Conteúdo

@Entity

Uma entidade é um objeto de domínio de persistência. Normalmente uma entidade representa uma tabela em um banco de dados relacional, e cada instância de entidade corresponde a uma linha (registro) nessa tabela. O artefato principal de programação de uma entidade é a classe.

Classes persistentes devem ser anotadas com a anotação @Entity, desta forma o Hibernate consegue mapear estas classes para uma tabela no banco de dados.

Ex:

```
package com.ifpr.carro;

import javax.persistence.Entity;

@Entity
public class Carro {

}
```

@Table

Esta anotação deve ser usada para especificar o nome da tabela que será usada para mapear a classe no banco de dados. Caso não seja usada esta anotação, o Hibernate assumirá que existe no banco de dados uma tabela com o mesmo nome da classe. Se o Hibernate estiver configurado no arquivo “persistence.xml” para criar as tabelas no banco de dados, ele irá criar uma tabela com o mesmo nome da classe caso esta anotação não tenha sido utilizada para especificar o nome da tabela.

Ex:

```
package com.ifpr.carro;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="tb_carro")
public class Carro {

}
```

@Column

Deve ser utilizada logo acima de um atributo de uma classe e serve para configurar opções das colunas no banco de dados como:

- nome da coluna;
- tamanho do campo;
- se deve ser criado um índice único;
- se pode ou não ser um valor nulo.

Ex:

```
package com.ifpr.carro;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="tb_carro")
public class Carro {

	@Column(name="modelo", unique=false, nullable=false)
	private String model;
}
```

@Id

Esta anotação deve ser usada para identificar qual atributo da classe será usado como chave primária na tabela do banco de dados.

Ex:

```
package com.ifpr.carro;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="tb_carro")
public class Carro {

	@Id
	private long codigo;

}
```

@GeneratedValue

Esta anotação deve ser utilizada junto com a anotação **@Id** e ela serve para determinar como a chave primária da tabela deve ser gerada. Recebe como parâmetro qual a estratégia que deve utilizada para criar um valor para a chave primária e varia conforme o banco de dados. O parâmetro chama-se "*strategy*" e ele recebe um atributo de uma Enum como valor.

- GenerationType.AUTO

Geração automática, o Hibernate escolhe qual a melhor estratégia de acordo com o banco de dados que está sendo utilizado.

- GenerationType.IDENTITY

Esse gerador suporta colunas de identidades no DB2, MySql, MS Sql Server, Sybase e HypersonicSQL, o atributo deve ser do tipo long, short ou int.

- GenerationType.SEQUENCE

Esse gerador, gera uma sequência no DB2, PostgreSQL, Oracle e SAP DB. Deve ser usado junto com o atributo "sequence" para especificar o nome da sequência que deve utilizada.

- GenerationType.TABLE

Esse gerador utiliza uma tabela auxiliar que guarda o último valor gerado para a chave primária.

Ex:

```
package com.ifpr.carro;

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

@Entity
@Table(name="tb_carro")
public class Carro {

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

}
```

@Enumerated

Esta anotação serve para identificar atributos enumerados, possui um parâmetro que identifica com qual tipo o campo enumerado deve ser salvo no banco de dados:

- EnumType.STRING
- EnumType.ORDINAL

Ex:

```
public enum Sexo {

	MASCULINO,
	FEMININO;
}
```

```
package com.ifpr.carro;

import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.Table;

@Entity
@Table(name="tb_carro")
public class Carro {

	@Enumerated(EnumType.STRING)
	private Sexo sexo;

}
```

@Temporal

Esta anotação deve ser utilizada para mapear atributos do tipo “Date” e pode recebe como parâmetro qual o tipo do campo no banco de dados:

- TemporalType.DATE

  Somente data no formato yyyy-MM-dd

- TemporalType.TIME

  Somente hora no formato HH:mm:SS

- TemporalType.TIMESTAMP

  Data no formato completo yyyy-MM-dd HH:mm:ss

Ex:

```
package com.ifpr.carro;

import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name="tb_carro")
public class Carro {

	@Temporal(TemporalType.DATE)
	private Date dataNascimento;

}
```

Associações

Quando usamos a palavra associações, sempre nos referimos a relacionamentos entre entidades. Veremos agora técnicas de mapeamento de associações de entidade com todos os tipos de multiplicidade, com e sem coleções.

