Universidade Presbiteriana Mackenzie

Linguagem de Programação III

JPA e Hibernate
Mapeamento de Associações e Consultas

Prof. Ms. Leandro Pupo Natale

Faculdade de Computação e Informática

Tópicos da Aula

- Modos de acesso dos provedores JPA
- Mapeamento de Hierarquia de Classes
- Mapeamento de Associações
- Relacionamentos Bidirecionais
- Anotação @Embeddable
- Fetch – tipo básico e associação
- Cascade
- JPQL
- Criteria

Provedores JPA

- **Provedores de JPA:**
  - Precisam ter acesso ao estado das entidades para poder administrá-las.
  - Exemplos:
    - Quando **persiste-se uma instância de uma entidade**, o provedor deve "pegar" os dados desse objeto e armazená-los no banco.
    - Quando **busca-se uma instância de uma entidade**, o provedor recupera as informações correspondentes do BD e "guarda" em um objeto.

Provedores JPA

- O JPA 2 define dois **modos de acesso** ao estado das instâncias das entidades:
- **Field Access:**
  - anotações de mapeamento nos **atributos**
  - os atributos dos objetos são acessados diretamente através de **reflection** e não é necessário implementar métodos getters e setters.
  - se os métodos getters e setters estiverem implementados, eles não serão utilizados pelo provedor JPA.

Provedores JPA

- O JPA 2 define dois modos de acesso ao estado das instâncias das entidades:
- **Property Access:**
  - anotações de mapeamento nos **métodos getters**
  - 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.

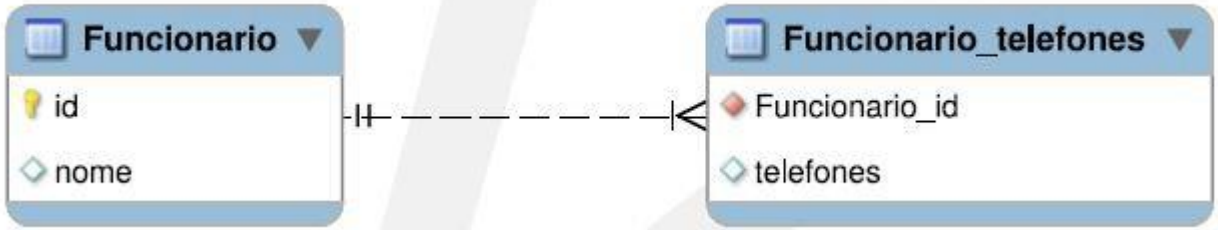
Anotação @ElementColletion

- Considere um sistema que controla o cadastro dos funcionários de uma empresa.
- Se os telefones de contato dos funcionários, considerando que cada funcionário possa ter um ou mais telefones, em Java pode-se utilizar coleções para esse atributo:

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

    private String nome;

    @ElementCollection
    private Collection<String> telefones;
}
```

Anotação @ElementColletion

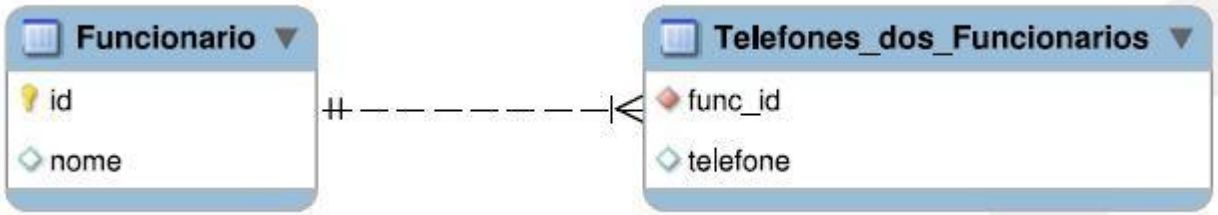
- A anotação @ElementCollection deve ser utilizada para que o mapeamento seja realizado.
- Nesse exemplo, o BD possuiria uma tabela chamada Funcionario_telefones contendo duas colunas.
  - Uma coluna seria usada para armazenar os identificadores dos funcionários e a outra para os telefones.



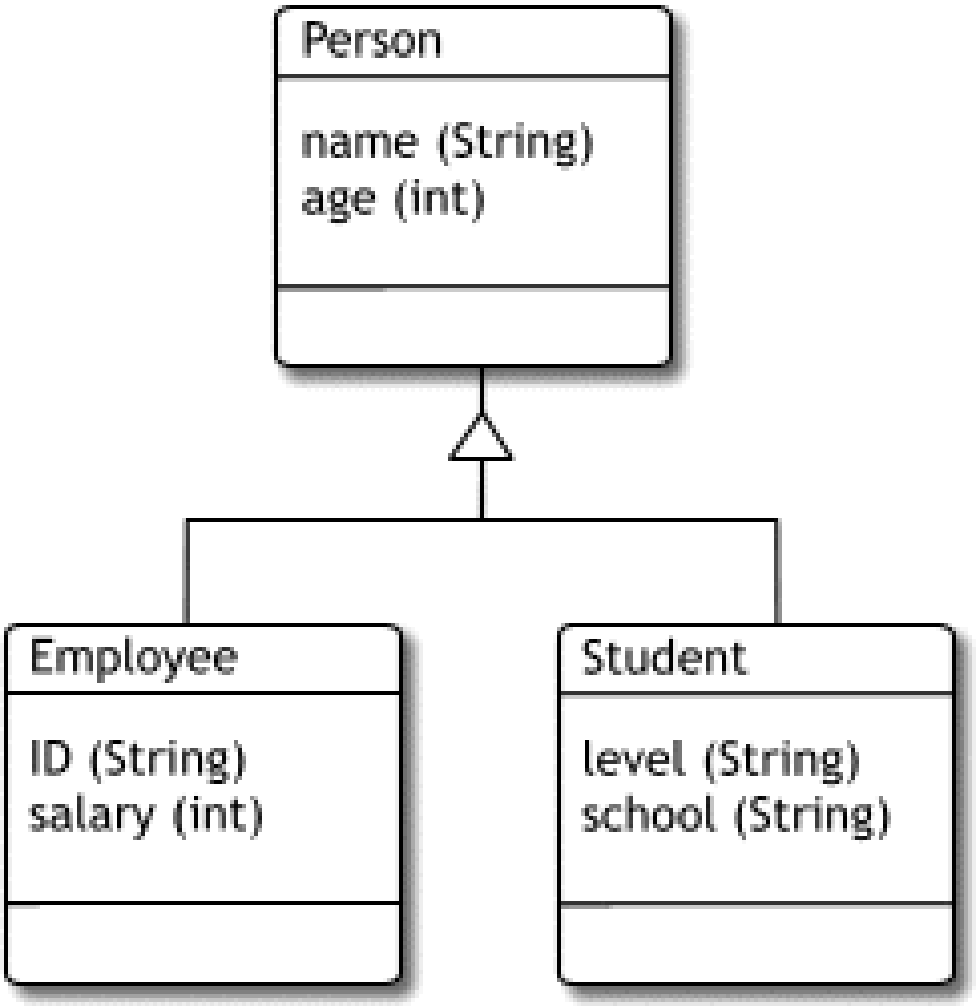
Anotação @ElementColletion

- A tabela criada pode ter um nome personalizado, por meio das anotações @CollectionTable e @Column

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

    private String nome;

    @ElementCollection
    @CollectionTable( name="Telefones_dos_Fu
    ncionarios",
    joinColumns=@JoinColumn(name="func_id"))
    @Column(name="telefone")
    private Collection<String> telefones;
}
```



