Aula 2 - Revisão de JPA (Java Persistence API)

Professor: Ricardo Luis dos Santos

IFSUL – Campus Sapucaia do Sul

Tabela para Revisão

Assunto (JPA)	Interesse?
1 – Vantagens e Desvantagens	4
2 – Principais Conceitos	7
3 – Anotações	7
4 – Operações CRUD sobre entidades	9
5 – Mapeamento Objeto Relacional (ORM)	9
6 – JPQL (consultas)	9
7 – WebServices (REST)	10

Tabela para Revisão

Assunto (JPA)	Interesse?
1 – Vantagens e Desvantagens	**4**
2 – Principais Conceitos	**7**
3 – Anotações	**7**
4 – Operações CRUD sobre entidades	**9**
5 – Mapeamento Objeto Relacional (ORM)	**9**
6 – JPQL (consultas)	**9**
7 – WebServices (REST)	10

Agenda

- Introdução
- Principais Conceitos
- Implementando uma Entidade
- Mapeamento Objeto Relacional – ORM
- API – EntityManager
- Linguagem de consulta – JPQL
- Resumo
- Bibliografia

Introdução

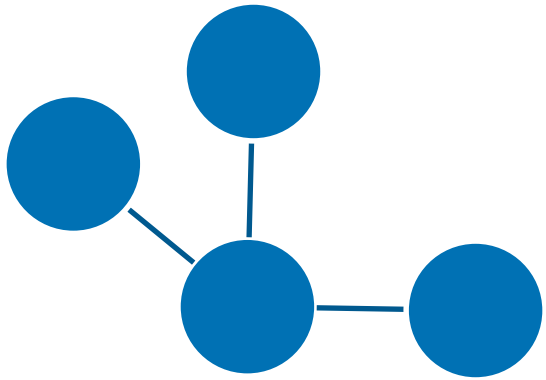
- Mapear o **modelo entidade-relacionamento** do banco de dados para o **modelo de orientação à objetos** é **extremamente complexo**
  - Demanda um grande quantidade tempo dos programadores
  - Dificuldade em mapear herança, agregações e composições de uma forma eficiente
  - Diversos problemas relacionados ao desempenho
  - Redução da produtividade para a escrita de *queries* SQL

Introdução

- Mapear o **modelo entidade-relacionamento** do banco de dados para o **modelo de orientação à objetos** é **extremamente complexo**
  - Redução da flexibilidade pela utilização de SQL específica a um determinado banco de dados
  - Dilema entre mapear as regras de negócio no banco de dados e na aplicação
  - Embora existam banco de dados orientados à objetos, banco de dados relacionais são o padrão para o mercado

Introdução

- Principal Problema

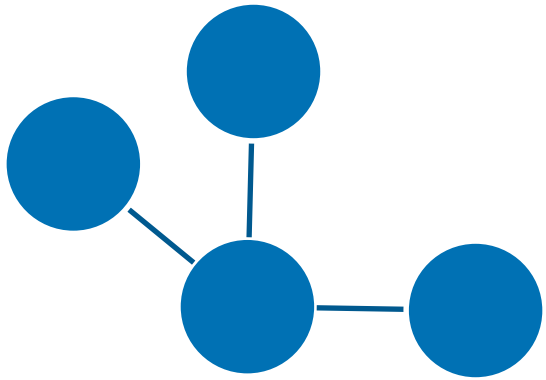


Programação Orientada à Objetos

≠

Introdução

- Principal Problema



Programação Orientada à Objetos

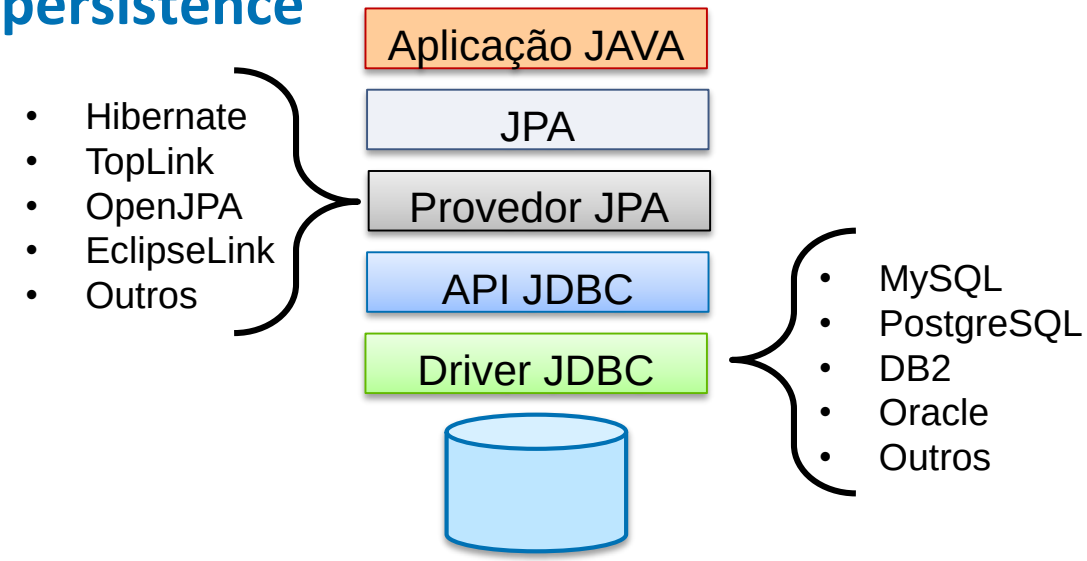
Java Persistence API

JPA

Banco de Dados Relacional

Principais Conceitos

- Java Persistence API
  - É uma abstração (especificação) em cima do JDBC que torna possível a independência do SQL
  - Utiliza anotações para o mapear os elementos
  - As classes e anotações da API JPA estão no pacote **javax.persistence**



