**Anotações JPA:**

* @Entity - Indica que uma classe é uma entidade que deve ser persistida como tabela.
* @Table - Indica o nome de uma tabela para qual a entidade é mapeada.
* @Column - Indica que um atributo de uma classe é uma coluna de uma tabela.
* @Id – Indica que um atributo é a chave primaria de uma entidade.
* @namedQuery - Define uma consulta.
* @namedQueries- Define varias consultas.
* @UniqueConstraint - Indica que uma propriedade não pode conter valores duplicados.
* @Transient - Indica que uma propriedade não deve ser persistida no banco de dados.
* @Inheritance - Define um relacionamento de herança ‡ entidade.
* @ManyToOne - Mapeamento muitos para um (N:1).
* @OneToMany - Mapeamento um para muitos (1:N).
* @OneToOne - Mapeamento um para um (1:1).

**Tipos de herança no Hibernate**

Estas formas são 3:

* Tabela Única por Hierarquia de Classes
* Tabela por Subclasse
* Tabela por Classe Concreta
* **Tabela Única por Hierarquia de Classes**

Este é o famoso “tabelão” com milhares de campos, onde é gerado 1 tabela para toda a hierarquia de classes. Se você tiver uma Classe Pessoa como Superclasse, e outras 2 classes PessoaFisica e PessoaJuridica estendendo de Pessoa, o Hibernate irá gerar apenas 1 tabela Pessoa com os dados de Pessoa + PessoaFisica + PessoaJuridica.

E como é feita a divisão de quem é Pessoa Física e quem é Pessoa Jurídica ? Através de um atributo chamado *discriminator*. No seu “tabelão” deverá ter um campo que faça a discriminação entre PessoaFisica (F) e PessoaJuridica (J). Veja abaixo na listagem 1 a implementação da classe Pessoa.

1. @Inheritance(strategy = InheritanceType.SINGLE\_TABLE): Identifica que a estratégia de herança será de uma única tabela, ou se preferir, um “tabelão” para tudo.
2. @DiscriminatorColumn(name = "tipo", length = 1, discriminatorType = DiscriminatorType.STRING): Identifica que o campo que armazenará o discriminator é o “tipo”, de tamanho 1, do tipo String.
3. @DiscriminatorValue("P"): Identifica que a Classe Pessoa será identifica com o atributo “P” na tabela Pessoa, ou seja, onde tiver “P” é porque aquele registro é uma pessoa.

### Tabela por Subclasse

Nessa estratégia nós teremos a Classe Pai e todas suas filhas geradas no banco de dados fisicamente, sendo que em todas as classes filhas teremos uma chave estrangeira que apontará para a classe pai.

Isto é, se tivermos 1 Classe Pessoa com 5 atributos, 1 Classe Pessoa Física com 10 atributos e 1 Classe Pessoa Jurídica com 15 atributos, o gerado será: 1 tabela pessoa com 5 atributos, 1 tabela pessoa\_fisica com 10 atributos e 1 tabela pessoa\_juridica com 15 atributos.

**@Inheritance(strategy = InheritanceType.JOINED)**: Identifica que a estratégia de herança será “JOINED”, ou seja, será feita uma junção através de chaves estrangeiras.

**@PrimaryKeyJoinColumn(name="idPessoa")**: Identifica qual campo fará essa “junção” entre a tabela pessoa\_fisica e a tabela pessoa.

### Tabela por Classe Concreta

Não menos importante, essa estratégia define que apenas as classes concretas geram suas respectivas tabelas, ou seja, em nosso exemplo apenas pessoaFisica e pessoaJuridica criarão tabelas, isso porque pessoa é uma classe abstrata. A nossa estratégia agora é “TABLE\_PER\_CLASS”, que identifica que apenas as classes concretas serão geradas.

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

# Relacionamento Bidirecional

Outra perspectiva do sentido de um relacionamento, nós permite definir os relacionamentos bidirecionais. Na prática, estes relacionamentos indicam que há um relacionamento em ambos os sentidos entre as entidades.

Diferentemente dos relacionamentos unidirecionais, os bidirecionais necessitam de mapeamento em ambas as entidades, uma vez que se relacionam, comportando-se como se existissem dois relacionamentos unidirecionais (um para cada entidade envolvida).

Nos relacionamentos bidirecionais temos o conceito de **Entidade possuidora** e **Entidade inverso**.

* Entidade possuidora: A tabela dessa entidade será a possuidora da chave estrangeira.
* Entidade inverso: O atributo deve ser anotado e configurado com mappedBy.

**Um para Um**

No geral, os mapeamentos bidirecionais, não se diferenciam muito quanto ao uso das anotações. Conforme podemos observar, no Código a seguir, temos um relacionamento **Um para Um** entre as entidades Gerente e Departamento. Quando realizarmos o mapeamento, vamos definir uma coluna na tabela Gerente que faz referência a chave primária de Departamento.

@Entity

public class Departamento {

    @OneToOne(mappedBy="departamento")

    private Gerente gerente;

}

// Agora precisamos mapear em ambas as entidades o relacionamento

@Entity

public class Gerente {

    @OneToOne

    @JoinColumn(name="DEPARTAMENTO\_ID")

    private Departamento departamento;

}

**Nota:**

A entidade **Inverso** leva a anotação @OneToOne com o parâmetro mappedBy.

A entidade **Possuidora** leva a anotação @JoinColumn. Porém, essa o uso dessa anotação, assim com a @Column não é obrigatória para identificarmos o relacionamento.

É importante destacar que o parâmetro declarado no mappedBy deve ter o mesmo nome do atributo que se desejar mapear. Neste exemplo, utilizamos, na entidade Departamento, mappedBy="departamento" e o atributo da entidade Gerente o atributo com nome departamento.

Além disso, nos relacionamentos bidirecionais, precisamos fazer a atribuição nas duas entidades, conforme visto no Código a seguir.

Gerente gerente = new Gerente();

Departamento departamento = new Departamento();

gerente.setDepartamento(departamento);

departamento.setGerente(gerente);

**Um para Muitos e Muitos para Um**

Nos relacionamentos unidirecionais temos duas maneiras de mapear o relacionamento de uma para muitas entidades, com as anotações @OneToMany e @ManyToOne. Entretanto, quando realizamos um mapeamento bidirecional temos uma única forma de fazê-lo.

Conforme podemos observar, no Código a seguir, temos um relacionamento **Um para Muitos** entre as entidades Gerente e Projeto; e, outro relacionamento de **Muitos para Um** entre Projeto e Gerente.

Quando realizarmos o mapeamento, será criada uma tabela auxiliar para armazenar as chaves de Gerente e Projeto. Contudo, sempre que necessitarmos de uma instância das entidades envolvidas no relacionamento bidirecional, o provider precisará realizar uma junção entre as três tabelas. Esse procedimento pode causar uma diminuição de performance da aplicação.

Uma outra estratégia permite que seja definida uma coluna na tabela Projeto que faz referência a chave primária de Gerente. Para isso, utilizamos a anotação @JoinColumn, no lado muitos do relacionamento.

@Entity

public class Projeto {

    @ManyToOne

    @JoinColumn(name="GERENTE\_ID")

    private Gerente gerente;

}

@Entity

public class Gerente {

    @OneToMany(mappedBy="gerente")

    private Set<Projeto> projetos;

}

**Nota:**

A especificação exige que o lado possuidor do relacionamento seja o lado **Muitos** e não o **Um**.

**Muitos para Muitos**

Na prática, um relacionamento **Muitos para Muitos** bidirecional se diferencia do unidirecional por manter referências em ambas as entidades. Como no mapeamento unidirecional, também será criada uma tabela auxiliar para manter as chaves estrangeiras das duas tabelas.

Conforme podemos observar, no Código a seguir, temos um relacionamento **Muitos para Muitos** entre as entidades Funcionario e Projeto; e, outro relacionamento de **Muitos para Muitos** entre Projeto e Funcionario.

@Entity

public class Projeto {

    @ManyToMany(mappedBy = "projetos")

    private Set<Funcionario> funcionarios;

}

@Entity

public class Funcionario {

    @ManyToMany

    @JoinTable(name = "TrabalhaEmProjeto"

            joinColumns = @JoinColumn(name = "func\_id"),

            inverseJoinColumns = @JoinColumn(name = "proj\_id"))

    private List<Projeto> projetos;

}

**Nota:**

No relacionamento bidirecional de **Muitos para Muitos** não faz diferença a entidade que será a possuidora ou inversa, pois sempre será gerado a tabela auxiliar.

**SessionFactory (org.hibernate.SessionFactory):** trata-se de um objeto pesado, thread-safe e imutável de coleções de mapeamentos objeto-relacional para um único banco

de dados – deve existir apenas uma instância na aplicação. É uma fábrica de

sessões e é necessário um SessionFactory por banco de dados utilizando um

arquivo de configuração separado – semelhante ao EntityManagerFactory do JPA.

**Session (org.hibernate.Session):** trata-se da comunicação entre aplicação e objetos, para isso acontecer ela faz uso de várias operações citada na assertiva D cada vez que uma interação com banco de dados for necessária. A Session é criada por uma SessionFactory (fábrica de sessões).