Mapeamento de Hierarquia de Classes



Mapeamento de Hierarquia de Classes

- A especificação JPA define 3 estratégias de mapeamento de polimorfismo, ou simplesmente mapeamento de herança:

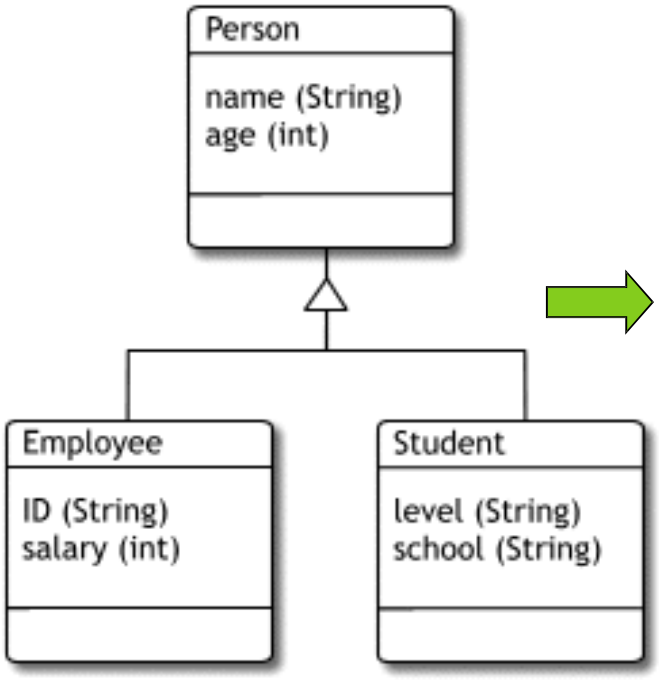
  - SINGLE TABLE: Uma tabela para toda a hierarquia de classes
  - TABLE PER CLASS : Uma tabela por classe concreta
  - JOINED: Uma tabela por subclasse

Estratégia 1 – Single Table

- **Uma Tabela para toda a Hierarquia de Classes**
  - Toda a hierarquia de classes é representada por uma única tabela no BD com o nome da **superclasse** e deve ser anotada com:
  - **@Inheritance(strategy=InheritanceType.SINGLE_TABLE).**
  - Todos os atributos da superclasse e os das subclasses são mapeados para colunas dessa tabela.
  - Tem-se uma coluna especial chamada **DTYPE** que identifica a classe do objeto correspondente ao registro.

Estratégia 1 – Single Table

- **Uma Tabela para toda a Hierarquia de classe**




```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Person { …}
```

```
@Entity
public class Employee extends Person { … }
```

```
@Entity
public class Student extends Person { … }
```

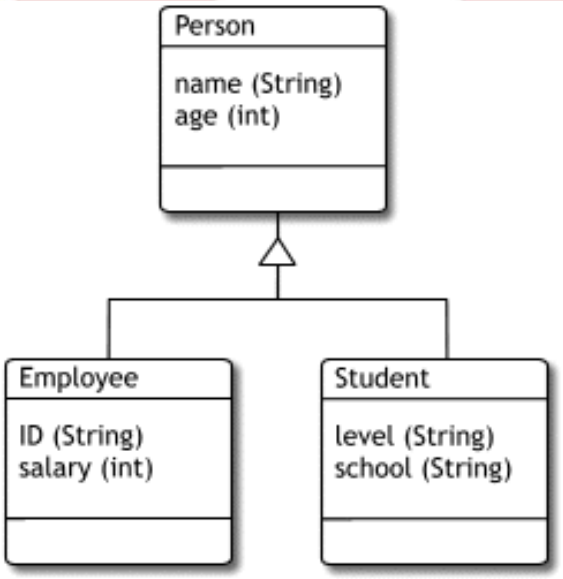
Estratégia 1 – Single Table

- **Uma Tabela para toda a Hierarquia de classe**
- Vantagem:
  - A estratégia Single Table é a mais comum e a que possibilita melhor desempenho em relação a velocidade das consultas.
- Desvantagem:
  - A desvantagem da Single Table é o **consumo desnecessário de espaço**, já que **nem todos os campos são utilizados** para todos os registros.

```
public class AdicionaPessoa {
    EntityManagerFactory factory =
        Persistence.createEntityManagerFactory ("Mapeamento_pu");
    EntityManager em = factory.createEntityManager ();

    em.getTransaction().begin();

    Person p1 = new Person();
    p1.setName("Jose");  p1.setAge(30);

    Employee p2 = new Employee();
    p2.setName("Roberto"); p2.setAge(27);
    p2.setsalary(3000);

    Student p3 = new Student ();
    p3.setName("Paulo"); p2.setAge(22);
    p3.setLevel("Bsch");
    p3.setSchool("Mack");

    em.persist(p1);
    em.persist(p2);
    em.persist(p3);
    em.getTransaction().commit();
    em.close();
    factory.close();
    }
}
```

Estratégia 2 – Table Per Class

- **Uma Tabela por classe Concreta**
  - Cada classe **concreta** será armazenada em uma tabela distinta
  - Os dados não são colocados em tabelas diferentes
  - Para remontar um objeto **não é necessário** realizar operações de **join.**



EMPLOYEE

NAME (varch.)	AGE (integer)	ID (varchar)	SAL (integer)
Joe	34	#31	80000

PK

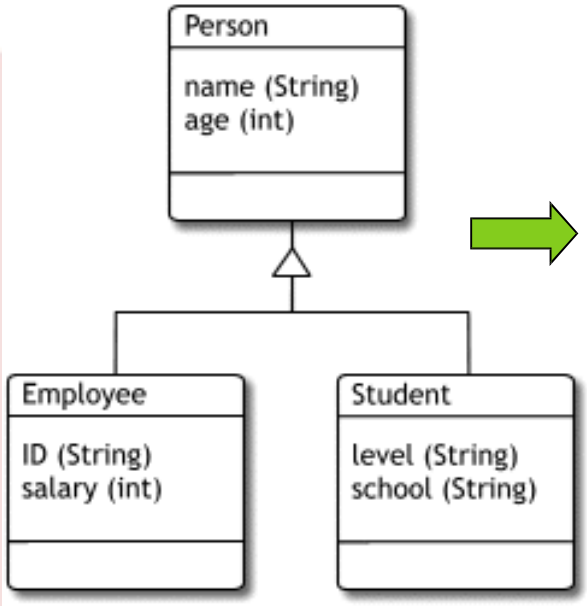
STUDENT

NAME (varch.)	AGE (integer)	LVL (varch.)	SCH (varch.)
Jack	22	MSc	Berkeley

PK

Estratégia 2 – Table Per Class

- **Uma Tabela por classe Concreta**
  - Se houver uma classe abstrata não será mapeada
  - As subclasses são mapeadas como uma classe qualquer




```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Person { … }
```

```
@Entity
public class Employee extends Person { … }
```

```
@Entity
public class Student extends Person { … }
```

Estratégia 2 – Table per Class

- **Uma Tabela por classe Concreta**
- Desvantagem:
  - Não suporta muito bem associações polimórficas.
  - Não existe um vínculo explícito no BD entres as tabelas correspondentes às classes da hierarquia
  - Não podemos utilizar a geração automática de chave primárias simples e numéricas.

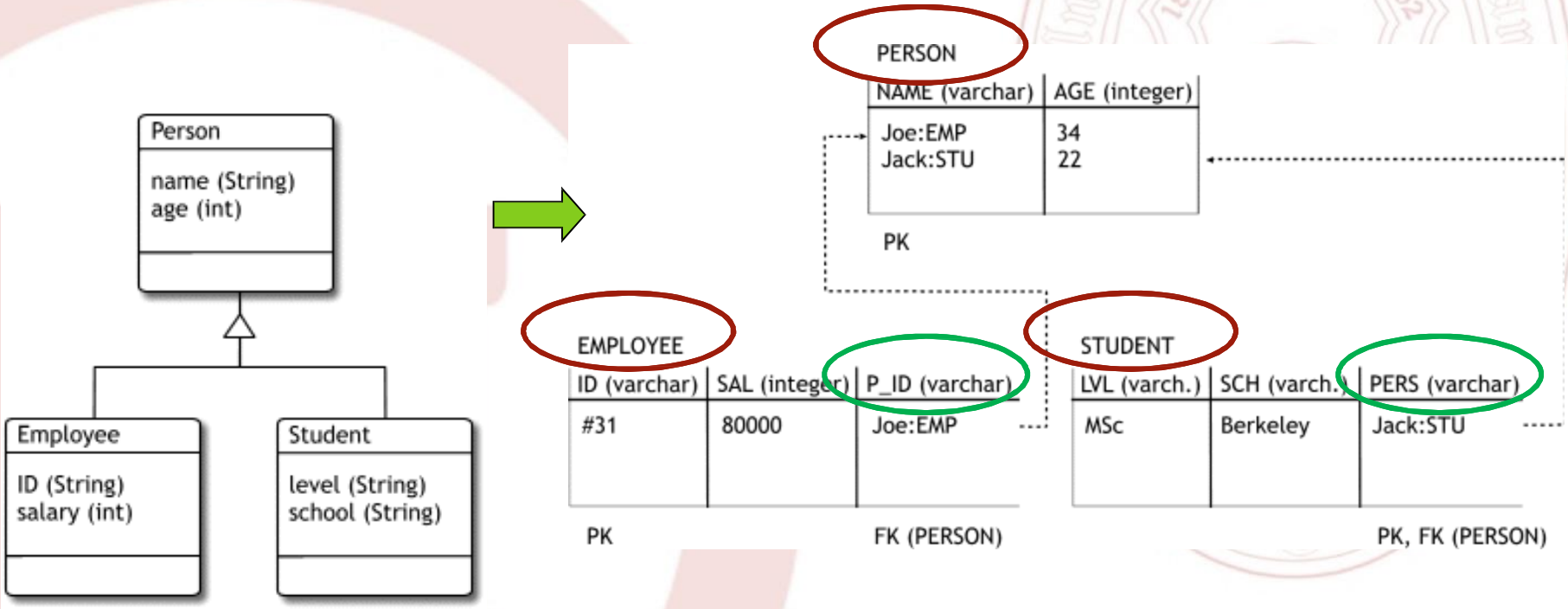
```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Person {
    @Id
    private Long id;
    private String nome;
}
```

Estratégia 3 - Joined

- **Uma Tabela por Classe**
- 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 **subclasses** possuem **chaves estrangeiras** vinculadas à tabela associada à **superclasse.**

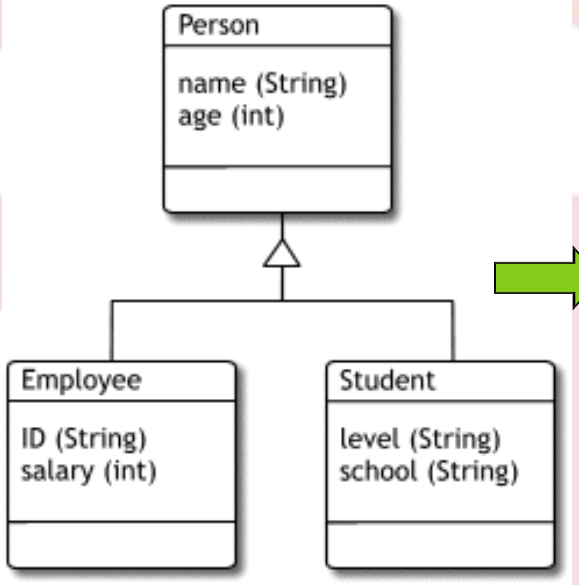
Estratégia 3 - Joined

- **Uma Tabela por Classe**



Estratégia 3 - Joined

- **Uma tabela por classe**




```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Person { …}
```

```
@Entity
public class Employee extends Person { … }
```

```
@Entity
@PrimaryKeyJoinColumn(name="Person_ID")
public class Student extends Person { … }
```

Estratégia 3 - Joined

- **Uma Tabela por Classe**
  - Cada classe será armazenada em uma tabela, inclusive a classe pai, sendo ela abstrata ou não.
  - Necessita de um único *join* para recuperar os dados.
  - Pode impactar no desempenho de uma aplicação devido ao número de *joins* que podem ser feitos em uma hierarquia completa.

Estratégia 3 - Joined

- **Uma Tabela por Classe**
- Vantagem:
  - O consumo de espaço utilizando a estratégia Joined é menor do que o utilizado pela estratégia Single Table.
- Desvantagem:
  - As **consultas** são **mais lentas** do que em Single Table
  - realizar **operações de join** para recuperar os dados dos objetos.

Mapeamento de Associações

- As **associações** entre as **entidades** de um domínio devem ser expressos na modelagem através de **vínculos** entre classes.
- De acordo com a JPA, pode-se definir 4 tipos de associações de acordo com a multiplicidade.
- Tipos de associações:
  - One-to-One (Um para Um) 1 – 1
  - One-to-Many (Um para Muitos) 1 – N (composição)
  - May-to-One (Muitos para Um) N – 1
  - May-to-Many (Muitos para Muitos) N - N

Mapeamento One-to-One

- Deve-se criar uma classe para cada entidade e aplicar nelas as anotações básicas de mapeamento.
- Como existe um relacionamento entre as entidades, deve-se expressar esse **vínculo** através de um atributo que pode ser inserido na classe.
- Deve-se informar ao provedor JPA que o relacionamento que existe entre as entidades é do tipo One to One.
- Fazemos isso aplicando a anotação **@OneToOne no atributo** que expressa o relacionamento.

Mapeamento One-to-One

- Exemplo:

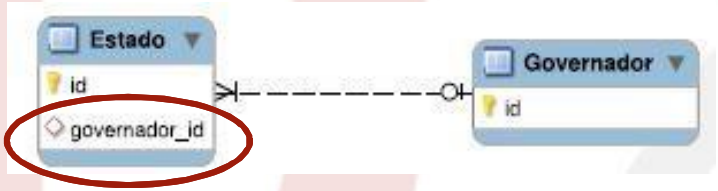

```
@Entity
public class Estado {
 @Id
 @GeneratedValue
 private Long id;

 @OneToOne (optional=false)
 private Governador governador;
}
```

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

}
```

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

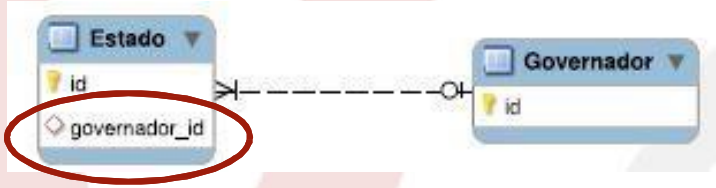
Mapeamento One-to-One

- No BD, 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.



Mapeamento One-to-One

- Por padrão, o nome da coluna de relacionamento é formado por:
  - **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.**



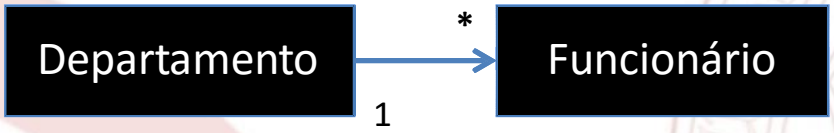
Mapeamento One-to-One

- Pode-se alterar, aplicando a anotação **@JoinColumn**

```
@Entity
public class Estado {
  ...
  @OneToOne (optional=false)
  @JoinColumn(name="gov_id")
  private Governador governador;
}
```

Mapeamento One-to-Many

- Exemplo:




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

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

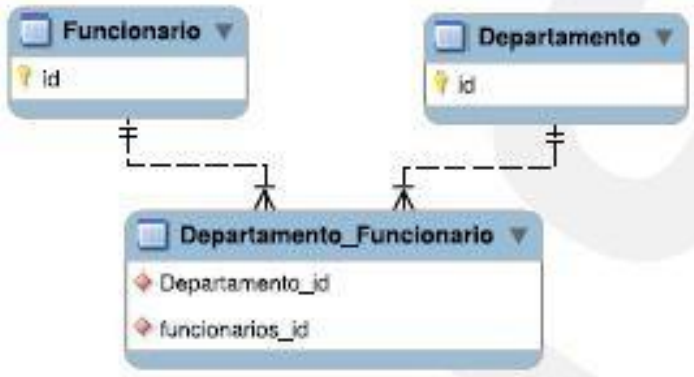
```
@Entity
public class Funcionario {
 @Id
 @GeneratedValue
 private Long id;