Principais Conceitos

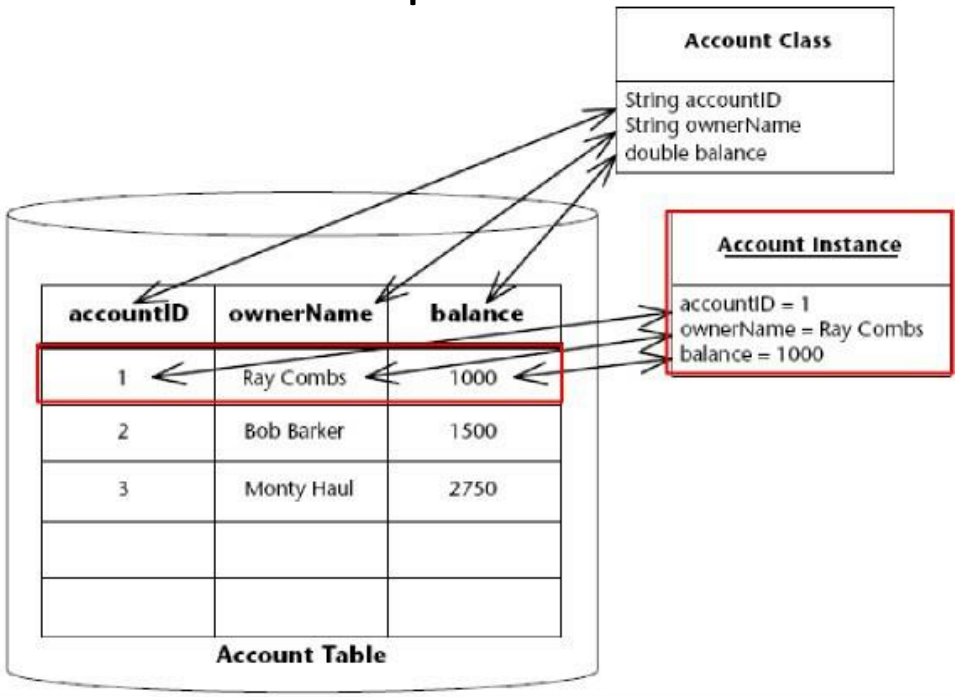
- Principais Componentes
  - Mapeamento Objeto Relacional (ORM – Object-Relational Mapping
    - Permite mapear entidades e seus relacionamentos (tabelas) em objetos que podem ser persistidos (Entities)
  - Uma API (EntityManager)
    - Permite desempenhar operações CRUD sobre um banco de dados (Criar, Ler, Atualizar e Remover)
  - Uma linguagem de consulta (JPQL – Java Persistence Query Language)
    - Permite recuperar dados com uma linguagem de consulta Orientada à Objetos

Principais Conceitos

- Quando mapeamos objetos para um BD relacional, para persisti-los ou consultá-los o termo **entidade** deve ser usado
  - Objetos são apenas mantidos em memória
  - Entidades são objetos que são mantidas na memória durante um algum intervalo de tempo e, então persistidas no banco de dados
    - Suportam herança, relacionamentos, entre outros. Desde que mapeados para o gerenciamento do JPA
  - Uma vez persistidas, as entidades podem ser consultadas através da JPQL

Principais Conceitos

- Entidade
  - A classe da entidade representa uma tabela do banco de dados
  - Um objeto da entidade representa uma linha na tabela



Implementando uma Entidade

- Um exemplo de entidade

```
1  import javax.persistence.Column;
2  import javax.persistence.Entity;
3  import javax.persistence.GeneratedValue;
4  import javax.persistence.Id;
5
6
7  @Entity
8  public class Livro {
9
10     @Id @GeneratedValue
11     private Long id;
12     @Column(nullable=false)
13     private String titulo;
14     private Float preco;
15     @Column(length=2000)
16     private String descricao;
17     private Integer numPaginas;
18
19     //contrutores, métodos, getters e setters
20
21 }
```

Implementando uma Entidade

- Um exemplo de entidade

```
1  import javax.persistence.Column;
2  import javax.persistence.Entity;
3  import javax.persistence.GeneratedValue;
4  import javax.persistence.Id;
5
6
7  @Entity
8  public class Livro {
9
10     @Id @GeneratedValue
11     private Long id;
12     @Column(nullable=false)
13     private String titulo;
14     private Float preco;
15     @Column(length=2000)
16     private String descricao;
17     private Integer numPaginas;
18
19     //contrutores, métodos, getters e setters
20
21 }
```

Importando os pacotes necessários

Indica que objetos dessa classe se tornem “persistível” no banco de dados

@Id indica que o atributo id é nossa chave primária

@GeneratedValue diz que queremos que esta chave seja populada pelo banco (auto increment)

Indica determinados atributos para uma coluna da tabela

Implementando uma Entidade

- Principais regras
  - A entidade deve ser anotada com @javax.persistence.Entity
  - A anotação @javax.persistence.Id deve ser usada para criar uma chave primária simples
  - A classe entidade deve possuir um construtor sem argumentos o qual deve ser *public* ou *protected*
  - A classe entidade não pode ser um *enum* ou *interface*, apenas *class* é permitido
  - A classe entidade não deve ser *final*. Nenhum dos métodos ou variáveis de instância pode ser *final*

Implementando uma Entidade

- Configurações por Exceção
  - Desde que as entidades respeitem as regras, o provedor de persistência pode fazer o mapeamento considerando algumas convenções
    - O nome da classe é mapeado como o nome da tabela
      - Para mudar o nome use a anotação @Table
  - Os nomes dos atributos são mapeados como nome das colunas
    - Para mudar o nome use a anotação @Column
  - Para mapear os tipos de dados, as mesmas regras do JDBC são válidas
    - Por exemplo, String mapeia para VARCHAR (padrão de 255 caracteres)

Mapeamento Objeto Relacional – ORM

- Tabelas
  - @Table
    - Pode-se definir propriedades da tabela, como seu nome
      - Por padrão, os nomes das tabelas são criadas com letra maiúscula, assim como nome da classe
    - Ex: @Table(name = “livro")

Mapeamento Objeto Relacional – ORM

- Tabelas
  - @SecondaryTable
    - Permite que os atributos de uma Entidade (classe) possa ser distribuído entre mais de uma tabela
    - Para várias tabelas secundárias use @SecondaryTables
    - Colunas que não especificarem a sua tabela serão mapeadas para a tabela primária
    - Importante: considere questões de desempenho se optar por utilizar tabelas secundárias

Mapeamento Objeto Relacional – ORM

- Tabelas
  - @SecondaryTable

```
import ...8 linhas

@Entity
@Table(name = "endereco")
@SecondaryTables({
    @SecondaryTable(name="cidade"),
    @SecondaryTable(name="pais")
})
public class FirstTable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String rua1;
    private String rua2;

    @Column(table="cidade")
    private String cidade;
    @Column(table="cidade")
    private String estado;
    @Column(table="cidade")
    private String cep;

    @Column(table="pais")
    private String pais;
}
```

Mapeamento Objeto Relacional – ORM

- Chaves Primárias
  - @Id
    - Define uma chave primária
    - Pode ser dos seguintes tipos:
      - Tipos primitivos: byte, int, short, long e char
      - Classes “wrapper”: Byte, Integer, Short, Long, Character
      - Strings, números e datas: String, BigInteger e Date

Mapeamento Objeto Relacional – ORM

- Chaves Primárias
  - @GeneratedValue
    - Define que a coluna tem um valor gerado automaticamente
    - O atributo **strategy**, que define a estratégia de geração de valores incrementados, possui quatro valores válidos:
      - SEQUENCE e IDENTITY: define uma coluna sequence ou identity, respectivamente.
      - TABLE: instrui o provedor de persistência a usar uma tabela para armazenar a “semente” da sequencia. É criada uma tabela com duas colunas - uma contendo o nome (arbitrário) e a outra o valor
      - AUTO: a escolha da estratégia para geração do chave é feita automaticamente
    - Ex: @GeneratedValue(strategy = GenerationType.AUTO)

Mapeamento Objeto Relacional – ORM

- Atributos
  - @Basic
    - É o tipo de mapeamento mais simples, definindo (através de seus parâmetros):
    - optional: indica se o atributo é obrigatório ou não (se pode ser null na base de dados)
    - fetch: indica se o atributo deve ser carregado sob-demanda (LAZY) ou se carrega-o imediatamente (EAGER)
  - Exemplo:

```
@Basic(optional=true, fetch=FetchType.LAZY)
private String descricao;
```

Mapeamento Objeto Relacional – ORM

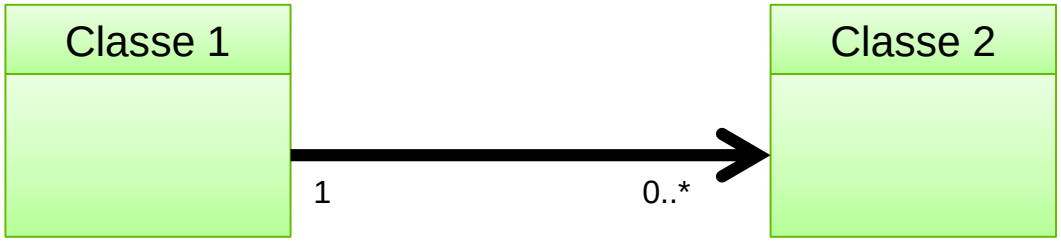
- Atributos
  - @Column
    - Define grande parte das propriedades comuns à colunas em banco de dados
      - Ex: name, length, unique etc.
  - Algumas outras anotações:
    - @Temporal
      - Usada para definir datas (Date, Time e Timestamp)
    - @Transient
      - Permite que um atributo não seja persistido
  - Lembre-se ainda: anotações de mapeamento de atributo também podem ser usadas no método ‘get’

Mapeamento Objeto Relacional – ORM

- Associações

Unidirecional

Bidirecional



Com Cardinalidade

Mapeamento Objeto Relacional – ORM

- Associações

Cardinalidade	Direcionamento
um para um	Unidirecional
um para um	Bidirecional
um para muitos	Unidirecional
um para muitos/muitos para um	Bidirecional
muitos para um	Unidirecional
muitos para muitos	Unidirecional
muitos para muitos	Bidirecional

- As seguintes notações mapeiam estes relacionamentos
  - @OneToOne, @OneToMany, @ManyToOne, e @ManyToMany

Mapeamento Objeto Relacional – ORM

- Associações
  - No paradigma OO, o direcionamento define qual classe "enxerga" qual
    - Em resumo, significa qual tem um atributo de referência para qual
      - Em um relacionamento bidirecional ambas se referem
  - No modelo entidade-relacionamento, existe uma decisão a mais para um relacionamento bidirecional:
    - Quem (qual tabela) fica com a informação de relacionamento (chave estrangeira)?

