persistence.xml

4 Crie uma entidade para modelar os usuários de uma rede social dentro de um pacote chamado `br.com.k19.modelo` no projeto **K19-Mapeamento-XML**.

```
1  public class Usuario {
2    private Long id;
3
4    private String email;
5
6    private Calendar dataDeCadastro;
7
8    private byte[] foto;
9
10   // GETTERS AND SETTERS
11 }
```

Código Java D.6: Usuario.java

5 Adicione o arquivo de mapeamento `orm.xml` na pasta **src/META-INF** do projeto **K19-Mapeamento-XML**.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <entity-mappings
4    xmlns="http://java.sun.com/xml/ns/persistence/orm"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
6   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
7                       http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
8   version="2.0">
9
10  <entity class="br.com.k19.modelo.Usuario" access="FIELD">
11    <attributes>
12      <id name="id">
13        <generated-value />
14      </id>
15
16      <basic name="email">
17        <column unique="true" />
18      </basic>
19
20      <basic name="dataDeCadastro">
21        <temporal>DATE</temporal>
22      </basic>
23
24      <basic name="foto">
25        <lob />
26      </basic>
27    </attributes>
28  </entity>
29
30 </entity-mappings>
```

Código XML D.12: orm.xml

6 Adicione um usuário no banco de dados. Crie uma classe chamada `AdicionaUsuario` em um pacote chamado `br.com.k19.testes` do projeto **K19-Mapeamento-XML**.

```
1  public class AdicionaUsuario {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4          Persistence.createEntityManagerFactory("K21_mapeamento_xml_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Usuario usuario = new Usuario();
10     usuario.setEmail("contato@k19.com.br");
11     usuario.setDataDeCadastro(Calendar.getInstance());
12
13     manager.persist(usuario);
14
15     manager.getTransaction().commit();
16
17     manager.close();
18     factory.close();
19   }
20 }
```

Código Java D.7: AdicionaUsuario.java

7 Abra o MySQL Workbench e observe as propriedades da tabela **Usuario** da base de dados **K21_mapeamento_xml_bd**.

Enums

Por padrão, os tipos enumerados de JAVA são mapeados para colunas numéricas inteiras no banco de dados. Cada elemento de um Enum é associado a um número inteiro. Essa associação é baseada na ordem em que os elementos do Enum são declarados. O primeiro elemento será associado ao valor 0, o segundo será associado ao valor 1 e assim por diante. Considere o exemplo a seguir.

```
1 package br.com.k19;
2
3 public enum Periodo {
4   MATUTINO,
5   NOTURNO
6 }
```

Código Java D.8: Periodo.java

```
1 package br.com.k19;
2
3 public class Turma {
4   private Long id;
5
6   private Periodo periodo;
7 }
```

Código Java D.9: Turma.java

O Enum `Periodo` possui dois elementos: `MATUTINO` e `NOTURNO`. O elemento `MATUTINO` será associado ao valor 0 e o elemento `NOTURNO` será associado ao valor 1.

A tabela correspondente à classe `Turma` possuirá um campo chamado `periodo`. Nos registros correspondentes às turmas de período matutino, esse campo possuirá o valor 0. Já nos registros correspondentes às turmas de período noturno, esse campo possuirá o valor 1.

Imagine que um novo período é adicionado, digamos, o período vespertino. Nesse caso, o Enum `Periodo` poderia vir a ser:

```
1 package br.com.k19;
2
3 public enum Periodo {
4   MATUTINO,
5   VESPERTINO,
6   NOTURNO
7 }
```

Código Java D.10: Periodo.java

Os valores já armazenados no banco de dados poderiam estar incorretos. Por exemplo, antes dessa modificação, o campo `periodo` das turmas noturnas deveria armazenar o valor 1. Após essa modificação, o valor correto passa a ser 2. Assim, os valores do campo `periodo` da tabela `Turma` devem ser atualizados de acordo. No entanto, essa atualização não é automática, e deve ser feita manualmente.

Para evitar esse problema, podemos fazer com que os elementos de um Enum sejam associados a uma string ao invés de um número inteiro. Isso pode ser feito com o uso da tag `<enumerated>`. Observe o exemplo abaixo.

```
1 <entity-mappings>
2   <entity class="br.com.k19.Turma">
3     <attributes>
```

```
4       <id name="id">
5         <generated-value />
6       </id>
7       <basic name="periodo">
8         <enumerated>STRING</enumerated>
9       </basic>
10    </attributes>
11  </entity>
12 </entity-mappings>
```

Código XML D.13: orm.xml

Nesse exemplo, os elementos `MATUTINO`, `VESPERTINO` e `NOTURNO` do Enum `Periodo` serão associados às strings `"MATUTINO"`, `"VESPERTINO"` e `"NOTURNO"`, respectivamente.

Coleções

Considere um sistema que controla o cadastro dos funcionários de uma empresa. Esses funcionários são modelados pela seguinte classe.

```
1 package br.com.k19;
2
3 public class Funcionario {
4   private Long id;
5   private String nome;
6 }
```

Código Java D.11: Funcionario.java

Devemos também registrar os telefones de contato dos funcionários, sendo que cada funcionário pode ter um ou mais telefones. Em Java, seria razoável utilizar coleções para armazenar os telefones dos funcionários. Veja o exemplo abaixo.

```
1 package br.com.k19;
2
3 public class Funcionario {
4   private Long id;
5   private String nome;
6   private Collection<String> telefones;
7 }
```

Código Java D.12: Funcionario.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Funcionario">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7        <element-collection name="telefones" />
8      </attributes>
9    </entity>
10 </entity-mappings>
```

Código XML D.14: orm.xml

A tag `<element-collection>` deve ser utilizada para que o mapeamento seja realizado. Nesse exemplo, o banco de dados possuiria uma tabela chamada `Funcionario_telefones` contendo duas

colunas. Uma coluna seria usada para armazenar os identificadores dos funcionários e a outra para os telefones. Veja uma ilustração das tabelas do banco de dados na figura abaixo.

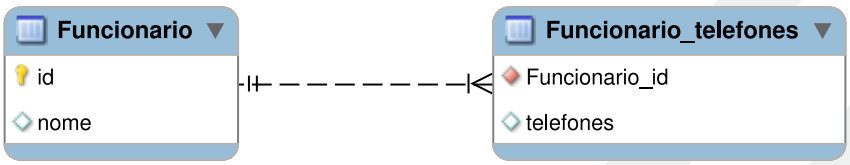


Figura D.3: Tabelas correspondentes à classe `Funcionario` *e ao atributo* `telefones`