}
```

Mapeamento One-to-Many

- No BD, além das tabelas correspondentes às classes, deve existir uma terceira para relacionar os registros das classes de entidade, chamada de **join table.**
- Para personalizar os nomes das colunas na join table e da própria join table, aplica-se a anotação
- **@JoinTable** no atributo que define o relacionamento.
- Por padrão, o nome da join table é formada pela:
  - **– Concatenação com "_" dos nomes das duas entidades.**

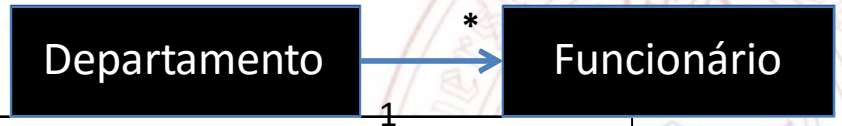
Mapeamento One-to-Many

- No exemplo de departamentos e funcionários, o nome do join table seria **Departamento_Funcionario.**
- Essa tabela possuirá 2 coulnas vinculadas às entidades que formam o relacionamento. No exemplo, a join table possuirá uma coluna chamada **Departamento_id** e outra chamada **funcionarios_id.**



Mapeamento One-to-Many

- Exemplo:




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

 @OneToMany
 @JoinTable(name="DEP_FUNC",
 joinColumns=@JoinColumn(name="DEP_ID"),
 inverseJoinColumns=@JoinColumn(name="FUNC_ID"))
 private Collection <Funcionario> funcionarios;
}
```

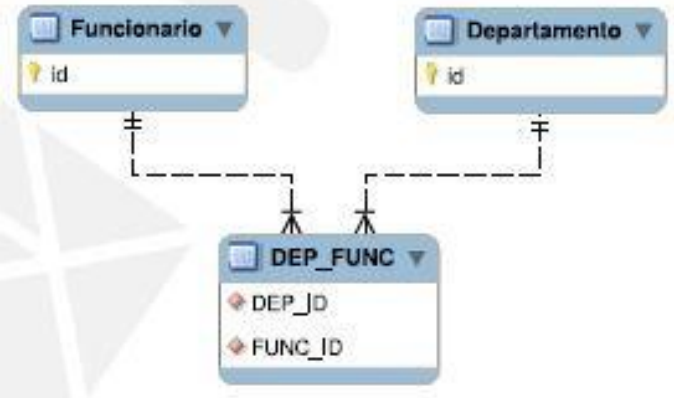
```
@Entity
public class Funcionario {
 @Id
 @GeneratedValue
 private Long id;

}
```

- **joinColumns:** Colunas que identificam a chave da entidade
- **inverseJoinColumns:** Colunas que identificam a chave da entidade de destino / target entity

Mapeamento One-to-Many

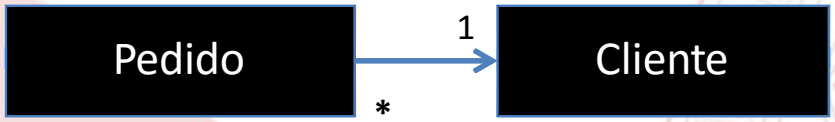
```
@Entity
public class Departamento {
 @Id
 @GeneratedValue
 private Long id;

 @OneToMany
 @JoinTable(name="DEP_FUNC",
 joinColumns=@JoinColumn(name="DEP_ID"),
 inverseJoinColumns=@JoinColumn(name="FUNC_ID"))
 private Collection <Funcionario> funcionarios;
}
```

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

}
```



Mapeamento Many-to-One

- Exemplo:

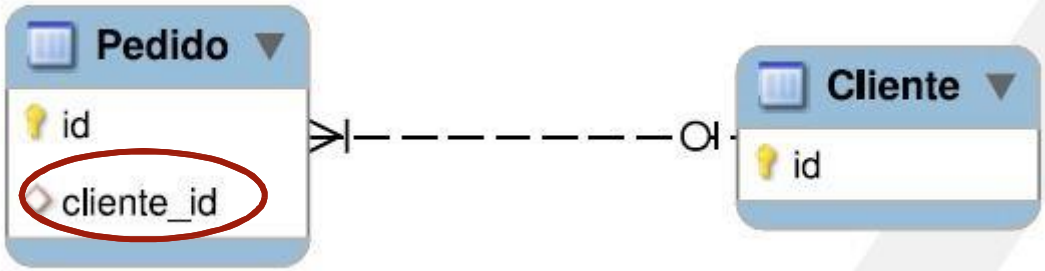



```
@Entity
public class Pedido {
 @Id
 @GeneratedValue
 private Long id;

 @ManyToOne
 private Cliente cliente;
}
```

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

}
```

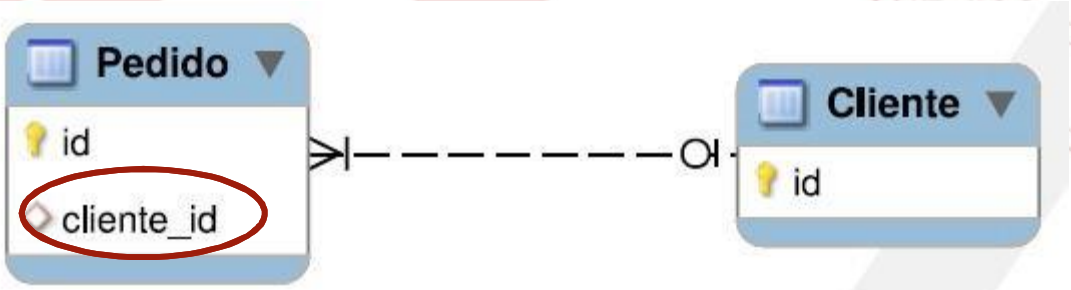
Mapeamento Many-to-One

- No BD, a tabela referente à classe Pedido possuirá uma coluna de relacionamento chamada de **join column** vinculada à tabela da classe Cliente.
- Em geral, **essa coluna será definida como uma chave estrangeira associada** à tabela referente à classe Cliente.