Mapeamento Objeto Relacional – ORM

- Associações (um para um)
  - @OneToOne
    - Indica um relacionamento um para um
      - Só é necessária no relacionamento bidirecional para indicar quem mapeia informações de relacionamento
    - Pode ser usado opcionalmente para definir propriedades de cascade, dentre outras
  - @JoinColumn
    - Elemento opcional que permite definir informações sobre a chave estrangeira (e.g., nome da coluna da chave estrangeira)

Mapeamento Objeto Relacional – ORM

- Associações (um para um)

```
15 public class Livro {
16
17     @OneToOne
18     @JoinColumn(name="fk_capa")
19     private Capa capa;
20
21     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

```
 7 public class Capa {
 8
 9     @OneToOne(mappedBy="capa")
10     private Livro livro;
11
12     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
IMAGEM	BLOB	Yes
TIPO	CHAR(1)	Yes

Mapeamento Objeto Relacional – ORM

- Associações (um para muitos)
  - O relacionamento um para muitos unidirecional também pode ser mapeado “por convenção”
    - Basta, para isto, que o tipo da lista seja uma @Entity
    - Opcionalmente pode-se usar @JoinColumn para definições adicionais

```
17 public class Livro {
18
19     @JoinColumn(name="id_livro")
20     private List<Rotulo> rotulos;
21
22     //...
```

```
 6 @Entity
 7 public class Rotulo {
 8
 9     @Id @GeneratedValue
10     private Long id;
11     private String descricao;
12
13     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
DESCRICAO	VARCHAR2(255)	Yes
ID_LIVRO	NUMBER(19,0)	Yes

Mapeamento Objeto Relacional – ORM

- Associações
  - Relacionamentos **um para muitos** bidirecionais são análogos ao um para um, fazendo o uso de @OneToMany (no lugar de @OneToOne)
  - No relacionamento **muitos para muitos** bidirecional, como em todo relacionamento bidirecional, deve-se definir quem é o “dono” do relacionamento
    - Para isto, usa-se o atributo mappedBy
    - A tabela de mapeamento pode ser configurada com @JoinTable
      - Esta anotação também pode ser usada no @OneToMany quando existir uma tabela de relacionamento

Mapeamento Objeto Relacional – ORM

- Associações
  - Note que @JoinTable fica na entidade “dona” do relacionamento, ou seja, a que não possui um atributo mapeada por outra

```
17 public class Livro {
18
19     @ManyToMany
20     @JoinTable(name = "associacao_livro_rotulo",
21                joinColumns = @JoinColumn(name = "fk_livro"),
22                inverseJoinColumns = @JoinColumn(name = "fk_rotulo"))
23     private List<Rotulo> rotulos;
24
25     //...
```

```
 9 @Entity
10 public class Rotulo {
11
12     @ManyToMany(mappedBy="rotulos")
13     private List<Livro> livrosEmQueAparece;
14
15     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes

Column Name	Data Type	Nullable
FK_LIVRO	NUMBER(19,0)	No
FK_ROTULO	NUMBER(19,0)	No

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
DESCRICAO	VARCHAR2(255)	Yes

Mapeamento Objeto Relacional – ORM

- Relacionamentos (herança)
  - @Inheritance
    - Define a estratégia de mapeamento de herança
  - @DiscriminatorColumn
    - Define o nome da coluna que identifica o tipo ao qual um determinado registro pertence
  - @DiscriminatorValue
    - Define o valor para o tipo da entidade na qual a anotação é utilizada

Mapeamento Objeto Relacional – ORM

- Relacionamentos (herança)
  - Existem três estratégias para mapeamento de herança – como sempre existe uma adotada por convenção:
    - Uma única tabela por hierarquia (SINGLE_TABLE): a soma dos atributos é distribuída em uma tabela (estratégia padrão)
    - *Joined-subclass (JOINED)*: nesta abordagem, cada entidade da hierarquia, concreta ou abstrata, é mapeada em uma tabela diferente
    - Uma tabela por classe concreta (TABLE_PER_CLASS): esta estratégia mapeia cada entidade concreta para uma tabela separada

Mapeamento Objeto Relacional – ORM

- Relacionamentos (herança)

```
21 @Entity
22 @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
23 @DiscriminatorColumn (name="tipo_livro",
24                       discriminatorType = DiscriminatorType.STRING)
25 @DiscriminatorValue("NORMAL")
26 public class Livro {
```

```
5  @Entity
6  @DiscriminatorValue("AUDIO")
7  public class LivroAudio extends Livro {
8
9      private byte[] mp3;
10
11     //...
```

Column Name	Data Type	Nullable
ID	NUMBER(19,0)	No
TIPO_LIVRO	VARCHAR2(31)	Yes
PRECO	NUMBER(19,4)	Yes
TITULO	VARCHAR2(255)	No
NUMPAGINAS	NUMBER(10,0)	Yes
DESCRICAO	VARCHAR2(2000)	Yes
FK_CAPA	NUMBER(19,0)	Yes
MP3	BLOB	Yes