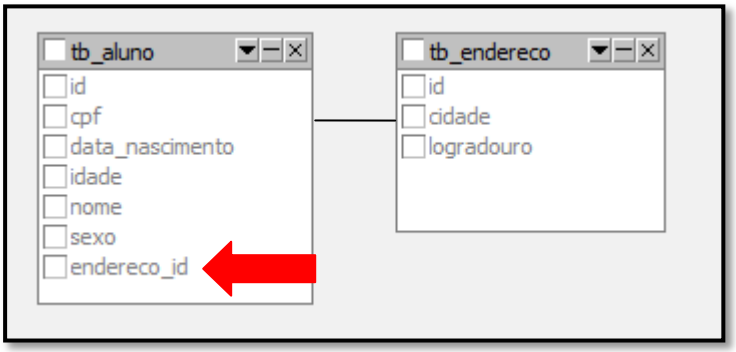
@OneToOne

Serve para identificar relacionamentos um-para-um. Esta anotação cria na tabela da classe em que foi declarada uma chave estrangeira para a chave primária da tabela da classe referenciada.

No exemplo abaixo a tabela "tb_aluno" possuirá uma chave estrangeira para a chave primária da tabela "tb_endereco". O parâmetro "cascade" especifica que qualquer alteração em um objeto da classe aluno será propagada para o atributo endereço.

Ex:

```
package ifpr.aluno;

import ifpr.endereco.Endereco;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name="tb_aluno")
public class Aluno {

	@OneToOne(cascade=CascadeType.ALL)
	private Endereco endereco;
}
```



A anotação @JoinColumn pode ser utilizada para especificar parâmetros para a chave estrangeira. O atributo “name” especifica o nome da chave estrangeira e o atributo “referencedColumnName” o nome da coluna que será referenciada na classe do relacionamento.

No exemplo abaixo será criado uma chave estrangeira para a tabela “tb_endereco” na tabela “tb_aluno” com o nome “endereco_id” que fará referência para o atributo “id” da classe Endereco.

Ex:

```
package ifpr.aluno;

import ifpr.endereco.Endereco;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.OneToOne;
import javax.persistence.Table;
import javax.persistence.JoinColumn;

@Entity
@Table(name="tb_aluno")
public class Aluno {

	@JoinColumn(name=”endereco_id”, referencedColumnName=”id”)
	@OneToOne(cascade=CascadeType.ALL)
	private Endereco endereco;
}
```

@OneToMany

Uma associação de entidade multivalorada é por definição uma coleção de referências de entidade. As associações um-para-muitos são o tipo mais importante de associações de entidade que envolve uma coleção.

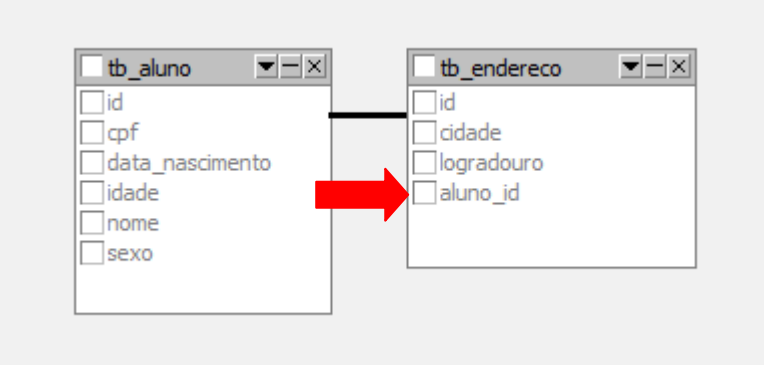
Por padrão, no Hibernate, as coleções são carregadas somente quando acessadas pela primeira vez na aplicação. Essa é uma característica importante se você for mapear uma coleção de referências de entidade possivelmente grande.

A anotação @JoinColumn pode ser utilizada para especificar parâmetros para a chave estrangeira. O atributo “name” especifica o nome da chave estrangeira e o atributo “referencedColumnName” o nome da coluna que será referenciada na classe do relacionamento.

Ex:

```
package ifpr.aluno;

import ifpr.endereco.Endereco;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name="tb_aluno")
public class Aluno {

	@JoinColumn(name = "aluno_id", referencedColumnName = "id")
	@OneToMany(cascade=CascadeType.ALL)
	private List<Endereco> endereco = new ArrayList<Endereco>();
}
```

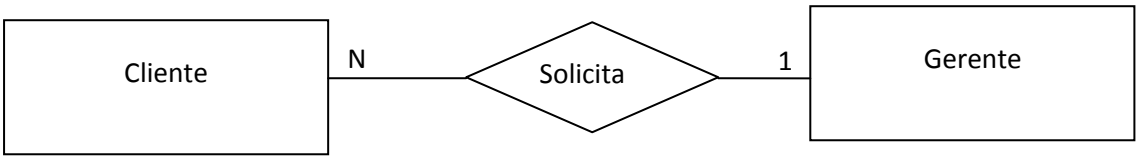
No exemplo acima a tabela endereço possuirá uma chave estrangeira para a tabela aluno, o nome da chave estrangeira está especificado no parâmetro “name” da anotação @JoinColumn e a coluna referenciada no atributo “`referencedColumnName`”. O banco ficará da seguinte maneira:



@ManyToOne

Esta anotação faz o inverso da anterior.