Mapeamento Many-to-One

- Por padrão, o nome da coluna de relacionamento é formado por:
  - **nome da entidade alvo do relacionamento, seguido pelo caractere "_" e pelo nome do atributo que define a chave primária da entidade alvo.**
- No exemplo de Pedido e Cliente, a join column teria o nome **cliente_id.**



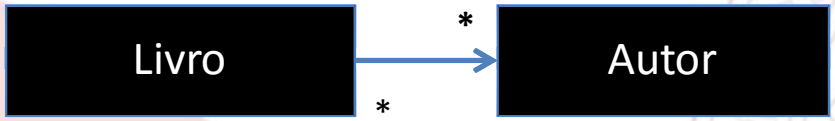
Mapeamento Many-to-One

- Pode-se alterar, aplicando a anotação **@JoinColumn**

```
@Entity
public class Pedido {
 ...
 @ManyToOne (optional=false)
 @JoinColumn(name="cli_id")
 private Cliente cliente;
}
```

Mapeamento Many-to-Many

- Exemplo:




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

 @ManyToMany
 private Collection <Autor> autores;
}
```

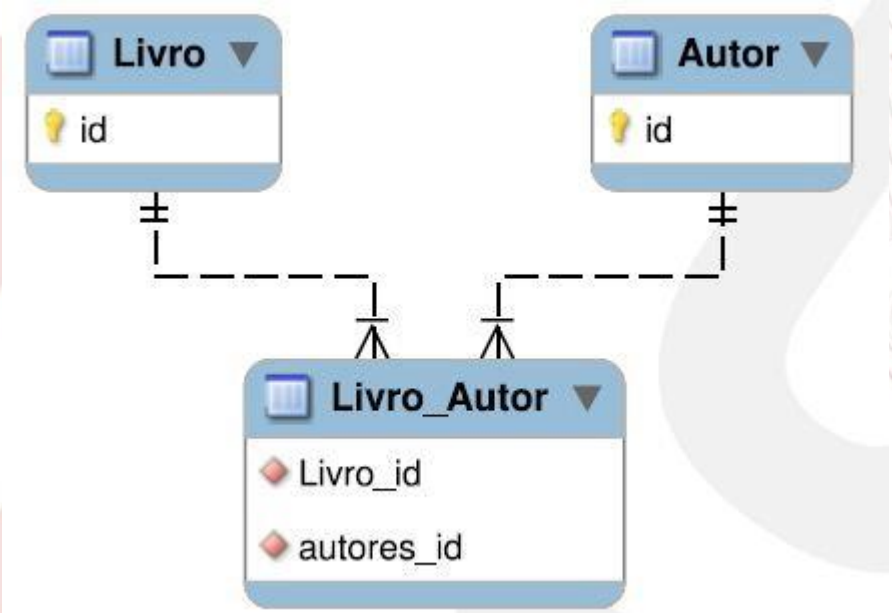
```
@Entity
public class Autor {
 @Id
 @GeneratedValue
 private Long id;

}
```

Mapeamento Many-to-Many

- No BD, além das tabelas correspondentes às classes de entidade, deve existir uma terceira para relacionar os registros dessas classes, chamada de **join table.**
- Essa join table é criada para relacionar os registros dos livros e os registros dos autores, por exemplo.
- Por padrão, o nome da **join table** é formada pela:
  - **– Concatenação com "_" dos nomes das duas entidades.**

Mapeamento Many-to-Many

- No exemplo de livros e autores, o nome do join table seria **Livro_Autor**, com 2 colunas vinculadas às entidades que formam o relacionamento, uma coluna chamada **Livro_id** e outra chamada **autores_id.**



Mapeamento Many-to-Many

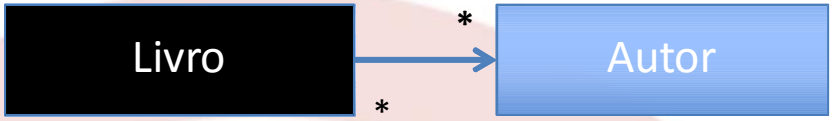
- Exemplo:


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

 @OneToMany
 @JoinTable(name="Liv_Aut",
 joinColumns=@JoinColumn(name="Liv_ID"),
 inverseJoinColumns=@JoinColumn(name="Aut_ID"))
 private Collection <Autor> autores;
}
```

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

}
```

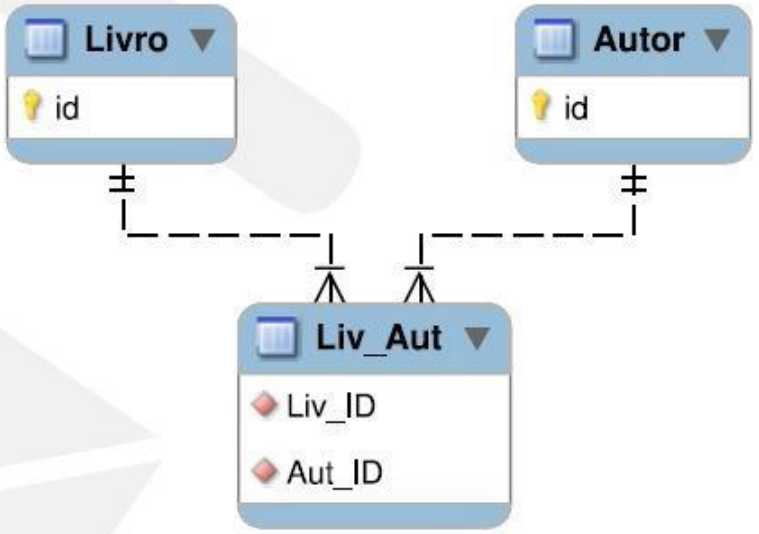
- **joinColumns:** Colunas que identificam a chave da entidade
- **inverseJoinColumns:** Colunas que identificam a chave da entidade de destino / target entity

Mapeamento Many-to-Many




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

}
```

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

 @OneToMany
 @JoinTable(name="Liv_Aut",
 joinColumns=@JoinColumn(name="Liv_ID"),
 inverseJoinColumns=
            @JoinColumn(name="Aut_ID"))
 private Collection <Autor> autores;
}
```



Relacionamentos Bidirecionais

- Ao se expressar uma associação coloca-se um atributo em uma das entidades, daí pode-se acessar a outra entidade a partir da primeira.
- Por exemplo, considere a associação entre governadores e estados.

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

 @OneToOne (optional=false)
 private Governador governador;

 //Getters e Setters
}
```

Relacionamentos Bidirecionais

- Como a associação está definida na classe Estado, pode-se acessar governador a partir de um estado.

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

- Também poderia-se expressar a associação da classe Governador e acessar um estado a partir de governador.

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

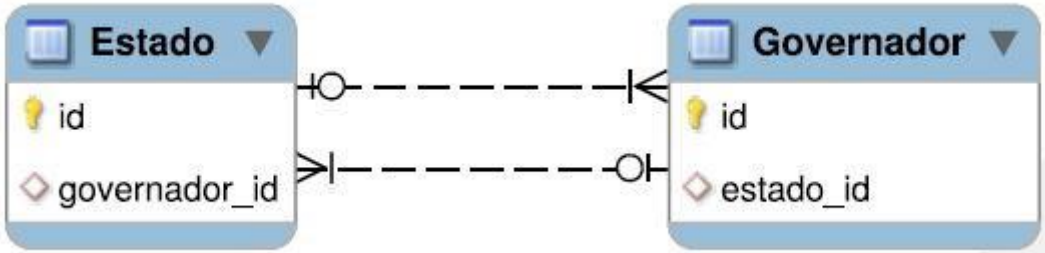
```
@Entity
public class Governador {
 @Id @GeneratedValue
 private Long id;
 @OneToOne (optional=false)
 private Estado estado;
 //Getters e Setters
}
```

Relacionamentos Bidirecionais

- Deve-se indicar em uma das classes que esse relacionamento bidirecional é a junção de dois relacionamentos unidirecionais.
- Para isso, adiciona-se 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.

Relacionamentos Bidirecionais

- No BD cria-se 2 colunas de relacionamentos.
- O provedor JPA considera 2 relacionamentos unidirecionais distintos entre as entidades.



- Porém, no modelo relacional, deve-se expressar apenas uma coluna de relacionamento entre estados e governadores. Ou seja, o relacionamento deve ser bidirecional.

Relacionamentos Bidirecionais

```
@Entity
public class Governador {
 @Id
 @GeneratedValue
 private Long id;
 @OneToOne (mappedBy="governador")
 private Estado estado;

 //Getters e Setters
}
```

Anotação @Embeddable

- Considere o domínio: Toda pessoa possui um endereço.
- Pode-se criar duas classes: Pessoa e Endereço:

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

 private String nome;

 @Temporal (TemporalType.DATE)
 private Calendar nascimento;

 @OneToOne
 private Endereco endereco;
}
```

Anotação @Embeddable

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

 private String pais;
 private String estado;
 private String cidade;
 private String logradouro;
 private String numero;
 private String complemento;
}
```

Anotação @Embeddable

- 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, pretende- se guardar os endereços das pessoas na própria tabela Pessoa, tornando desnecessária a existência da tabela Endereco.
- Porém, deseja-se manter as classes Pessoa e Endereco.

Anotação @Embeddable

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

Anotação @Embeddable

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

 private String nome;

 @Temporal (TemporalType.DATE)
 private Calendar nascimento;

 private Endereco endereco;
}
```

```
@Embeddable
public class Endereco {

 private String pais;
 private String estado;
 private String cidade;
 private String logradouro;
 private String numero;
 private String complemento;
}
```

- Também poderia-se:
  - Na classe Pessoa, **substituir** a anotação de multiplicidade **@OneToOne** por **@Embeddable**.
  - Na classe Endereco, deve-se **remover** a anotação **@Entity**.

Carregando um Objeto com os Dados do Banco de Dados

- Os Entity Managers administram as instâncias das entidades, eles são responsáveis pelo carregamento do estado dos objetos.
- Viu-se que há dois **modos de carregar um objeto**
- com os dados obtidos de um BD:
  - **LAZY**: o provedor posterga ao máximo a busca dos dados no banco de dados.
  - **EAGER:** o provedor busca imediatamente os dados no banco de dados.

Fetch – Tipo Básico

- Alguns dados de um determinado objeto podem ser muito grandes e nem sempre necessários.
- Por exemplo: um objeto do tipo Livro, que possua como atributos título, preço e resumo:
  - Título e preço são dados pequenos,
  - Resumo pode ser um dado relativamente grande.
- Suponha que o resumo será utilizado pela aplicação em situações bem esporádicas. Assim, seria interessante que o **valor desse atributo fosse carregado apenas quando utilizado**.

Fetch – Tipo Básico

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

```
@Basic(fetch=FetchType.LAZY)
Protected String getNome(){
  return nome;
}
```

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

Fetch – Associação

- Pode-se também definir o modo de carregamento que se deseja utilizar para as associações.
- **Por padrão** tem-se os seguintes modos de carregamento ou recuperação dos dados de um BD:
  - EAGER: One-to-One e Many-to-One
  - LAZY: One-to-Many e Many-to-Many
- Porém, pode-se alterar esse comportamento padrão, aplicando-se a propriedade fetch na anotação.

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

Cascade

- Importância do cascade
  - O cascade define a propagação das operações de persistência
  - Importante para o ciclo de vida das entidades relacionadas
  - Cascade mal planejado pode destruir uma aplicação
    - Problemas de desempenho
    - Perda de dados por cascade - CascadeType.Remove
    - Deixar “lixo” no banco de dados

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.
- Exemplo, suponha uma associação entre estado e governador.

Cascade

- 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.

```
em.getTransaction().begin();
Governador governador = new Governador();
governador.setNome("João Roberto");
Estado estado = new Estado();
estado.setNome("São Paulo");

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

em.persist(estado);
em.getTransaction().commit();
```

Cascade

- Para evitar erro, é preciso persistir os 2 objetos:

```
em.getTransaction().begin();
Governador governador = new Governador();
governador.setNome("João Roberto");
Estado estado = new Estado();
estado.setNome("São Paulo");

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

em.persist(estado);
em.persist(governador);

em.getTransaction().commit();
```

- Ou configurar para que a operação persist() seja aplicada em cascata com o atributo **cascade**.

Cascade

- Classe estado configurada para a operação persist() em cascata.

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

 private String nome;

 @OneToOne (cascade=CascadeType.PERSIST, fetch=FetchType.LAZY)
 private Governador governador;

 //Getters e Setters
}
```

Cascade

- O atributo cascade das anotações de relacionamento pode ser utilizado para configurar o comportamento em cascata para as outras operações dos EntityManagers:
  - CascadeType.PERSIST
  - CascadeType.DETACH
  - CascadeType.MERGE
  - CascadeType.REFRESH
  - CascadeType.REMOVE
  - CascadeType.ALL

Consultas com 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.

```
public void Exemplo() {
String jpql = "SELECT e FROM Aluno e";
Query query = em.createQuery(jpql);
}
```

Consultas com JPQL

- **Consultas Dinâmicas**:
  - Apesar da flexibilidade, criar consultas dinâmicas pode prejudicar a performance da aplicação.
  - 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.