Edit	Id	Tipo Livro	Preco	Titulo	Numpaginas	Descricao	Fk Capa	Mp3
	5	AUDIO	34,5	O Restaurante no Fim do Universo	335	Humor e ficção científica	1	[datatype]
	2	NORMAL	38,5	O Guia do Mochileiro das Galáxias	380	Humor e ficção científica	1	[datatype]

API - EntityManager

- O elemento central da manipulação e consulta da base de dados é feita pela API da classe **EntityManager**
  - Provê API para criação, remoção, busca e sincronização dos objetos com o banco de dados
  - Além disto, permite a execução de consultas JPQL
    - As consultas JPQL assemelham-se com a SQL, mas operam sobre objetos utilizando, por exemplo, a notação de ponto (.) dentro das consultas

API - EntityManager

- Uma classe Produto

```java
@Entity
public class Produto {

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

    @Column(name = "nome")
    private String descricao;

    private double preco;

    public Produto() {

    }

    //CRIAR GETTERS E SETTERS

    public Produto(int codigo, String descricao, double preco) {

        this.codigo = codigo;
        this.descricao = descricao;
        this.preco = preco;
    }

}
```

API - EntityManager

- Adicionando um registro

```java
package br.com.etecmam.conexaojpa;

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

public class CriaBanco {

    public static void main(String[] args) {

        //Criar Fábrica de Entity Manager infirmando nome da
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("LojaJPA");

        //Criar Entity Manager
        EntityManager em = emf.createEntityManager();

        //Iniciar Transação
        em.getTransaction().begin();

        //Criar Produto
        Produto p1 = new Produto(1,"SMART TV 50 POLEGADAS", 2000.0);

        //Salvar o produto criado no Banco
        em.persist(p1);

        //Finalizar Transação
        em.getTransaction().commit();

    }

}
```

API - EntityManager

- Pesquisando um registro

```
public class PesquisarProduto {

    public static void main(String[] args) {

        //Criar Fábrica de Entity Manager infirmando nome da
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("LojaJPA");

        //Criar Entity Manager
        EntityManager em = emf.createEntityManager();

        //Iniciar Transação
        em.getTransaction().begin();

        //REALIZA UMA PESQUISA  DE UM PRODUTO ESPECÍFICO AO BANCO
        Produto p = em.find(Produto.class, 1);

        System.out.println(p.getCodigo() + " " +
                           p.getDescricao() + " " +
                           p.getPreco());

    }


}
```

API - EntityManager

- Atualizando um registro

```
public class AtualizarProduto {

public static void main(String[] args) {

        //Criar Fábrica de Entity Manager infirmando nome da
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("LojaJPA");

        //Criar Entity Manager
        EntityManager em = emf.createEntityManager();

        //Iniciar Transação
        em.getTransaction().begin();

        Produto p1 = em.find(Produto.class, 1);
        p1.setPreco(2500);
        em.merge(p1);

        //Finalizar Transação
        em.getTransaction().commit();

    }

}
```

API - EntityManager

- Deletando um registro

```
public class AtualizarProduto {


    public static void main(String[] args) {

        //Criar Fábrica de Entity Manager infirmando nome da
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("LojaJPA");

        //Criar Entity Manager
        EntityManager em = emf.createEntityManager();

        //Iniciar Transação
        em.getTransaction().begin();

        Produto p1 = em.find(Produto.class, 1);

        em.remove(p1);

        //Finalizar Transação
        em.getTransaction().commit();


    }


}
```

Operações CRUD sobre entidades

- Verificar a atividade de casa (terminar a atividade da última aula)

Linguagem de consulta – JPQL

- JPA oferece uma linguagem própria para consulta de entidades
  - Portabilidade entre bancos
- Consultas são representadas e executadas por um objeto Query
- Consultas em JPQL podem ser definidas em qualquer classe Java, dentro de um método por exemplo

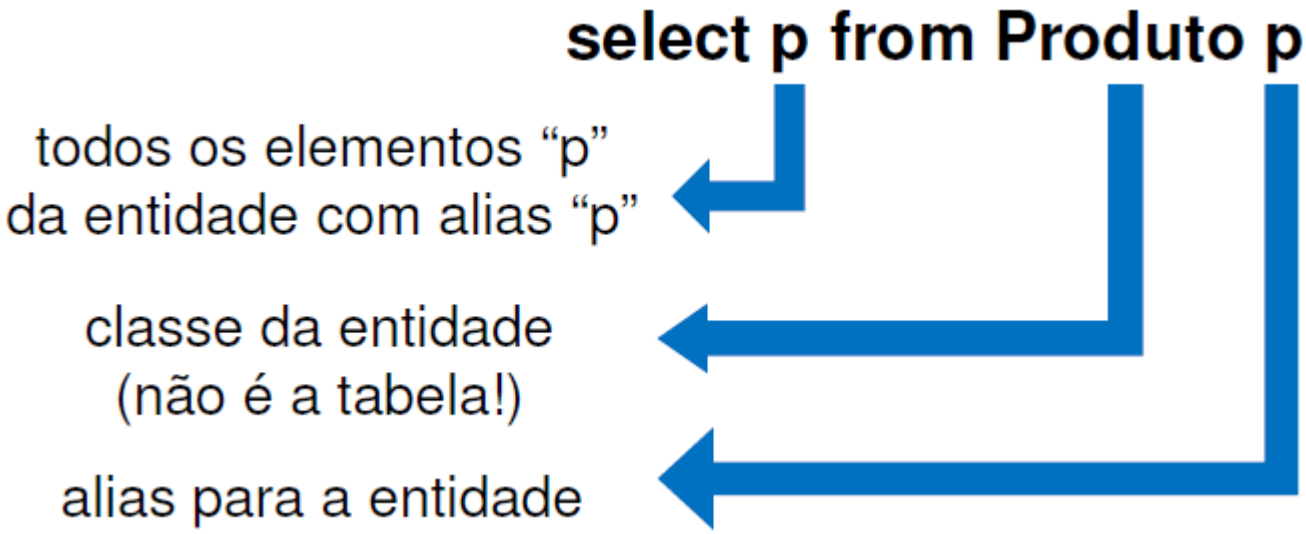
Linguagem de consulta – JPQL

- 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
- Exemplo de consulta para buscar todas as entidades no banco:

```
Query consulta1 = em.createQuery("select p from Produto p");
List<Produto> produtos = consulta1.getResultList();
for(Produto prod : produtos){
    System.out.println("Nome: "+prod.getNome());
}
```

Linguagem de consulta – JPQL

- Como escrever as consultas



Linguagem de consulta – JPQL

- Como escrever as consultas
  - Buscar uma única entidade passando dados via parâmetros

```
Query consulta3 = em.createQuery("select p from Produto p where p.id = :id");
consulta3.setParameter("id", 1);
Produto prod = (Produto)consulta3.getSingleResult();
System.out.println("Nome: " + prod.getNome());
```

  - Operador like

```
Query consulta4 =
            em.createQuery("select p from Produto p where p.nome like :nome");
consulta4.setParameter("nome", "Arroz%");
List<Produto> produtos = consulta4.getResultList();
```

Linguagem de consulta – JPQL

- Como escrever as consultas
  - consulta.setMaxResults(2);
    - Funciona como o LIMIT do Banco de dados;
- Pode navegar entre **os objetos**

```
Query consulta =  em.createQuery("Select
endereco.cliente.nome from Endereco as endereco where
endereco. cliente.id = :id");

consulta.setParameter("id", 1)
```

Linguagem de consulta – JPQL

- Como escrever as consultas
  - Count()

```
Query consulta5 = em.createQuery("select count(p) from Produto p");
Long quantidade = (Long)consulta5.getSingleResult();
System.out.println("Quantidade: " + quantidade);
```

```
Query consulta6 =
        em.createQuery("select count(distinct p.nome) from Produto p");
Long quantidade = (Long)consulta6.getSingleResult();
System.out.println("Quantidade de Nomes Distintos: " + quantidade);
```

```
Query consulta7 =
    em.createQuery("select p.nome, count(p) from Produto p group by p.nome");
List<Object[]> linhas = consulta7.getResultList();
for (Object[] linha : linhas) { // cada elemento é um vetor, representa a linha
   String nome = (String)linha[0];
   Long count = (Long)linha[1];
   System.out.println("nome: "+nome+" Quantidade: "+count);
}
```

Linguagem de consulta – JPQL

- Como escrever as consultas
  - Subconsultas

```
Query consulta =
        em.createQuery("select p1 from Produto p1
                                    where  p1.precoUnitario >=
                                ( select avg(p2.precoUnitario)
                                  from Produto p2 )“
                        );
List<Produto> produtos = consulta.getResultList();
for(Produto prod : produtos){
        System.out.println("Nome: "+prod.getNome());
}
```

Linguagem de consulta – JPQL

- Como escrever as consultas
  - Com verificação de intervalo

```
Query consulta = em.createQuery("select p
                                   from Produto p
                                  where p.precoUnitario
                                        between :valorMin and :valorMax");
consulta.setParameter("valorMin", 1.50f);   // " f " força conversão para float
consulta.setParameter("valorMax", 2.50f);
List<Produto> produtos = consulta.getResultList();
for (Produto prod : produtos) {
    System.out.println("Nome: " + prod.getNome());
}
```

Linguagem de consulta – JPQL

- Funções que podem ser utilizadas
  - 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

Linguagem de consulta – JPQL

- Named Queries
  - 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
  - A anotação @NamedQueries é utilizada quando queremos definir várias consultas

Linguagem de consulta – JPQL

- Named Queries
  - 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

Linguagem de consulta – JPQL

- Nas entidades pode-se utilizar

```
@NamedQueries(
    @NamedQuery(name= “Lista filmes”, query = “Select filme from Filme as filme”),
    @NamedQuery(name= “Lista filmes tempo”, query = “Select filme from Filme as filme where filme.duracao > :tempo”)
    )
```

```
Query consulta = em.createNamedQuery(“Lista filmes”);
List<Filmes> resultado = consulta.getResultList();
```

  - Somente isso e pode ser utilizada normalmente

Linguagem de consulta – JPQL

- Nas entidades pode-se utilizar

```
@NamedQueries(
    @NamedQuery(name= “Lista filmes”, query = “Select filme from Filme as filme”),
    @NamedQuery(name= “Lista filmes tempo”, query = “Select filme from Filme as filme where filme.duracao > :tempo”)
    )
```

```
Query consulta = em.createNamedQuery(“Lista filmes tempo”);
consulta.setParameter(“tempo”, 90);
```

  - Somente isso e pode ser utilizada normalmente

Linguagem de consulta – JPQL

- 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

SELECT f.nome , f. departamento . nome FROM Funcionario f;

Linguagem de consulta – JPQL

- Resultados especiais

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 ); 3

List < Object []> lista = query . getResultList ();

for ( Object [] tupla : lista ) {

     System.out.println ("Funcionário : " + tupla [0]);

    System.out.println ("Departamento : " + tupla [1]);

}
```

Linguagem de consulta – JPQL

- Paginação
  - Quando existe uma grande quantidade de dados, buscar todos os livros sem nenhum filtro vai **sobrecarregar o tráfego da rede e a memória** utilizada
  - A paginação do resultado de uma consulta é realizada através dos métodos setFirstResult() e setMaxResults()

```
Query query = manager.createQuery ("select livro from Livro livro ");

query.setFirstResult (10) ;

query.setMaxResults (20) ;

List livros = query.getResultList();
```

Resumo

- A Java Persistence API é utilizada para mapear banco de dados relacionais e facilitar o desenvolvimento de código o tornando mais ágil
- Java Persistence API e seus principais componentes
  - Mapeamento Objeto Relacional (ORM)
  - API (EntityManager)
  - Uma linguagem de consulta (JPQL)
- Principais anotações ORM da JPA
- Operações CRUD sobre entidades
- Consultas formadas por diferentes elementos

Resumo

- A Java Persistence API é utilizada para mapear banco de dados relacionais e facilitar o desenvolvimento de código o tornando mais ágil
- Java Persistence API e seus principais componentes
  - Mapeamento Objeto Relacional (ORM)
  - API (EntityManager)
  - Uma linguagem de consulta (JPQL)
- Principais anotações ORM da JPA
- Operações CRUD sobre entidades
- Consultas formadas por diferentes elementos

Resumo

Anotação	Descrição
@Entity	Indica que objetos dessa classe se tornem “persistível” no banco de dados
@Table	Pode-se definir propriedades da tabela, como seu nome
@SecondaryTable	Permite que os atributos de uma Entidade (classe) possa ser distribuído entre mais de uma tabela
@Id	Define uma chave primária
@GeneratedValue	Define que a coluna tem um valor gerado automaticamente
@Column	Define grande parte das propriedades comuns à colunas em BD
@Transisent	Informa que o atributo não será persistido
@OneToOne	Define um relacionamento um-para-um
@ManyToOne	Define um relacionamento muitos-para-um
@OneToMany	Define um relacionamento um-para-muitos
@ManyToMany	Define um relacionamento muitos-para-muitos
@Inheritance	Permite utilizar herança

Bibliografia

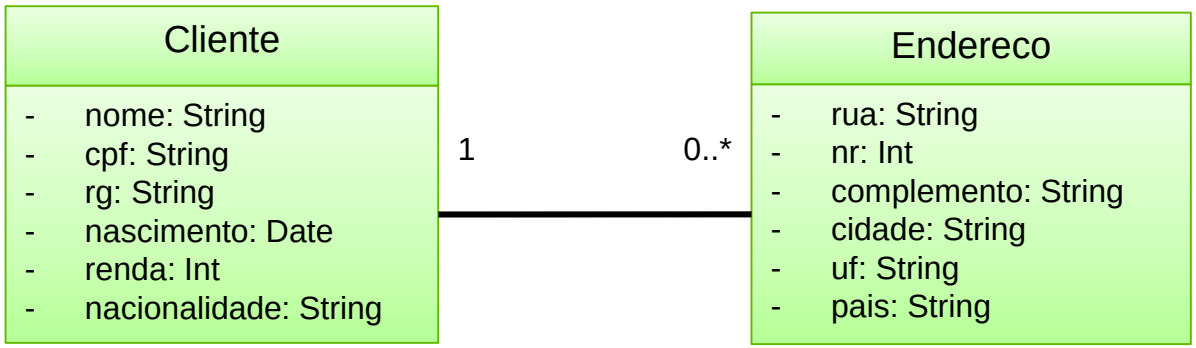
- EDSON GONÇALVES**. DESENVOLVENDO APLICAÇOES WEB COM JSP, SERVLETS: JAVASERVER FACES, HIBERNATE, EJB 3 PERSISTENCE**. Editora CIENCIA MODERNA.