Ex:

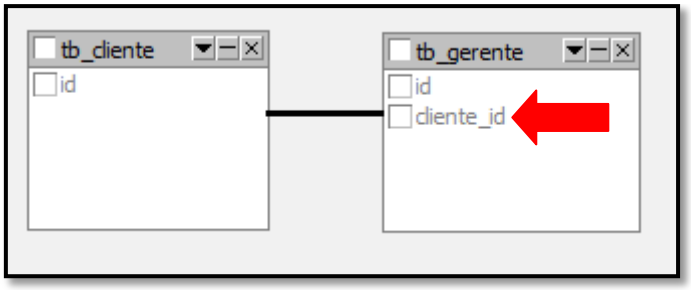


Em um relacionamento many-to-one várias instâncias de uma entidade podem estar relacionadas com uma única instância de outra entidade. Ex: vários clientes solicitam atendimento para o gerente de suas contas.

Ex:

```
package ifpr.gerente;

import ifpr.cliente.Cliente;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name="tb_gerente")
public class Gerente {

	@JoinColumn(name="cliente_id", referencedColumnName = "id")
	@ManyToOne()
	private Cliente cliente = new Cliente();
}
```

Veja como ficará o banco:



@ManyToMany

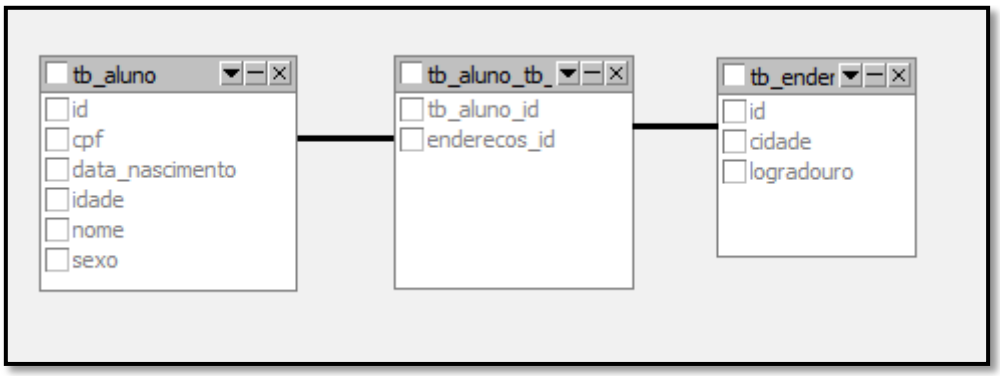
Um relacionamento muitos-para-muitos sempre deve ser resolvido por dois relacionamentos um-para-muitos. Neste caso, inicialmente, as chaves de ambos os objetos deverão ser identificadas e, a seguir, um "objeto de interseção" deverá ser descoberto ou criado. A chave do objeto de interseção é a combinação ou concatenação das chaves dos dois objetos originais. O objeto de interseção nada mais é do que uma tabela intermediária que possui duas chaves estrangeiras como chave primária.

Para mapear este tipo de relacionamento, basta colocar a anotação @ManyToMany em cima do atributo que deseja mapear. Caso o Hibernate esteja configurado para criar o banco, ele irá criar automaticamente uma tabela intermediária para fazer a associação entre as entidades envolvidas no relacionamento. Caso seja necessário especificar o nome da tabela de junção podemos utilizar a anotação @JoinTable.

No exemplo abaixo um aluno pode ter mais de um endereço e um mesmo endereço pode pertencer a mais de aluno, o Hibernate criará uma tabela intermediária chamada "tb_aluno_tb_endereco" e fará referência para as chaves primárias das duas classes:

```
package ifpr.gerente;

import ifpr.cliente.Cliente;
import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name="tb_aluno")
public class Aluno {

	@Id
	@Column(name="aluno_id")
	private int id;

	@JoinTable(
		name="tb_aluno_tb_endereco",
		joinColumns={@JoinColumn(name="aluno_id")},
		inverseJoinColumns={@JoinColumn(name="endereco_id")}
	)
	@ManyToMany
	private List<Endereco> enderecos;
}
```

Veja como ficarão as tabelas no banco:



Herança

Mapear uma hierarquia de classes para tabelas pode ser uma questão muito complexa, e iremos ver algumas das estratégias mais utilizadas para mapear herança no banco de dados.

A herança é uma visível disparidade estrutural entre o mundo orientado a objetos e o mundo relacional, pois os sistemas orientados a objetos modelam tanto o relacionamento "**é um"** quanto "**tem um".** Os modelos baseados em SQL fornecem somente os relacionamentos "**tem um"** entre as entidades e não suportam herança de tipos e mesmo quando ela é disponibilizada, geralmente é proprietária e incompleta.