Consultas com JPQL

- **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 pré-compiladas no BD melhorando a performance das consultas.

Consultas com JPQL

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

Consultas com JPQL

- **Named Query**:

```
@NamedQuery (name ="Aluno.findAll",query="SELECT p FROM Aluno p")
 class Aluno {
 . . .
 }
```

```
@NamedQueries( {
@NamedQuery (name ="Aluno.findAll",query="SELECT p FROM Aluno p")
@NamedQuery (name ="Aluno.count",
                          query="SELECT COUNT(p) FROM Aluno p")
})
 class Aluno {
 . . .
 }
```

Consultas com JPQL

- Para executar uma Named Query, deve-se 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.

```
public void listaAlunos() {
    Query query = em.createNamedQuery("Aluno.findALL");
    List<Aluno> alunos = query.getResultList();
}
```

Parametrizando as Consultas com JPQL

- Para tornar as consultas em JPQL mais genéricas e evitar problemas com SQL Injection, deve-se parametrizá-las.
- Para adicionar um parâmetro basta utilizar caractere **":"** seguido do nome do argumento.

```
@NamedQuery (name ="Aluno.findbyIdade",
        query="SELECT p FROM Aluno p WHERE p.idade > : idade")
```

```
public void listaAlunos() {
    Query query = em.createNamedQuery("Aluno.findbyIdade");
    query.setParameter("idade", 18);
    List<Aluno> alunosMais18 = query.getResultList();
}
```

Parametrizando as Consultas com JPQL

- Além de definir os valores dos argumentos, pode-se adicionar parâmetros em uma consulta de maneira ordinal com **"?"** seguido de um número.

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

```
public void listaAlunos() {
    Query query = em.createNamedQuery("Aluno.findbyIdade");
    query.setParameter(1, 18);
    List<Aluno> alunosMais18 = query.getResultList();
}
```

Consultas OO com Criteria

- O segundo mecanismo definido pela especificação JPA 2 para consultas OO é chamado de Criteria e utiliza um conjunto de classes e interfaces e funciona basicamente como uma biblioteca.
- As duas interfaces principais são **CriteriaBuilder** e
- **CriteriaQuery**
- As consultas em Criteria são construídas por um Criteria Builder que é obtido através de um EntityManager.

Consultas OO com Criteria

- A definição de uma consulta emCriteria começa na chamada do método **createQuery()** de um Criteria Builder.
- O parâmetro desse método indica o tipo esperado do resultado da consulta.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery <Autor> c = cb.createQuery(Autor.class);
```

Consultas OO com 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.
- Para definir o que se deseja selecionar do espaço de dados da consulta, devemos utilizar o método **select()** da Criteria Query.

Consultas OO com Criteria

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery <Autor> c = cb.createQuery(Autor.class);

//Definido o espaço de dados da consulta
Root<Autor> a = c.from(Autor.class);

//Selecionando uma raiz do espaço de dados
c.select(a);
```

- Com a consulta em Criteria definida, deve-se invocar um Entity Manager para poder executá-la da seguinte forma:

```
TypedQuery <Autor> query = em.createQuery(c);
List<Autor> autores = query.getResultList();
```

Consultas OO com Criteria

- Vantagens:
  - **Verificação de erros** – Muitos erros podem ser identificados em tempo de compilação do projeto;
  - **Segurança** - Como a JPA trata de criar estas consultas, fica-se quase imune a SQL injections.
- Desvantagens:
  - **Complexidade** - Muitos desenvolvedor estão acostumados a utilizar linguagens SQL/HQL/JPQL e migrar pra esta API Criteria não é simples.

Consultas OO com JPQL e Criteria

- 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** são mais facilmente definidas em JPQL.
  - Por ex., buscar todos os livros cadastrados no BD sem nenhum filtro

Consultas OO com JPQL e Criteria

- As consultas que **dependem de informações externas fixas** são mais facilmente criadas em JPQL utilizando os parâmetros de consulta.
  - Por ex., buscar todos os livros cadastrados no BD que possuam preço maior do que um valor definido pelos usuários
- As **consultas que dependem de informações externas não fixas** são mais facilmente definidas em Criteria.
  - Por ex., 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.

Bibliografia Recomendada

- http://www.hibernate.org
- Persistência com JPA2 e Hibernate. Treinamento K19.
- Enterprise JavaBeans- Tutorial Java EE 6. Parte VI Persistence http://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html
- Sintaxe do JPQL: http://docs.oracle.com/cd/E28613_01/apirefs.1211/e24396/ejb3_langref.html

creative commons

Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)

Você tem a liberdade de:

Compartilhar — copiar, distribuir e transmitir a obra.

Remixar — criar obras derivadas.

Sob as seguintes condições:

Atribuição — Você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra).

Compartilhamento pela mesma licença — Se você alterar, transformar ou criar em cima desta obra, você poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente.

Este é um resumo amigável da Licença Jurídica (a licença integral) Advertência

Imagens

- http://www.digitalprank.org/wp-content/uploads/2008/03/ejb.png
- http://sandersconsulting.com/Portals/58319/images/checklist.jpg
- http://vip.cs.utsa.edu/classes/cs4393f2006/lectures/images/overview-j2eeArchitecture.gif
- http://www.dave-woods.co.uk/wp-content/uploads/2010/02/screens.gif

- http://www.ishopping.pk/product_images/y/423/iMac-Lion__45051_std.png
- http://www.wegotserved.com/wp-content/uploads/2011/08/MacMiniServer.jpg

Obrigado

Leandro Pupo Natale

Leandro.natale@mackenzie.br