- HARVEY M. DEITEL, PAUL DEITEL DEITEL. **Java: como programar**. Edição 8. Editora Pearson Prentice Hall, 2010.
- GILLIARD CORDEIRO. **APLICAÇOES JAVA PARA A WEB COM JSF E JPA**. Editora CASA DO CODIGO
- Mike Keith, Merrick Schincariol. **Pro JPA 2: Mastering the Java™ Persistence API**. Editora Apress, 2009.

Perguntas?


INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SUL-RIO-GRANDENSE

Atividade Prática

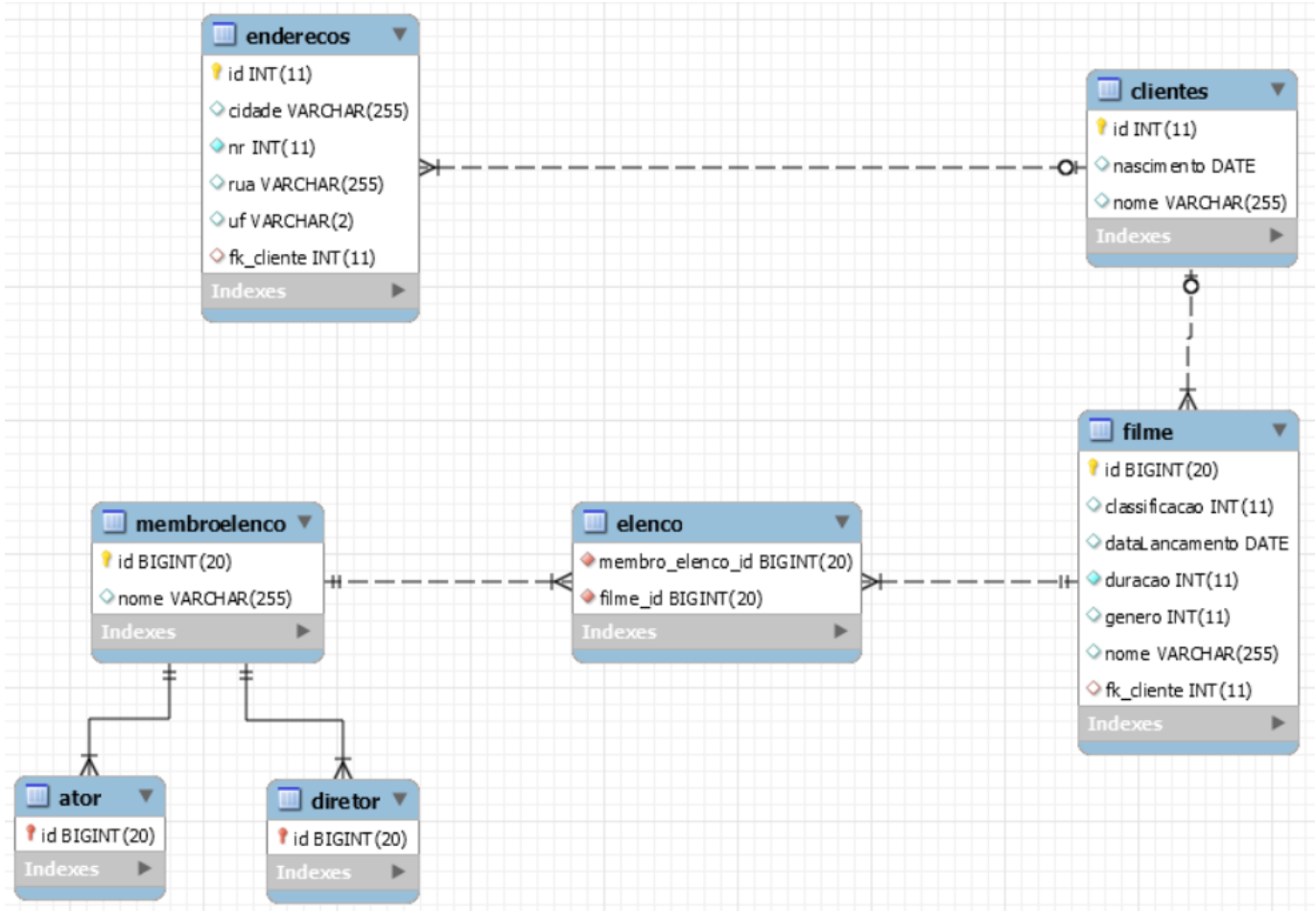
- Defina Entidades JPA para o seguinte modelo:



Trabalho sobre JPQL

- Locadora do Butuca
  - Trabalha com a locação de filmes
  - Cadastro de clientes
  - Cadastro de filmes e seus elencos
- Terminar o projeto preliminar disponível em ww.ricardoluis.com
- Trabalho **em duplas**
- Data para apresentação 08/09/2015

Trabalho sobre JPQL



Trabalho sobre JPQL

- Criar um menu para executar as diversas consultas necessárias (20%)
- Finalizar o gerador de dados (30%)
  - Trabalhar com inserção como os exemplos demonstrados
  - 10 Clientes (cada cliente deve ter 2 endereços)
  - 20 Filmes (1 diretor e pelo menos 2 atores por filme)
  - Popular a locação somente para alguns filmes
    - Nem todos os clientes devem ter filmes locados
    - Nem todos os filmes devem estar locados

Trabalho sobre JPQL

- Consultas que devem ser implementadas (50%)
  - 01 - Quantos filmes existem ao todo?
  - 02 - Quais filmes em que há algum ator que contenha no nome uma String informada pelo usuário?
  - 03 - Quais os filmes dirigidos por tal diretor em que um ator fez parte do elenco?
  - Query 04 - Quais os filmes em que um ator A atuou com outro B? Ambos informados pelo usuário
  - 05 - Listar os filmes de um ator por gênero em ordem ascendente pela data de lançamento
  - 06 - Listar os filmes de um diretor por gênero em ordem ascendente pela data de lançamento
  - 07 - Listar os filmes de um diretor por faixa etária (classificação) em ordem descendente pela data de lançamento

Trabalho sobre JPQL

- Consultas que devem ser implementadas (50%)
  - 08 - Listar os filmes de um diretor por gênero e por faixa etária em ordem ascendente pela data de lançamento
  - 09 – Mostrar a média de duração total dos filmes
  - 10 - Mostrar a média de duração dos filmes por gênero
  - 11 - Listar a quantidade de filmes por gênero
  - 12 – Retornar os dados de um filme pelo seu nome (LIKE “%nome%”)
  - 13 – Retornar os dados de um cliente pelo seu nome (LIKE “%nome%”)
  - 14 – Exibir o cadastro de todos os filmes cadastrados
  - 15 – Exibir o cadastro de todos os clientes cadastrados