Existem quatro abordagens diferentes para representar uma hierarquia de herança no banco de dados:

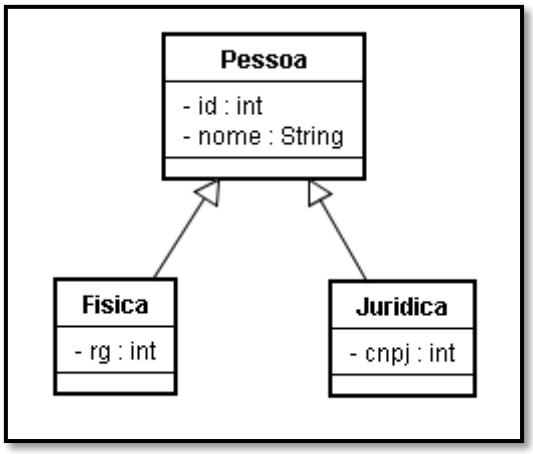
a) Tabela por classe concreta com polimorfismo implícito.
b) Tabela por classe concreta – Descarta completamente os relacionamentos com polimorfismo e com herança do esquema SQL.
c) Tabela por hierarquia de classes – Habilita o polimorfismo através da desnormalização do esquema SQL, e utiliza uma coluna discriminadora de tipo que guarda a informação do tipo.
d) Tabela por subclasse – Representa os relacionamentos **"é um"** com relacionamentos **"tem um"**(chave estrangeira)**.**

Veremos como funcionam apenas as seguintes estratégias:
- Tabela por hierarquia de classe
- Tabela por subclasse

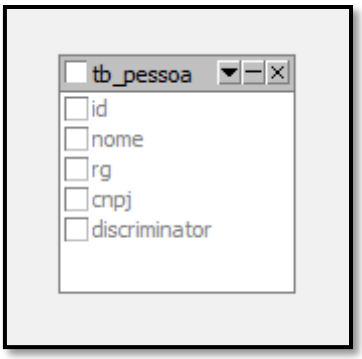
Tabela por hierarquia de classe

Uma hierarquia de classe completa pode ser mapeada para uma única tabela. Essa tabela inclui colunas para todas as propriedades de todas as classes na hierarquia. A subclasse concreta representada por uma determinada linha é identificada pelo valor de uma coluna discriminadora de tipo.

Ex:



A hierarquia de classes acima pode ser mapeada para uma tabela no banco de dados contendo todos os campos de todas as classes, veja como fica a tabela no banco:



Essa estratégia possui excelente desempenho e simplicidade. É o modo de melhor desempenho para representar o polimorfismo, é fácil de implementar, relatórios *ad hoc* são possíveis sem junções ou uniões complexas e a evolução do esquema é simples.

As estratégias de mapeamento de herança são feitos utilizando a anotação @Inheritance, esta anotação recebe um parâmetro chamado “strategy” que pode ser configurado com a enum “InheritanceType”.

Veja como fica o código das classes:

Pessoa.java

```
package ifpr.pessoa;

import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorType;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name="tb_pessoa")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
	name="discriminator",
	discriminatorType=DiscriminatorType.CHAR
)
public class Pessoa
{
	@Id
	private int id;

	private String nome;
}
```

Fisica.java

```
package ifpr.pessoa;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("F")
public class Fisica extends Pessoa
{
	private int rg;
}
```

Juridica.java

```
package ifpr.pessoa;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("J")
public class Juridica extends Pessoa
{
	private int cnpj;
}
```

Tabela por subclasse

Esta opção representa relacionamentos de herança como associações relacionais de chave estrangeira. Nesta estratégia toda classe/subclasse que declare propriedades persistentes – incluindo classes abstratas e até mesmo interfaces possuem sua própria tabela. A principal vantagem dessa estratégia é que o esquema SQL é normalizado, porém é mais difícil de implementar, até mesmo relatórios *ad-hoc* são mais complexos e seu desempenho pode ser inaceitável para hierarquia de classes muito grandes e complexas. As consultas sempre necessitam de junções de muitas tabelas ou de várias leituras seqüenciais.
Veja como ficam as classes do exemplo anterior com esta estratégia.

Pessoa.java

```
package ifpr.pessoa;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name="tb_pessoa")
@Inheritance(strategy=InheritanceType.JOINED)
public class Pessoa
{
	@Id
	private int id;

	private String nome;
}
```

Fisica.java

```
package ifpr.pessoa;

import javax.persistence.Entity;

@Entity
public class Fisica extends Pessoa
{
	private int rg;
}
```

Juridica.java

```
package ifpr.pessoa;

import javax.persistence.Entity;

@Entity
public class Juridica extends Pessoa
{
	private int cnpj;
}
```

Por fim veja como ficam as tabelas no banco de dados: