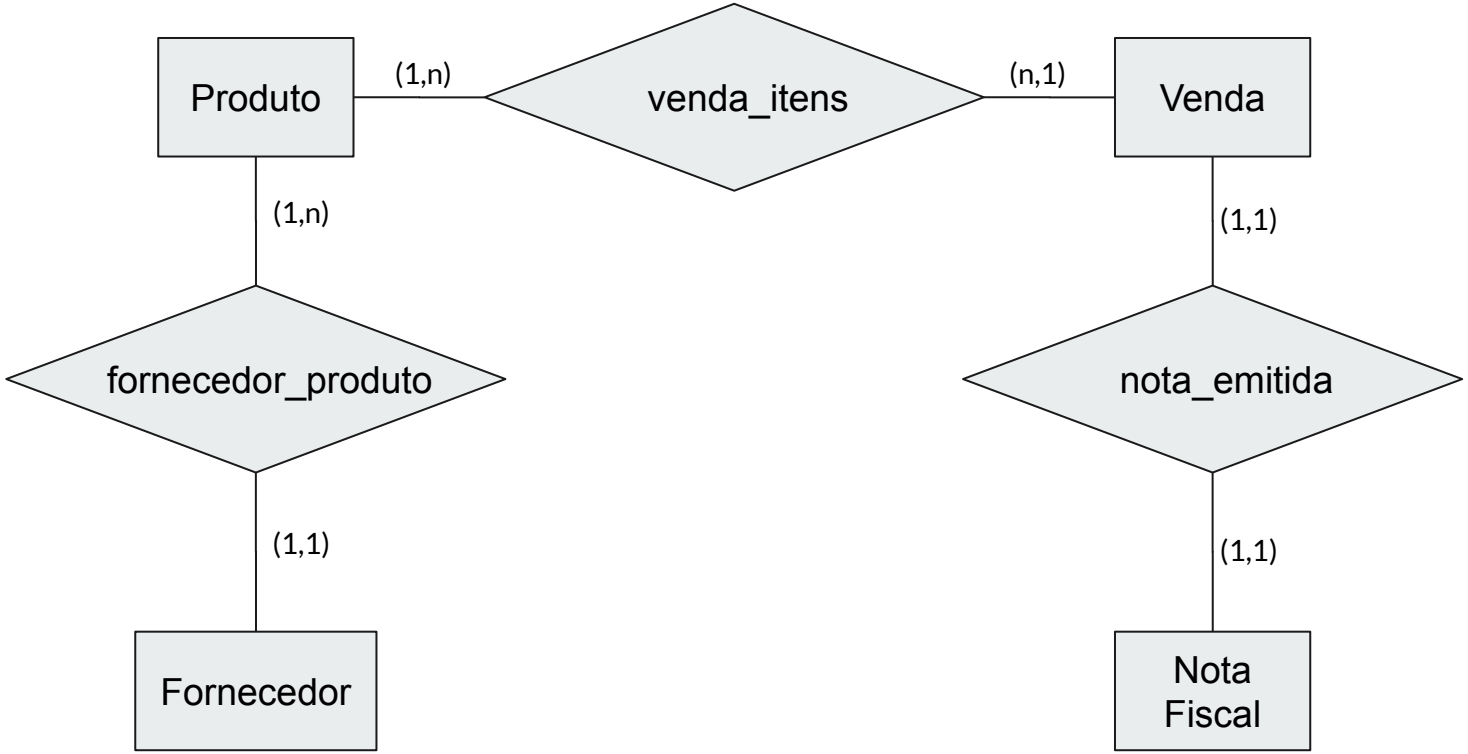
JPA (JAVA PERSISTENCE API) - Mapeamento Objeto-Relacional (MOR) - Parte 3-4

Prof. Francisco do Nascimento

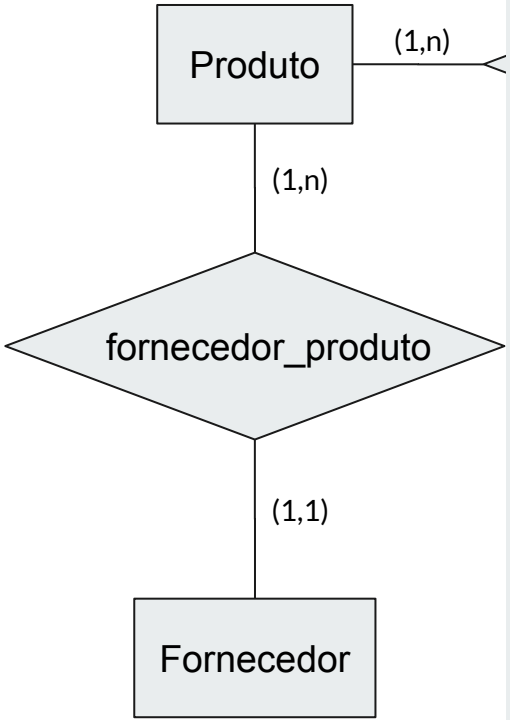
JPA

- Parte 1: Modelagem Conceitual
- Parte 2: Mapeamento de Objeto-Relacional - Classes, Atributos
- **Parte 3: Mapeamento de Objeto-Relacional - Relacionamentos, Herança**
- **Parte 4: API de consulta – JPQL**

Parte 3:
Mapeamento Objeto-Relacional
- Relacionamentos, Herança



Modelagem Conceitual



Sobre o relacionamento Produto - Fornecedor, temos:

(a) cada produto tem um fornecedor e cada fornecedor tem um produto (UM PARA UM)

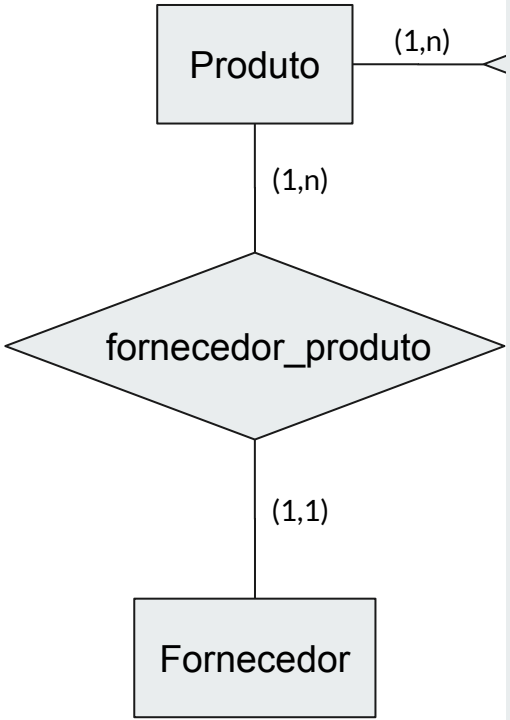
(b) cada produto tem um fornecedor e cada fornecedor pode ter vários produtos (UM PARA MUITOS)

(c) cada fornecedor pode ter vários produtos e cada produto pode ter vários fornecedores (MUITOS PARA MUITOS)

```
public class Produto {
   _____ fornecedor ?
}
public class Fornecedor {
   _____ produto ?
}
```

Decisões sobre atributos:

(1) Valor único ou múltiplo
(2) Obrigatório ou opcional
(3) Anotação:
- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany



Muitos para Um

Sobre o relacionamento Produto - Fornecedor, temos:

(a) cada produto tem um fornecedor e cada fornecedor tem um produto (UM PARA UM)

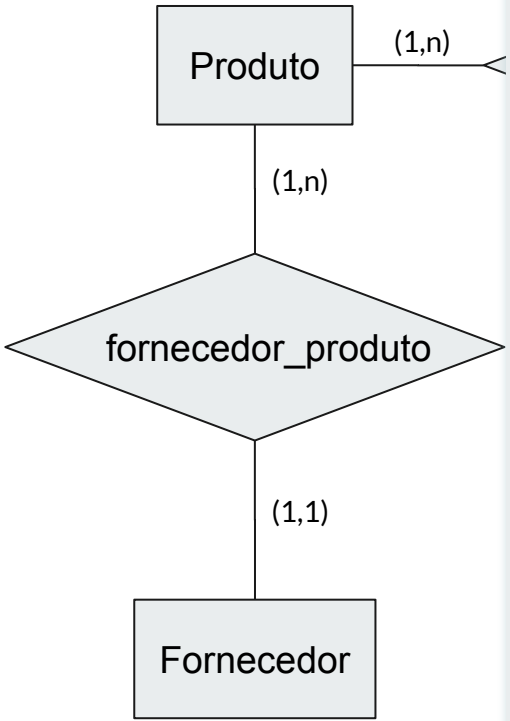
(b) cada produto tem um fornecedor e cada fornecedor pode ter vários produtos (UM PARA MUITOS)

(c) cada fornecedor pode ter vários produtos e cada produto pode ter vários fornecedores (MUITOS PARA MUITOS)

```
@Entity
public class Produto {
    private Fornecedor fornecedor;
}
@Entity
public class Fornecedor {
    private List<Produto> produtos;
}
```

(1) Valor único
(2) Obrigatório

(1) Valor múltiplo
(2) Opcional



Muitos para Um
@ManyToOne

Sobre o relacionamento Produto - Fornecedor, temos:

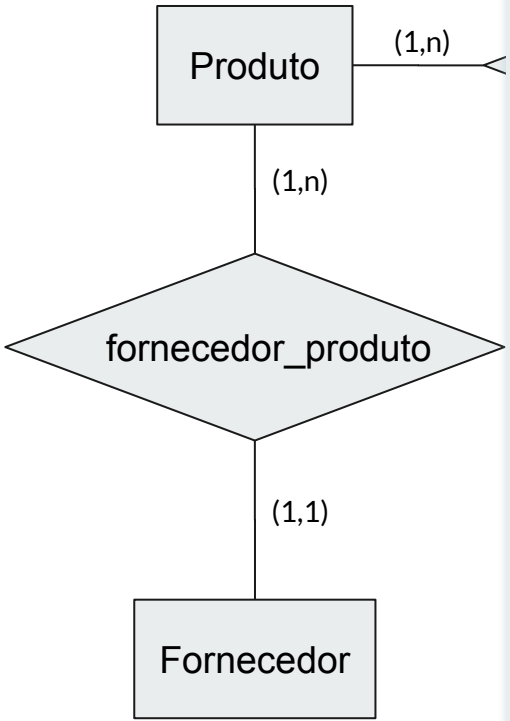
(a) cada produto tem um fornecedor e cada fornecedor tem um produto (UM PARA UM)

(b) cada produto tem um fornecedor e cada fornecedor pode ter vários produtos (UM PARA MUITOS)

(c) cada fornecedor pode ter vários produtos e cada produto pode ter vários fornecedores (MUITOS PARA MUITOS)

::: Relação Unidirecional (Não usa o lado opcional)

```
@Entity
public class Produto {
    @ManyToOne // Muitos produtos para um fornecedor
    private Fornecedor fornecedor; // atributo obrigatório
}
@Entity
public class Fornecedor {
    private List<Produto> produtos; // atributo opcional
}
```



Muitos para Um
@ManyToOne

Sobre o relacionamento Produto - Fornecedor, temos:

(a) cada produto tem um fornecedor e cada fornecedor tem um produto (UM PARA UM)

(b) cada produto tem um fornecedor e cada fornecedor pode ter vários produtos (UM PARA MUITOS)

(c) cada fornecedor pode ter vários produtos e cada produto pode ter vários fornecedores (MUITOS PARA MUITOS)

:: Relação Bidirecional (ambos - obrigatório + opcional)

```
@Entity
public class Produto {
    @ManyToOne // Muitos produtos para um fornecedor
    private Fornecedor fornecedor; // atributo obrigatório
}
@Entity
public class Fornecedor {
    @OneToMany(mappedBy="fornecedor")
   // Um fornecedor para muitos produtos
    private List<Produto> produtos; // atributo opcional
}
```

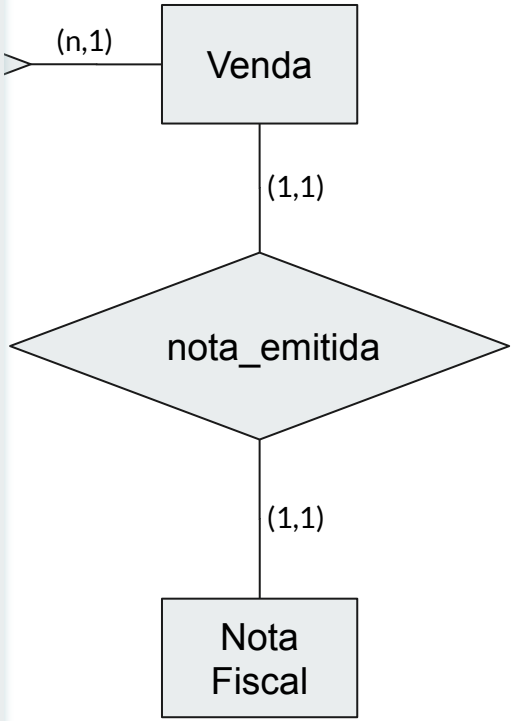
A propriedade mappedBy conecta os dois atributos

Sobre o relacionamento Venda - Nota Fiscal, temos:

(a) cada venda tem uma nota fiscal e cada nota fiscal tem uma venda (UM PARA UM)

(b) cada venda tem uma nota fiscal e cada nota fiscal pode ter várias vendas (UM PARA MUITOS)

(c) cada venda pode ter várias notas fiscais e cada nota fiscal pode ser de várias vendas (MUITOS PARA MUITOS)




```
public class Venda {
    _______ notaFiscal ?
}

public class NotaFiscal {
    _________ venda ?
}
```

Decisões sobre atributos:

(1) Valor único ou múltiplo
(2) Obrigatório ou opcional
(3) Anotação:

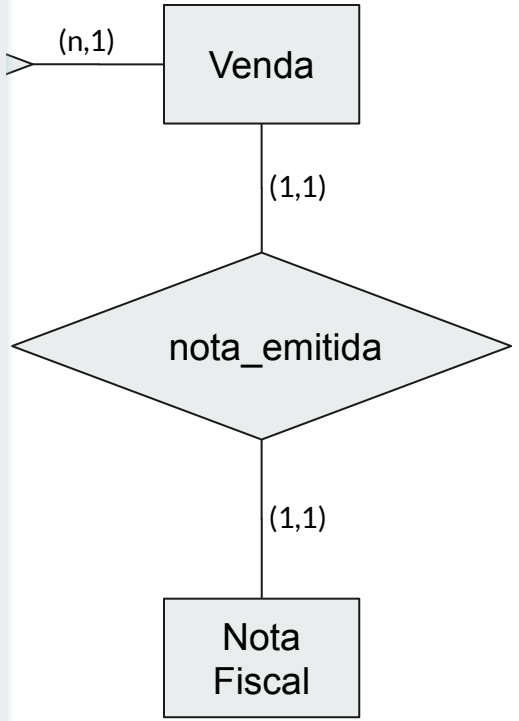
```
@OneToOne
@OneToMany
@ManyToOne
@ManyToMany
```

Sobre o relacionamento Venda - Nota Fiscal, temos:

(a) cada venda tem uma nota fiscal e cada nota fiscal tem uma venda (UM PARA UM)

(b) cada venda tem uma nota fiscal e cada nota fiscal pode ter várias vendas (UM PARA MUITOS)

(c) cada venda pode ter várias notas fiscais e cada nota fiscal pode ser de várias vendas (MUITOS PARA MUITOS)



(1) Valor único
(2) Obrigatório

```
public class Venda {
    private NotaFiscal notaFiscal;
}
public class NotaFiscal {
    private Venda venda;
}
```

(1) Valor único
(2) Opcional

Um para Um
@OneToOne

Sobre o relacionamento Venda - Nota Fiscal, temos:

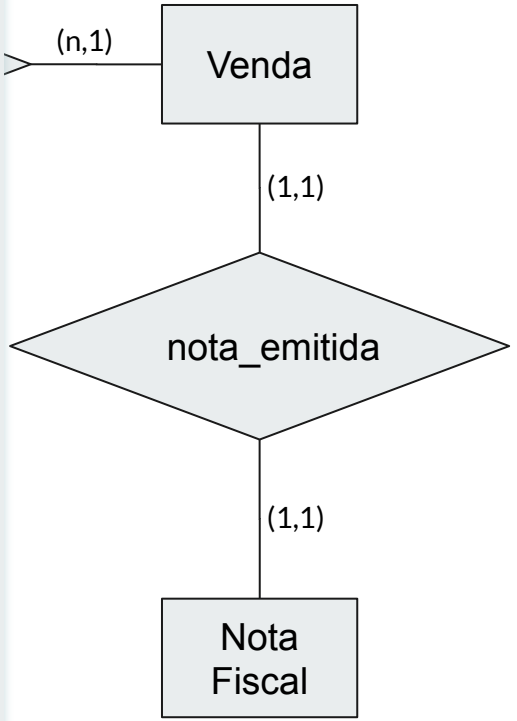
(a) cada venda tem uma nota fiscal e cada nota fiscal tem uma venda (UM PARA UM)

(b) cada venda tem uma nota fiscal e cada nota fiscal pode ter várias vendas (UM PARA MUITOS)

(c) cada venda pode ter várias notas fiscais e cada nota fiscal pode ser de várias vendas (MUITOS PARA MUITOS)

::: **Relação Unidirecional (Não usa o lado opcional)**

```
public class Venda {
    @OneToOne
    private NotaFiscal notaFiscal;
}
public class NotaFiscal {
    private Venda venda;
}
```



Um para Um
@OneToOne

Sobre o relacionamento Venda - Nota Fiscal, temos:

(a) cada venda tem uma nota fiscal e cada nota fiscal tem uma venda (UM PARA UM)

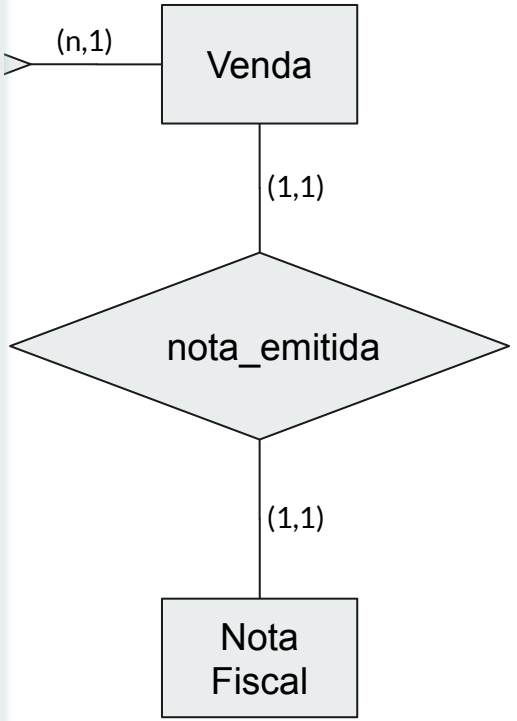
(b) cada venda tem uma nota fiscal e cada nota fiscal pode ter várias vendas (UM PARA MUITOS)

(c) cada venda pode ter várias notas fiscais e cada nota fiscal pode ser de várias vendas (MUITOS PARA MUITOS)

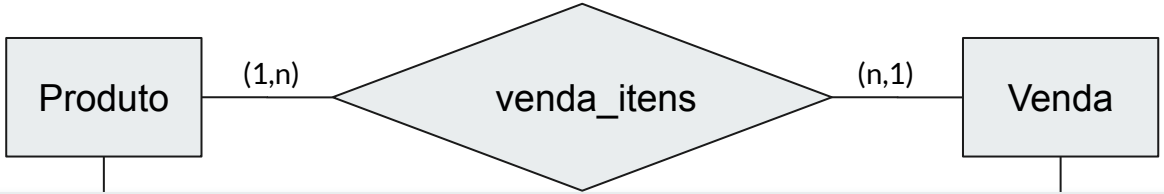
:: Relação Bidirecional (ambos: obrigatório + opcional)

```
public class Venda {
    @OneToOne
     private NotaFiscal notaFiscal;
}
public class NotaFiscal {
     @OneToOne(mappedBy="notaFiscal")
     private Venda venda;
}
```

A propriedade mappedBy conecta os dois atributos



Um para Um
@OneToOne



Sobre o relacionamento Produto - Venda, temos:

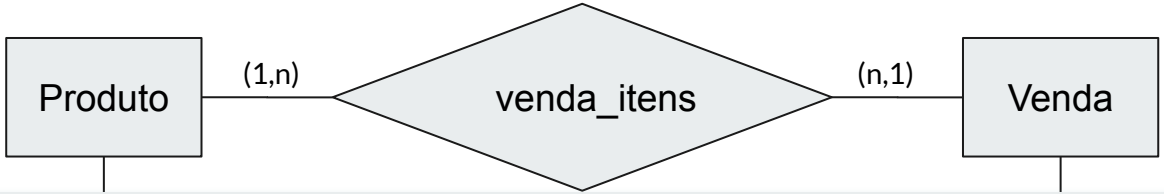
(a) cada produto está em uma venda e cada venda tem apenas um produto
(UM PARA UM)

(b) cada produto está em uma venda e cada venda pode ter vários produtos
(UM PARA MUITOS)

(c) cada produto pode estar em várias vendas e cada venda por ter vários produtos
(MUITOS PARA MUITOS)

```
public class Produto {
    _______ venda;
}
public class Venda {
    _______ produto;
}
```

Decisões sobre atributos:
(1) Valor único ou múltiplo
(2) Obrigatório ou opcional
(3) Anotação:
@OneToOne
@OneToMany
@ManyToOne
@ManyToMany



Sobre o relacionamento Produto - Venda, temos:

(a) cada produto está em uma venda e cada venda tem apenas um produto (UM PARA UM)

(b) cada produto está em uma venda e cada venda pode ter vários produtos (UM PARA MUITOS)

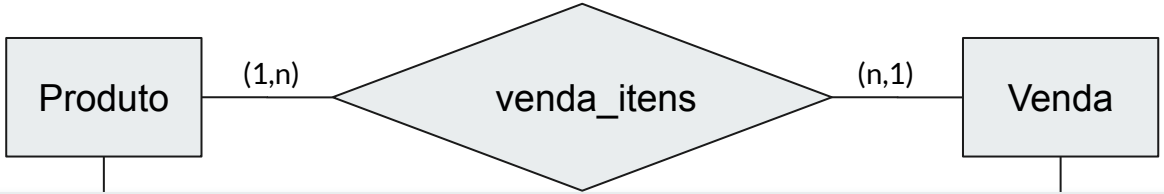
(c) cada produto pode estar em várias vendas e cada venda por ter vários produtos (MUITOS PARA MUITOS)

Muitos para muitos

```
public class Produto {
    List<Venda> vendas;
}
public class Venda {
    List<Produto> produtos;
}
```

(1) Valor múltiplo
(2) Opcional

(1) Valor múltiplo
(2) Obrigatório



Sobre o relacionamento Produto - Venda, temos:

(a) cada produto está em uma venda e cada venda tem apenas um produto (UM PARA UM)

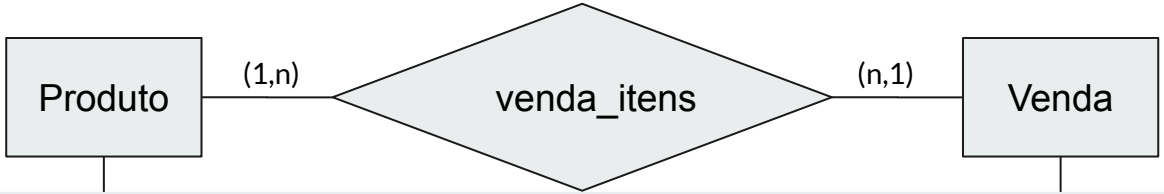
(b) cada produto está em uma venda e cada venda pode ter vários produtos (UM PARA MUITOS)

(c) cada produto pode estar em várias vendas e cada venda por ter vários produtos (MUITOS PARA MUITOS)

::: **Relação Unidirecional (Não usa o lado opcional)**

```
public class Produto {
    List<Venda> vendas;
}
public class Venda {
    @ManyToMany
     List<Produto> produtos;
}
```