A tabela criada para guardar os telefones dos funcionários também pode ter o seu nome personalizado, assim como os nomes de suas colunas. Para isso, devemos usar as tags `<collection-table>`, `<join-column>` e `<column>`.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Funcionario">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7        <element-collection name="telefones">
8          <column name="telefone" />
9          <collection-table name="Telefones_dos_Funcionarios">
10           <join-column name="func_id" />
11         </collection-table>
12       </element-collection>
13     </attributes>
14   </entity>
15 </entity-mappings>
```

Código XML D.15: orm.xml

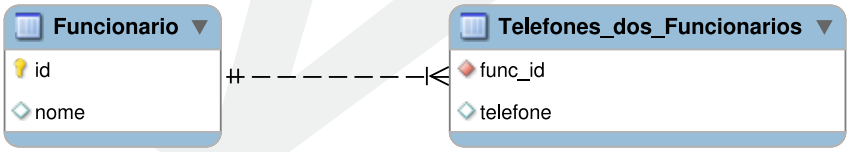
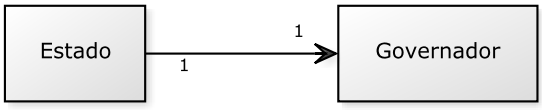


Figura D.4: Personalizando os nomes da tabela e das colunas

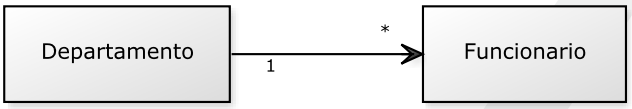
Relacionamentos

Os relacionamentos entre as entidades de um domínio devem ser expressos na modelagem através de vínculos entre classes. De acordo com a JPA, podemos definir quatro tipos de relacionamentos de acordo com a cardinalidade.

One to One (Um para Um): Por exemplo, um estado é governado por apenas um governador e um governador governa apenas um estado.

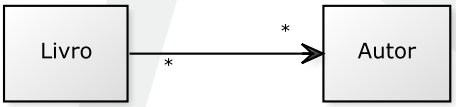


One to Many (Um para Muitos): Por exemplo, um departamento possui muitos funcionários e um funcionário trabalha em apenas em um departamento.



Many to One (Muitos para Um): Por exemplo, um pedido pertence a apenas um cliente e um cliente faz muitos pedidos.

Many to Many (Muitos para Muitos): Por exemplo, um livro possui muitos autores e um autor possui muitos livros.



One to One

Suponha que em nosso domínio existam duas entidades: Estado e Governador. Devemos criar uma classe para cada entidade e editar o arquivo `orm.xml` de acordo.

```
1 package br.com.k19;
2
3 class Estado {
4   private Long id;
5 }
```

Código Java D.13: Estado.java

```
1 package br.com.k19;
2
3 class Governador {
4   private Long id;
5 }
```

Código Java D.14: Governador.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Estado">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7      </attributes>
8    </entity>
9
10   <entity class="br.com.k19.Governador">
```

```
11      <attributes>
12        <id name="id">
13          <generated-value />
14        </id>
15      </attributes>
16    </entity>
17  </entity-mappings>
```

Código XML D.16: orm.xml

Como existe um relacionamento entre estados e governadores, devemos expressar esse vínculo através de um atributo que pode ser inserido na classe `Estado`.

```
1  package br.com.k19;
2
3  class Estado {
4    private Long id;
5
6    private Governador governador;
7  }
```

Código Java D.15: Estado.java

Além disso, devemos informar ao provedor JPA que o relacionamento que existe entre um estado e um governador é do tipo One to One. Fazemos isso aplicando usando a tag **<one-to-one>** no atributo que expressa o relacionamento.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Estado">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7
8        <one-to-one name="governador" />
9      </attributes>
10   </entity>
11
12   <entity class="br.com.k19.Governador">
13     <attributes>
14       <id name="id">
15         <generated-value />
16       </id>
17     </attributes>
18   </entity>
19 </entity-mappings>
```

Código XML D.17: orm.xml

No banco de dados, a tabela referente à classe `Estado` possuirá uma coluna de relacionamento chamada de **join column**. Em geral, essa coluna será definida como uma chave estrangeira associada à tabela referente à classe `Governador`.

Por padrão, o nome da coluna de relacionamento é formado pelo nome do atributo que estabelece o relacionamento, seguido pelo caractere "_" e pelo nome do atributo que define a chave primária da entidade alvo. No exemplo de estados e governadores, a join column teria o nome **governador_id**.

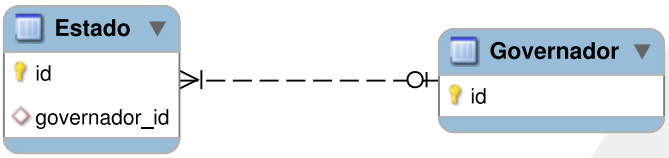


Figura D.5: Tabelas correspondentes às classes `Estado` *e* `Governador`

Podemos alterar o nome padrão das join columns usando a tag **<join-column>**, conforme apresentado no exemplo abaixo.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Estado">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7
8        <one-to-one name="governador">
9          <join-column name="gov_id"/>
10       </one-to-one>
11     </attributes>
12   </entity>
13
14   ...
15 </entity-mappings>
```

Código XML D.18: orm.xml

Figura D.6: Personalizando o nome da coluna de relacionamento

Mais Sobre

Por padrão, em um relacionamento One to One, um objeto da primeira entidade não precisa estar necessariamente relacionado a um objeto da segunda entidade. Para exigir que cada objeto da primeira entidade esteja relacionado a um objeto da segunda entidade, devemos usar o atributo **optional** da tag `<one-to-one>`.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Estado">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7
8        <one-to-one name="governador" optional="false" />
9      </attributes>
10   </entity>
11
12   ...
13 </entity-mappings>
```

Código XML D.19: orm.xml

Exercícios de Fixação

8 Implemente duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**: `Estado` e `Governador`.

```
1  public class Governador {
2    private Long id;
3    private String nome;
4
5    // GETTERS AND SETTERS
6  }
```

Código Java D.16: Governador.java

```
1  public class Estado {
2    private Long id;
3    private String nome;
4    private Governador governador;
5
6    // GETTERS AND SETTERS
7  }
```

Código Java D.17: Estado.java

9 Edite o arquivo `orm.xml` da pasta `src/META-INF` do projeto **K19-Mapeamento-XML** de acordo.

```
1  ...
2  <entity class="br.com.k19.modelo.Estado">
3    <attributes>
4      <id name="id">
5        <generated-value />
6      </id>
7
8      <one-to-one name="governador" />
9    </attributes>
10 </entity>
11
12 <entity class="br.com.k19.modelo.Governador">
13   <attributes>
14     <id name="id">
15       <generated-value />
16     </id>
17   </attributes>
18 </entity>
19 ...
```

Código XML D.20: orm.xml

10 Adicione um governador e um estado no banco de dados. Crie uma classe chamada `AdicionaEstadoGovernador` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento-XML**.

```
1  public class AdicionaEstadoGovernador {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_mapeamento_xml_pu");
5      EntityManager manager = factory.createEntityManager();
6
```

```
7     manager.getTransaction().begin();
8
9     Governador g = new Governador();
10    g.setNome("Rafael Cosentino");
11
12    Estado e = new Estado();
13    e.setNome("São Paulo");
14    e.setGovernador(g);
15
16    manager.persist(g);
17    manager.persist(e);
18
19    manager.getTransaction().commit();
20
21    manager.close();
22    factory.close();
23  }
24 }
```

Código Java D.18: AdicionaEstadoGovernador.java

11 Abra o MySQL Workbench e observe as propriedades das tabelas `Estado` e `Governador` da base de dados **K21_mapeamento_xml_bd**.

One to Many

Suponha que em nosso domínio existam as entidades Departamento e Funcionário. Criaríamos duas classes com as anotações básicas de mapeamento.

```
1 package br.com.k19;
2
3 class Departamento {
4   private Long id;
5 }
```

Código Java D.19: Departamento.java

```
1 package br.com.k19;
2
3 class Funcionario {
4   private Long id;
5 }
```

Código Java D.20: Funcionario.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Departamento">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7      </attributes>
8    </entity>
9
10   <entity class="br.com.k19.Funcionario">
11     <attributes>
12       <id name="id">
13         <generated-value />
14       </id>
15     </attributes>
```

```
16   </entity>
17 </entity-mappings>
```

Código XML D.21: orm.xml

Como existe um relacionamento entre departamentos e funcionários, devemos expressar esse vínculo através de um atributo que pode ser inserido na classe `Departamento`. Supondo que um departamento possa ter muitos funcionários, devemos utilizar uma coleção para expressar esse relacionamento.

```
1 package br.com.k19;
2
3 class Departamento {
4   private Long id;
5
6   private Collection<Funcionario> funcionarios;
7 }
```

Código Java D.21: Departamento.java

Para informar a cardinalidade do relacionamento entre departamentos e funcionários, devemos usar a tag **<one-to-many** na coleção.

```
1  <entity-mappings>
2    ...
3
4    <entity class="br.com.k19.Funcionario">
5      <attributes>
6        <id name="id">
7          <generated-value />
8        </id>
9
10       <one-to-many name="funcionarios" />
11     </attributes>
12   </entity>
13 </entity-mappings>
```

Código XML D.22: orm.xml

No banco de dados, além das duas tabelas correspondentes às classes `Departamento` e `Funcionario`, deve existir uma terceira tabela para relacionar os registros dos departamentos com os registros dos funcionários. Essa terceira tabela é chamada de tabela de relacionamento ou **join table**.

Por padrão, o nome da join table é a concatenação com "_" dos nomes das duas entidades. No exemplo de departamentos e funcionários, o nome do join table seria **Departamento_Funcionario**. Essa tabela possuirá duas colunas vinculadas às entidades que formam o relacionamento. No exemplo, a join table `Departamento_Funcionario` possuirá uma coluna chamada **Departamento_id** e outra chamada **funcionarios_id**.

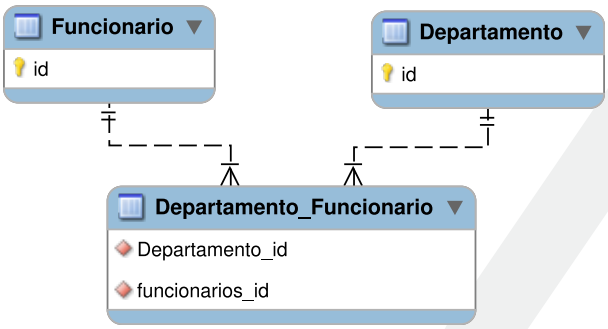


Figura D.7: Tabelas correspondentes às classes `Departamento` *e* `Funcionario`

Para personalizar os nomes das colunas da join table e da própria join table, podemos usar a tag **<join-table>** no atributo que define o relacionamento.

```
1  <entity-mappings>
2    ...
3
4    <entity class="br.com.k19.Funcionario">
5      <attributes>
6        <id name="id">
7          <generated-value />
8        </id>
9
10       <one-to-many name="funcionarios">
11         <join-table name="DEP_FUNC">
12           <join-column name="DEP_ID"/>
13           <inverse-join-column name="FUNC_ID"/>
14         </join-table>
15       </one-to-many>
16     </attributes>
17   </entity>
18 </entity-mappings>
```

Código XML D.23: orm.xml

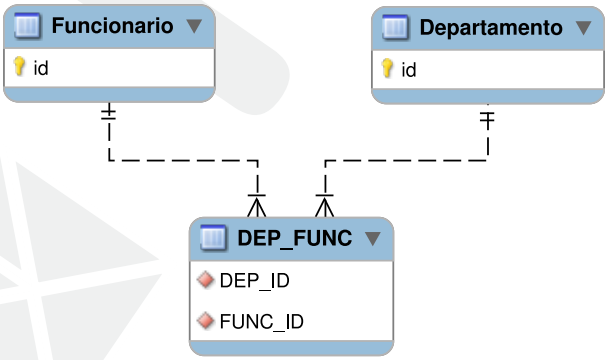


Figura D.8: Personalizando a tabela de relacionamento

Exercícios de Fixação

12 Implemente duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**: `Departamento` e `Funcionario`.

```
1  public class Funcionario {
2    private Long id;
3    private String nome;
4
5    // GETTERS AND SETTERS
6  }
```

Código Java D.22: Funcionario.java

```
1  public class Departamento {
2    private Long id;
3    private String nome;
4    private Collection<Funcionario> funcionarios = new ArrayList<Funcionario>();
5
6    // GETTERS AND SETTERS
7  }
```

Código Java D.23: Departamento.java

13 Edite o arquivo `orm.xml` da pasta `src/META-INF` do projeto **K19-Mapeamento-XML** de acordo.

```
1  ...
2  <entity class="br.com.k19.modelo.Funcionario">
3    <attributes>
4      <id name="id">
5        <generated-value />
6      </id>
7    </attributes>
8  </entity>
9
10 <entity class="br.com.k19.modelo.Departamento">
11   <attributes>
12     <id name="id">
13       <generated-value />
14     </id>
15
16     <one-to-many name="funcionarios" />
17   </attributes>
18 </entity>
19 ...
```

Código XML D.24: orm.xml

14 Adicione um departamento e um funcionário no banco de dados. Crie uma classe chamada `AdicionaDepartamentoFuncionario` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento-XML**.

```
1  public class AdicionaDepartamentoFuncionario {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_mapeamento_xml_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Funcionario f = new Funcionario();
10     f.setNome("Rafael Cosentino");
11
```

```
12      Departamento d = new Departamento();
13      d.setNome("Financeiro");
14      d.getFuncionarios().add(f);
15
16      manager.persist(f);
17      manager.persist(d);
18
19      manager.getTransaction().commit();
20
21      manager.close();
22      factory.close();
23    }
24  }
```

Código Java D.24: AdicionaDepartamentoFuncionario.java

15 Abra o MySQL Workbench e observe as propriedades das tabelas `Departamento`, `Funcionario` e `Departamento_Funcionario` da base de dados **K21_mapeamento_xml_bd**.

Many to One

Suponha que em nosso domínio existam as entidades Pedido e Cliente. As duas classes que modelariam essas entidades seriam definidas com as anotações principais de mapeamento.

```
1  package br.com.k19;
2
3  class Pedido {
4    private Long id;
5  }
```

Código Java D.25: Pedido.java

```
1  package br.com.k19;
2
3  class Cliente {
4    private Long id;
5  }
```

Código Java D.26: Cliente.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pedido">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7      </attributes>
8    </entity>
9
10   <entity class="br.com.k19.Cliente">
11     <attributes>
12       <id name="id">
13         <generated-value />
14       </id>
15     </attributes>
16   </entity>
17 </entity-mappings>
```

Código XML D.25: orm.xml

Como existe um relacionamento entre pedidos e clientes, devemos expressar esse vínculo através de um atributo que pode ser inserido na classe `Pedido`. Supondo que um pedido pertença a um único cliente, devemos utilizar um atributo simples para expressar esse relacionamento.

```
1  package br.com.k19;
2
3  class Pedido {
4    private Long id;
5    private Cliente cliente;
6  }
```

Código Java D.27: Pedido.java

Para informar a cardinalidade do relacionamento entre pedidos e clientes, devemos utilizar a tag **<many-to-one>**.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pedido">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7
8        <many-to-one name="cliente" />
9      </attributes>
10   </entity>
11
12   ...
13 </entity-mappings>
```

Código XML D.26: orm.xml

No banco de dados, a tabela referente à classe `Pedido` possuirá uma **join column** vinculada à tabela da classe `Cliente`. Por padrão, o nome da join column é formado pelo nome da entidade alvo do relacionamento, seguido pelo caractere "_" e pelo nome do atributo que define a chave primária da entidade alvo.

Figura D.9: Tabelas correspondentes às classes `Pedido` e `Cliente`

No exemplo de pedidos e clientes, o nome da join column seria **cliente_id**. Podemos alterar o nome padrão das join columns usando a tag **<join-column>**.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pedido">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7
8        <many-to-one name="cliente">
9          <join-column name="cli_id"/>
10       </many-to-one>
11     </attributes>
12   </entity>
```

```
13
14    ...
15  </entity-mappings>
```

Código XML D.27: orm.xml

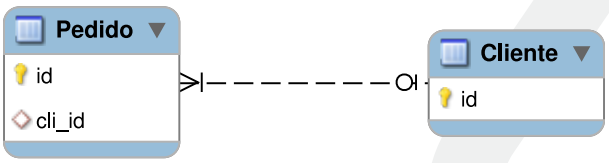


Figura D.10: Personalizando a tabela `Pedido`

Mais Sobre

Por padrão, em um relacionamento Many to One, um objeto da primeira entidade não precisa estar necessariamente relacionado a um objeto da segunda entidade. Para exigir que cada objeto da primeira entidade esteja relacionado a um objeto da segunda entidade, devemos usar o atributo **optional** da tag `many-to-one`.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pedido">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7
8        <many-to-one name="cliente" optional="false" />
9      </attributes>
10   </entity>
11
12   ...
13 </entity-mappings>
```

Código XML D.28: orm.xml

Exercícios de Fixação

16 Implemente duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**: `Pedido` e `Cliente`.

```
1  public class Cliente {
2    private Long id;
3    private String nome;
4
5    // GETTERS AND SETTERS
6  }
```

Código Java D.28: Cliente.java

```
1  public class Pedido {
2    private Long id;
3    private Calendar data;
```

```
4   private Cliente cliente;
5
6   // GETTERS AND SETTERS
7 }
```

Código Java D.29: Pedido.java

17 Edite o arquivo `orm.xml` da pasta `src/META-INF` do projeto **K19-Mapeamento-XML** de acordo.

```
1  ...
2  <entity class="br.com.k19.modelo.Pedido">
3    <attributes>
4      <id name="id">
5        <generated-value />
6      </id>
7
8      <basic name="data">
9        <temporal>DATE</temporal>
10     </basic>
11
12     <many-to-one name="cliente" />
13   </attributes>
14 </entity>
15
16 <entity class="br.com.k19.modelo.Cliente">
17   <attributes>
18     <id name="id">
19       <generated-value />
20     </id>
21   </attributes>
22 </entity>
23 ...
```

Código XML D.29: orm.xml

18 Adicione um cliente e um departamento no banco de dados. Crie uma classe chamada `AdicionaPedidoCliente` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento-XML**.

```
1  public class AdicionaPedidoCliente {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_mapeamento_xml_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
9      Cliente c = new Cliente();
10     c.setNome("Rafael Cosentino");
11
12     Pedido p = new Pedido();
13     p.setData(Calendar.getInstance());
14     p.setCliente(c);
15
16     manager.persist(c);
17     manager.persist(p);
18
19     manager.getTransaction().commit();
20
21     manager.close();
22     factory.close();
23   }
24 }
```

Código Java D.30: AdicionaPedidoCliente.java

19 Abra o MySQL Workbench e observe as propriedades das tabelas `Cliente` e `Pedido` da base de dados **K21_mapeamento_xml_bd**.

Many to Many

Suponha que em nosso domínio existam as entidades Livro e Autor. As classes com as anotações básicas de mapeamento seriam mais ou menos assim:

```
1 package br.com.k19;
2
3 class Livro {
4   private Long id;
5 }
```

Código Java D.31: Livro.java

```
1 package br.com.k19;
2
3 class Autor {
4   private Long id;
5 }
```

Código Java D.32: Autor.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Livro">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7      </attributes>
8    </entity>
9
10   <entity class="br.com.k19.Autor">
11     <attributes>
12       <id name="id">
13         <generated-value />
14       </id>
15     </attributes>
16   </entity>
17 </entity-mappings>
```

Código XML D.30: orm.xml

Como existe um relacionamento entre livros e autores, devemos expressar esse vínculo através de um atributo que pode ser inserido na classe `Livro`. Supondo que um livro possa ser escrito por muitos autores, devemos utilizar uma coleção para expressar esse relacionamento.

```
1 package br.com.k19;
2
3 class Livro {
4   private Long id;
5
6   private Collection<Autor> autores;
7 }
```

Código Java D.33: Livro.java

Para informar a cardinalidade do relacionamento entre livros e autores, devemos utilizar a tag **<many-to-many>**.

```
<entity-mappings>
  <entity class="br.com.k19.Livro">
    <attributes>
      <id name="id">
        <generated-value />
      </id>

      <many-to-many name="autores" />
    </attributes>
  </entity>

  ...
</entity-mappings>
```

Código XML D.31: orm.xml

No banco de dados, além das duas tabelas correspondentes às classes `Livro` e `Autor`, uma join table é criada para relacionar os registros dos livros com os registros dos autores. Por padrão, o nome da join table é a concatenação com "_" dos nomes das duas entidades. No exemplo de livros e autores, o nome do join table seria **Livro_Autor**. Essa tabela possuirá duas colunas vinculadas às entidades que formam o relacionamento. No exemplo, a join table `Livro_Autor` possuirá uma coluna chamada **Livro_id** e outra chamada **autores_id**.

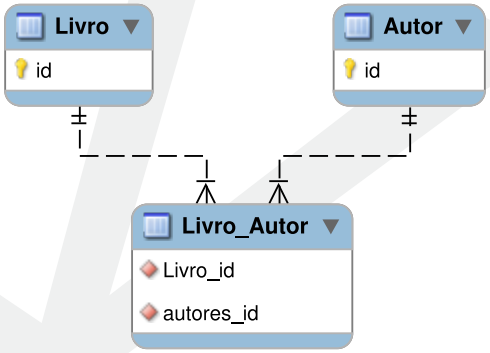


Figura D.11: Tabelas correspondentes às classes `Livro` *e* `Autor`

Para personalizar o nome join table e os nomes de suas colunas, podemos usar a tag **<join-table>** no atributo que define o relacionamento.

```
<entity-mappings>
  <entity class="br.com.k19.Livro">
    <attributes>
      <id name="id">
        <generated-value />
      </id>

      <many-to-many name="autores">
        <join-table name="Liv_Aut">
          <join-column name="Liv_ID"/>
          <inverse-join-column name="Aut_ID"/>
        </join-table>
      </many-to-many>
    </attributes>
  </entity>

  ...
```

```
18 </entity-mappings>
```

Código XML D.32: orm.xml

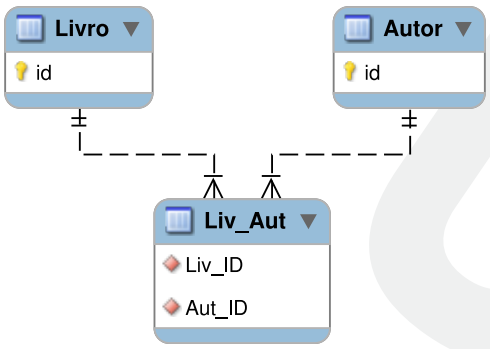


Figura D.12: Personalizando a tabela de relacionamento

Exercícios de Fixação

20 Implemente duas entidades no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**: `Livro` e `Autor`.

```
1 public class Autor {
2   private Long id;
3   private String nome;
4
5   // GETTERS AND SETTERS
6 }
```

Código Java D.34: Autor.java

```
1 public class Livro {
2   private Long id;
3   private String nome;
4   private Collection<Autor> autores = new ArrayList<Autor>();
5
6   // GETTERS AND SETTERS
7 }
```

Código Java D.35: Livro.java

21 Edite o arquivo `orm.xml` da pasta `src/META-INF` do projeto **K19-Mapeamento-XML** de acordo.

```
1 ...
2 <entity class="br.com.k19.modelo.Livro">
3   <attributes>
4     <id name="id">
5       <generated-value />
6     </id>
7
8     <many-to-many name="autores" />
9   </attributes>
10 </entity>
11
```

```
<entity class="br.com.k19.modelo.Autor">
  <attributes>
    <id name="id">
      <generated-value />
    </id>
  </attributes>
</entity>
...
```

Código XML D.33: orm.xml

22 Adicione um livro e um autor no banco de dados. Crie uma classe chamada `AdicionaLivroAutor` no pacote `br.com.k19.testes` do projeto **K19-Mapeamento-XML**.

```
public class AdicionaLivroAutor {
  public static void main(String[] args) {
    EntityManagerFactory factory =
      Persistence.createEntityManagerFactory("K21_mapeamento_xml_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    Autor a = new Autor();
    a.setNome("Rafael Cosentino");

    Livro l = new Livro();
    l.setNome("JPA2");
    l.getAutores().add(a);

    manager.persist(a);
    manager.persist(l);

    manager.getTransaction().commit();

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

Código Java D.36: AdicionaLivroAutor.java

23 Abra o MySQL Workbench e observe as propriedades das tabelas `Livro`, `Autor` e `Livro_Autor` da base de dados **K21_mapeamento_xml_bd**.

Relacionamentos Bidirecionais

Quando expressamos um relacionamento colocando um atributo em uma das entidades, podemos acessar a outra entidade a partir da primeira. Por exemplo, considere o relacionamento entre governadores e estados.

```
class Estado {
  private Long id;
  private Governador governador;

  // GETTERS E SETTERS
}
```

Código Java D.37: Estado.java

```
class Governador {
  private Long id;

  // GETTERS E SETTERS
}
```

Código Java D.38: Governador.java

```
<entity-mappings>
  <entity class="br.com.k19.Estado">
    <attributes>
      <id name="id">
        <generated-value />
      </id>

      <one-to-one name="governador" />
    </attributes>
  </entity>

  <entity class="br.com.k19.Governador">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
    </attributes>
  </entity>

</entity-mappings>
```

Código XML D.34: orm.xml

Como o relacionamento está definido na classe `Estado`, podemos acessar o governador a partir de um estado.

```
Estado e = manager.find(Estado.class, 1L);
Governador g = e.getGovernador();
```

Código Java D.39: Acessando o governador a partir de um estado

Também podemos expressar o relacionamento na classe `Governador`. Dessa forma, poderíamos acessar um estado a partir de um governador.

```
class Governador {
  private Long id;
  private Estado estado;

  // GETTERS E SETTERS
}
```

Código Java D.40: Governador.java

```
<entity-mappings>
  ...

  <entity class="br.com.k19.Governador">
    <attributes>
      <id name="id">
        <generated-value />
      </id>

      <one-to-one name="estado" />
    </attributes>
  </entity>
```

```
13 </entity-mappings>
```

Código XML D.35: orm.xml

```
1 Governador g = manager.find(Governador.class, 1L);
2 Estado e = g.getEstado();
```

Código Java D.41: Acessando um estado a partir de um governador

A figura abaixo ilustra as tabelas `Estado` e `Governador` no banco de dados, assim como as join columns correspondentes aos relacionamentos.

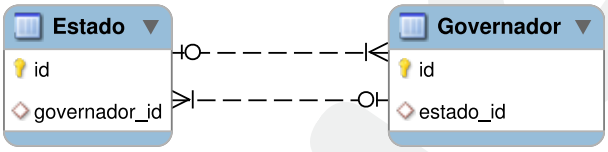


Figura D.13: Tabelas `Estado` e `Governador` no banco de dados

Note que foram criadas duas colunas de relacionamentos. A primeira na tabela `Estado` com o nome `governador_id` e a segunda na tabela `Governador` com o nome `estado_id`. Nesse caso, o provedor JPA está considerando dois relacionamentos unidirecionais distintos entre essas entidades.

No entanto, de acordo com o modelo relacional, a relação entre estados e governadores deveria ser expressa com apenas uma coluna de relacionamento. Ou seja, o relacionamento entre governadores e estados deveria ser bidirecional. Assim, devemos indicar em uma das classes que esse relacionamento bidirecional é a junção de dois relacionamentos unidirecionais. Para isso, devemos usar o atributo **mapped-by** da tag `<one-to-one>` em uma das entidades. O valor do `mapped-by` deve ser o nome do atributo que expressa o mesmo relacionamento na outra entidade.

```
1  <entity-mappings>
2    ...
3
4    <entity class="br.com.k19.Governador">
5      <attributes>
6        <id name="id">
7          <generated-value />
8        </id>
9
10       <one-to-one name="estado" mapped-by="governador"/>
11     </attributes>
12   </entity>
13 </entity-mappings>
```

Código XML D.36: orm.xml

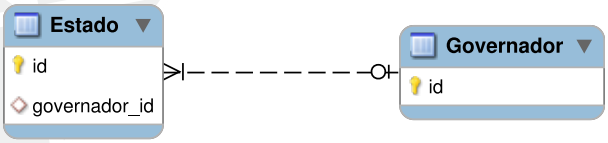


Figura D.14: Tabelas `Estado` e `Governador` no banco de dados

Exercícios de Fixação

24 Considere um sistema de cobrança de ligações telefônicas. Nesse sistema, temos uma entidade chamada Ligação e uma entidade chamada Fatura. Cada ligação está associada a uma única fatura, enquanto que uma fatura está associada a múltiplas ligações. Implemente classes para modelar essas duas entidades em uma pacote chamado `br.com.k19.modelo` no projeto **K19-Mapeamento-XML**.

```
1 public class Fatura {
2   private Long id;
3   private Calendar vencimento;
4   private Collection<Ligacao> ligacoes = new ArrayList<Ligacao>();
5
6   // GETTERS E SETTERS
7 }
```

Código Java D.42: Fatura.java

```
1 public class Ligacao {
2   private Long id;
3   private Integer duracao;
4   private Fatura fatura;
5
6   // GETTERS E SETTERS
7 }
```

Código Java D.43: Ligacao.java

25 Edite o arquivo `orm.xml` da pasta `src/META-INF` do projeto **K19-Mapeamento-XML** de acordo.

```
1  ...
2  <entity class="br.com.k19.modelo.Fatura">
3    <attributes>
4      <id name="id">
5        <generated-value />
6      </id>
7
8      <basic name = "vencimento">
9        <temporal>DATE</temporal>
10     </basic>
11
12     <one-to-many name="ligacoes" />
13   </attributes>
14 </entity>
15
16 <entity class="br.com.k19.modelo.Ligacao">
17   <attributes>
18     <id name="id">
19       <generated-value />
20     </id>
21
22     <many-to-one name="fatura" />
23   </attributes>
24 </entity>
25 ...
```

Código XML D.37: orm.xml

26 Crie um teste para adicionar algumas ligações e uma fatura no pacote **br.com.k19.testes** no projeto **K19-Mapeamento-XML**.

```java
public class AdicionaFaturaLigacao {
  public static void main(String[] args) {
    EntityManagerFactory factory =
      Persistence.createEntityManagerFactory("K21_mapeamento_xml_pu");
    EntityManager manager = factory.createEntityManager();

    manager.getTransaction().begin();

    Ligacao ligacao1 = new Ligacao();
    ligacao1.setDuracao(162);

    Ligacao ligacao2 = new Ligacao();
    ligacao2.setDuracao(324);

    Fatura fatura = new Fatura();
    fatura.setVencimento(new GregorianCalendar(2012, 11, 20));

    fatura.getLigacoes().add(ligacao1);
    fatura.getLigacoes().add(ligacao2);

    ligacao1.setFatura(fatura);
    ligacao2.setFatura(fatura);

    manager.persist(fatura);
    manager.persist(ligacao1);
    manager.persist(ligacao2);

    manager.getTransaction().commit();

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

Código Java D.44: AdicionaFaturaLigacao.java

27 Através do MySQL Workbench, verifique as tabelas criadas. Observe que a tabela `Ligacao` possui uma coluna de relacionamento chamada `fatura_id` e a tabela `Fatura_Ligacao` vincula os registros das tabelas `Ligacao` e `Fatura`.

28 Através do MySQL Workbench, apague primeiro a tabela `Fatura_Ligacao` e, em seguida, apague as tabelas `Fatura` e `Ligacao`.

29 Altere o código do arquivo `orm.xml` da pasta **src/META-INF** do projeto **K19-Mapeamento-XML** de forma a criar um relacionamento bidirecional entre as faturas e as ligações.

```
...
<entity class="br.com.k19.modelo.Fatura">
  <attributes>
    <id name="id">
      <generated-value />
    </id>

    <basic name = "vencimento">
      <temporal>DATE</temporal>
    </basic>

    <one-to-many name="ligacoes" mapped-by="fatura"/>
  </attributes>
</entity>

```

```
<entity class="br.com.k19.modelo.Ligacao">
  <attributes>
    <id name="id">
      <generated-value />
    </id>

    <many-to-one name="fatura" />
  </attributes>
</entity>
...
```

Código XML D.38: orm.xml

30 Execute a classe `AdicionaFaturaLigacao` para adicionar a futura e suas ligações. Através do MySQL Workbench, verifique as tabelas criadas. Note que foram criadas apenas duas tabelas: `Fatura` e `Ligacao`.

Objetos Embutidos

Suponha que em nosso domínio exista uma entidade chamada `Pessoa`. Toda pessoa possui um endereço, que é formado por país, estado, cidade, logradouro, número, complemento e CEP. Para melhorar a organização da nossa aplicação, podemos criar e mapear duas classes: `Pessoa` e `Endereco`.

```
class Pessoa {
  private Long id;
  private String nome;
  private Calendar nascimento;
  private Endereco endereco;
}
```

Código Java D.45: Pessoa.java

```
class Endereco {
  private Long id;

  private String pais;

  private String estado;

  private String cidade;

  private String logradouro;

  private int numero;

  private String complemento;

  private int cep;
}
```

Código Java D.46: Endereco.java

```
<entity-mappings>
  <entity class="br.com.k19.Pessoa">
    <attributes>
      <id name="id">
        <generated-value />
      </id>
```

```
7
8       <basic name = "nascimento">
9         <temporal>DATE</temporal>
10      </basic>
11
12      <one-to-one name="endereco"/>
13    </attributes>
14  </entity>
15
16  <entity class="br.com.k19.Endereco">
17    <attributes>
18      <id name="id">
19        <generated-value />
20      </id>
21    </attributes>
22  </entity>
23 </entity-mappings>
```

Código XML D.39: orm.xml

Da forma como os mapeamentos estão definidos, duas tabelas serão criadas: uma para a classe `Pessoa` e outra para a classe `Endereco`. Na tabela `Pessoa`, haverá uma coluna de relacionamento.

Para recuperar os dados do endereço de uma pessoa, duas tabelas precisam ser consultadas através de uma operação de join. Esse tipo de operação no banco de dados é custoso.

Suponha que a tabela `Endereco` esteja relacionada apenas com a tabela `Pessoa`. Nesse caso, seria interessante se pudéssemos guardar os endereços das pessoas na própria tabela `Pessoa`, tornando desnecessária a existência da tabela `Endereco`. No entanto, gostaríamos de manter as classes `Pessoa` e `Endereco`.

Isso pode ser feito da seguinte forma. No arquivo `orm.xml`, devemos remover o elemento de cardinalidade `<one-to-one>` e o mapeamento da entidade `Endereco`. Além disso, devemos acrescentar o elemento `<embeddable/>` para mapear a classe `Endereco`. Além disso, não devemos definir uma chave para a classe `Endereco`, pois ela não define uma entidade.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pessoa">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7
8        <basic name = "nascimento">
9          <temporal>DATE</temporal>
10       </basic>
11     </attributes>
12   </entity>
13
14   <embeddable class="br.com.k19.Endereco"/>
15 </entity-mappings>
```

Código XML D.40: orm.xml

```
1  class Endereco {
2    private String pais;
3
4    private String estado;
5
6    private String cidade;
7
8    private String logradouro;
9
```

```
10   private int numero;
11
12   private String complemento;
13
14   private int cep;
15 }
```

Código Java D.47: Endereco.java

Podemos conseguir o mesmo resultado da seguinte forma. Na arquivo `orm.xml`, devemos substituir o elemento de cardinalidade `<One-to-one>` por `<embedded>` e remover o mapeamento da entidade `Endereco`. Também, não devemos definir uma chave para a classe `Endereco`, pois ela não define uma entidade.

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pessoa">
3      <attributes>
4        <id name="id">
5          <generated-value />
6        </id>
7
8        <basic name = "nascimento">
9          <temporal>DATE</temporal>
10       </basic>
11
12       <embedded name="endereco"/>
13     </attributes>
14   </entity>
15 </entity-mappings>
```

Código XML D.41: orm.xml

```
1  class Endereco {
2    private String pais;
3
4    private String estado;
5
6    private String cidade;
7
8    private String logradouro;
9
10   private int numero;
11
12   private String complemento;
13
14   private int cep;
15 }
```

Código Java D.48: Endereco.java

Exercícios de Fixação

31 Crie uma classe para modelar endereços no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**.

```
1  public class Endereco {
2
3    private String estado;
4
```

```
5   private String cidade;
6
7   private String logradouro;
8
9   private int numero;
10
11  // GETTERS AND SETTERS
12 }
```

Código Java D.49: Endereco.java

32 No pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**, crie uma classe chamada `Candidato`.

```
1  public class Candidato {
2    private Long id;
3
4    private String nome;
5
6    private Calendar nascimento;
7
8    private Endereco endereco;
9
10   // GETTERS E SETTERS
11 }
```

Código Java D.50: Candidato.java

33 Altere o código do arquivo `orm.xml` da pasta **src/META-INF** do projeto **K19-Mapeamento-XML** de forma a criar um relacionamento bidirecional entre as faturas e as ligações.

```
1  ...
2  <entity class="br.com.k19.modelo.Candidato">
3    <attributes>
4      <id name="id">
5        <generated-value />
6      </id>
7
8      <basic name = "nascimento">
9        <temporal>DATE</temporal>
10     </basic>
11
12     <embedded name="endereco"/>
13   </attributes>
14 </entity>
15 ...
```

Código XML D.42: orm.xml

34 Crie uma classe chamada `AdicionaCandidatoEndereco` para adicionar alguns candidatos e endereços no pacote `br.com.k19.testes` do projeto **K19-Mapeamento-XML**.

```
1  public class AdicionaCandidatoEndereco {
2    public static void main(String[] args) {
3      EntityManagerFactory factory =
4        Persistence.createEntityManagerFactory("K21_mapeamento_xml_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      manager.getTransaction().begin();
8
```

```
9       Endereco e = new Endereco();
10      e.setEstado("São Paulo");
11      e.setCidade("São Paulo");
12      e.setLogradouro("Av. Bigadeiro Faria Lima");
13      e.setNumero(1571);
14
15      Pessoa p = new Pessoa();
16      p.setNome("Rafael Cosentino");
17      p.setNascimento(new GregorianCalendar(1984, 10, 30));
18      p.setEndereco(e);
19
20      manager.persist(p);
21
22      manager.getTransaction().commit();
23
24      manager.close();
25      factory.close();
26    }
27  }
```

Código Java D.51: AdicionaCandidatoEndereco.java

Herança

O mapeamento objeto-relacional descreve como os conceitos de orientação a objetos são mapeados para os conceitos do modelo relacional. De todos os conceitos de orientação a objetos, um dos mais complexos de se mapear é o de Herança.

A especificação JPA define três estratégias para realizar o mapeamento de herança.

- Single Table
- Joined
- Table Per Class

Single Table

A estratégia Single Table é a mais comum e a que possibilita melhor desempenho em relação a velocidade das consultas. Nessa estratégia, a super classe deve ser mapeada com

`<inheritance strategy=SINGLE_TABLE>`.

O provedor JPA criará apenas uma tabela com o nome da super classe para armazenar os dados dos objetos criados a partir da super classe ou das sub classes. Todos os atributos da super classe e os das sub classes serão mapeados para colunas dessa tabela. Além disso, uma coluna especial chamada **DTYPE** será utilizada para identificar a classe do objeto correspondente ao registro.

```
1  public class Pessoa {
2    private Long id;
3
4    private String nome;
5  }
```

Código Java D.52: Pessoa.java

```java
public class PessoaJuridica extends Pessoa {
  private String cnpj;
}
```

Código Java D.53: PessoaJuridica.java

```java
public class PessoaFisica extends Pessoa {
  private String cpf;
}
```

Código Java D.54: PessoaFisica.java

```xml
<entity-mappings>
  <entity class="br.com.k19.Pessoa">
    <inheritance strategy="SINGLE_TABLE"/>
    <attributes>
      <id name="id">
        <generated-value />
      </id>
    </attributes>
  </entity>

  <entity class="br.com.k19.PessoaJuridica">

  </entity>

  <entity class="br.com.k19.PessoaFisica">

  </entity>
</entity-mappings>
```

Código XML D.43: orm.xml

Figura D.15: Tabela correspondente às classes `Pessoa`, `PessoaJuridica` *e* `PessoaFisica`

A desvantagem da Single Table é o consumo desnecessário de espaço, já que nem todos os campos são utilizados para todos os registros. Por exemplo, se uma pessoa jurídica fosse cadastrada, o campo `cpf` não seria utilizado. Da mesma forma, se uma pessoa física fosse cadastrada, o campo `cnpj` não seria utilizado.

Joined

Nessa estratégia, uma tabela para cada classe da hierarquia é criada. Em cada tabela, apenas os campos referentes aos atributos da classe correspondente são adicionados. Para relacionar os registros das diversas tabelas e remontar os objetos quando uma consulta for realizada, as tabelas relacionadas às sub-classes possuem chaves estrangeiras vinculadas à tabela associada à super-classe.

```java
public class Pessoa {
```

```
2   private Long id;
3
4   private String nome;
5 }
```

Código Java D.55: Pessoa.java

```
1 public class PessoaJuridica extends Pessoa {
2   private String cnpj;
3 }
```

Código Java D.56: PessoaJuridica.java

```
1 public class PessoaFisica extends Pessoa {
2   private String cpf;
3 }
```

Código Java D.57: PessoaFisica.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pessoa">
3      <inheritance strategy="JOINED"/>
4      <attributes>
5        <id name="id">
6          <generated-value />
7        </id>
8      </attributes>
9    </entity>
10
11   <entity class="br.com.k19.PessoaJuridica">
12
13   </entity>
14
15   <entity class="br.com.k19.PessoaFisica">
16
17   </entity>
18 </entity-mappings>
```

Código XML D.44: orm.xml

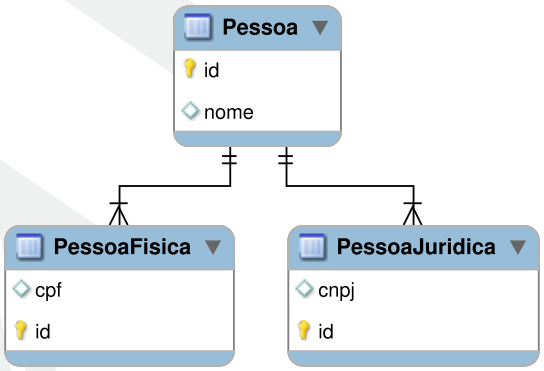


Figura D.16: Tabelas correspondentes às classes `Pessoa`, `PessoaJuridica` *e* `PessoaFisica`

O consumo de espaço utilizando a estratégia **Joined** é menor do que o utilizado pela estratégia **Single Table**. Contudo, as consultas são mais lentas, pois é necessário realizar operações de **join** para recuperar os dados dos objetos.

Table Per Class

Nessa estratégia, uma tabela para cada classe concreta da hierarquia é criada. Contudo, os dados de um objeto não são colocados em tabelas diferentes. Dessa forma, para remontar um objeto não é necessário realizar operações de **join**. A desvantagem desse modo é que não existe um vínculo explícito no banco de dados entres as tabelas correspondentes às classes da hierarquia.

```
1 public class Pessoa {
2   private Long id;
3
4   private String nome;
5 }
```

Código Java D.58: Pessoa.java

```
1 public class PessoaJuridica extends Pessoa {
2   private String cnpj;
3 }
```

Código Java D.59: PessoaJuridica.java

```
1 public class PessoaFisica extends Pessoa{
2   private String cpf;
3 }
```

Código Java D.60: PessoaFisica.java

```
1  <entity-mappings>
2    <entity class="br.com.k19.Pessoa">
3      <inheritance strategy="TABLE_PER_CLASS"/>
4      <attributes>
5        <id name="id">
6          <generated-value />
7        </id>
8      </attributes>
9    </entity>
10
11   <entity class="br.com.k19.PessoaJuridica">
12
13   </entity>
14
15   <entity class="br.com.k19.PessoaFisica">
16
17   </entity>
18 </entity-mappings>
```

Código XML D.45: orm.xml

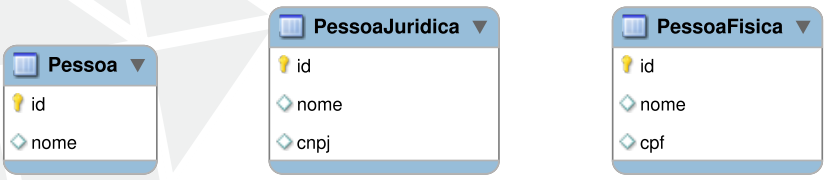


Figura D.17: Tabelas correspondentes às classes `Pessoa`, `PessoaJuridica` *e* `PessoaFisica`

Na estratégia **Table Per Class**, não podemos utilizar a geração automática de chave primárias simples e numéricas.

Exercícios de Fixação

35 Adicione uma classe chamada `Pessoa` no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**.

```
1  public class Pessoa {
2    private Long id;
3
4    private String nome;
5
6    // GETTERS E SETTERS
7  }
```

Código Java D.61: Pessoa.java

36 Faça a classe `PessoaJuridica` no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**.

```
1  public class PessoaJuridica extends Pessoa {
2    private String cnpj;
3
4    // GETTERS E SETTERS
5  }
```

Código Java D.62: PessoaJuridica.java

37 Faça a classe `PessoaFisica` no pacote `br.com.k19.modelo` do projeto **K19-Mapeamento-XML**.

```
1  public class PessoaFisica extends Pessoa {
2    private String cpf;
3
4    // GETTERS E SETTERS
5  }
```

Código Java D.63: PessoaFisica.java

38 Altere o código do arquivo `orm.xml` da pasta **src/META-INF** do projeto **K19-Mapeamento-XML** de forma a criar um relacionamento bidirecional entre as faturas e as ligações.

```
1  ...
2  <entity class="br.com.k19.modelo.Pessoa">
3    <inheritance strategy="SINGLE_TABLE"/>
4    <attributes>
5      <id name="id">
6        <generated-value />
7      </id>
8    </attributes>
9  </entity>
10
11 <entity class="br.com.k19.modelo.PessoaJuridica">
12
13 </entity>
14
15 <entity class="br.com.k19.modelo.PessoaFisica">
```

```
16
17 </entity>
18 ...
```

Código XML D.46: orm.xml

39 Crie um teste para adicionar pessoas. No pacote `br.com.k19.testes` do projeto **K19-Mapeamento-XML** adicione a seguinte classe:

```
1  public class AdicionaPessoa {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence.createEntityManagerFactory("↩
           K21_mapeamento_xml_pu");
4      EntityManager manager = factory.createEntityManager();
5
6      manager.getTransaction().begin();
7
8      Pessoa p1 = new Pessoa();
9      p1.setNome("Marcelo");
10
11     PessoaFisica p2 = new PessoaFisica();
12     p2.setNome("Rafael");
13     p2.setCpf("1234");
14
15     PessoaJuridica p3 = new PessoaJuridica();
16     p3.setNome("K19");
17     p3.setCnpj("567788");
18
19     manager.persist(p1);
20     manager.persist(p2);
21     manager.persist(p3);
22
23     manager.getTransaction().commit();
24
25     manager.close();
26     factory.close();
27   }
28 }
```

Código Java D.64: AdicionaPessoa.java

RESPOSTAS

APÊNDICE

Exercício Complementar 1.1

```
1  package br.com.k19.modelo;
2
3  import javax.persistence.Entity;
4  import javax.persistence.GeneratedValue;
5  import javax.persistence.Id;
6
7  @Entity
8  public class Autor {
9
10   @Id
11   @GeneratedValue
12   private Long id;
13
14   private String nome;
15
16   // GETTERS E SETTERS
17 }
```

Código Java 1.4: Autor.java

Exercício Complementar 1.2

```
1  package br.com.k19.testes;
2
3  import javax.persistence.EntityManagerFactory;
4  import javax.persistence.Persistence;
5
6  public class GeraTabelas {
7    public static void main(String[] args) {
8      EntityManagerFactory factory =
9        Persistence.createEntityManagerFactory("K21_livraria_pu");
10
11     factory.close();
12   }
13 }
```

Código Java 1.5: GeraTabelas.java

Exercício Complementar 1.3

```
1  package br.com.k19.testes;
```

```
2
3  import java.util.Scanner;
4
5  import javax.persistence.EntityManager;
6  import javax.persistence.EntityManagerFactory;
7  import javax.persistence.Persistence;
8
9  import br.com.k19.modelo.Autor;
10
11 public class InsereAutorComJPA {
12
13   public static void main(String[] args) {
14     EntityManagerFactory factory =
15       Persistence.createEntityManagerFactory("K21_livraria_pu");
16
17     EntityManager manager = factory.createEntityManager();
18
19     Autor novoAutor = new Autor();
20
21     Scanner entrada = new Scanner(System.in);
22
23     System.out.println("Digite o nome do autor: ");
24     novoAutor.setNome(entrada.nextLine());
25
26     entrada.close();
27
28     manager.persist(novoAutor);
29
30     manager.getTransaction().begin();
31     manager.getTransaction().commit();
32
33     factory.close();
34   }
35 }
```

Código Java 1.17: InsereAutorComJPA.java

Execute a classe `InsereAutorComJPA`.

Exercício Complementar 1.4

```
1  package br.com.k19.testes;
2
3  import java.util.List;
4
5  import javax.persistence.EntityManager;
6  import javax.persistence.EntityManagerFactory;
7  import javax.persistence.Persistence;
8  import javax.persistence.Query;
9
10 import br.com.k19.modelo.Autor;
11
12 public class ListaAutoresComJPA {
13
14   public static void main(String[] args) {
15     EntityManagerFactory factory =
16       Persistence.createEntityManagerFactory("K21_livraria_pu");
17
18     EntityManager manager = factory.createEntityManager();
19
20     Query query = manager.createQuery("SELECT a FROM Autor a");
21     List<Autor> autores = query.getResultList();
22
23     for(Autor a : autores) {
```

```
24        System.out.println("AUTOR: " + a.getNome());
25      }
26    }
27  }
```

Código Java 1.18: ListaAutoresComJPA.java

Execute a classe `ListaAutoresComJPA`.

Exercício Complementar 5.1

```
1  public class ListaAutores {
2    public static void main(String[] args) {
3      EntityManagerFactory factory = Persistence
4          .createEntityManagerFactory("K21_criteria_pu");
5      EntityManager manager = factory.createEntityManager();
6
7      CriteriaBuilder cb = manager.getCriteriaBuilder();
8      CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
9      Root<Autor> l = c.from(Autor.class);
10     c.select(l);
11
12     TypedQuery<Autor> query = manager.createQuery(c);
13     List<Autor> autores = query.getResultList();
14
15     for (Autor autor : autores) {
16       System.out.println(autor.getNome());
17     }
18   }
19 }
```

Código Java 5.10: ListaAutores.java