Muitos para muitos
@ManyToMany



Sobre o relacionamento Produto - Venda, temos:

(a) cada produto está em uma venda e cada venda tem apenas um produto (UM PARA UM)

(b) cada produto está em uma venda e cada venda pode ter vários produtos (UM PARA MUITOS)

(c) cada produto pode estar em várias vendas e cada venda por ter vários produtos (MUITOS PARA MUITOS)

::: Relação Bidirecional (ambos: obrigatório + opcional)

```
public class Produto {
    @ManyToMany(mappedBy="produtos")
    List<Venda> vendas;
}
public class Venda {
   @ManyToMany
    List<Produto> produtos;
}
```

A propriedade mappedBy conecta os dois atributos

Muitos para muitos
@ManyToMany

@JoinColumn

- @JoinColumn informa a coluna que é chave estrangeira
- Parâmetros da @JoinColumn
  - name: nome da coluna
  - insertable: indica que o atributo será inserido quando o objeto for inserido
  - updatable: indica que o atributo será atualizado quando o objeto for atualizado

Evite relacionamentos bidirecionais em JPA

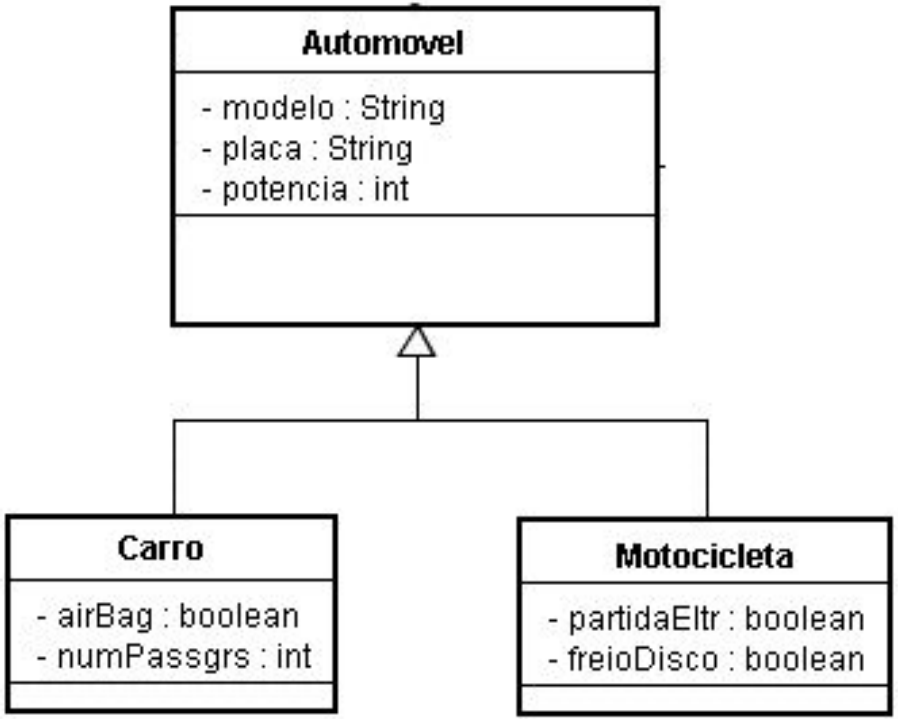
http://blog.triadworks.com.br/jpa-por-que-voce-deveria-evitar-relacionamento-bidirecional

https://www.computersciencemaster.com.br/exercicios-java-persistence-api-1/

Tipos de relacionamentos entre entidades

- JPA permite mapear os relacionamentos entre classes
  - Associação
    - @OneToOne, @OneToMany, @ManyToMany, @ManyToOne
  - Herança
    - @Inheritance
      - Estratégias:
        Única tabela - InheritanceType.SINGLE_TABLE
        Tabela por subclasse - InheritanceType.JOINED
        Tabela por classe concreta - InheritanceType.TABLE_PER_CLASSE

Mapeamento de Hierarquia de Classes



Usando @MappedSuperclass

```
@MappedSuperclass
public abstract class Automovel {}

@Entity
public class Carro {}

@Entity
public class Motocicleta {}
```

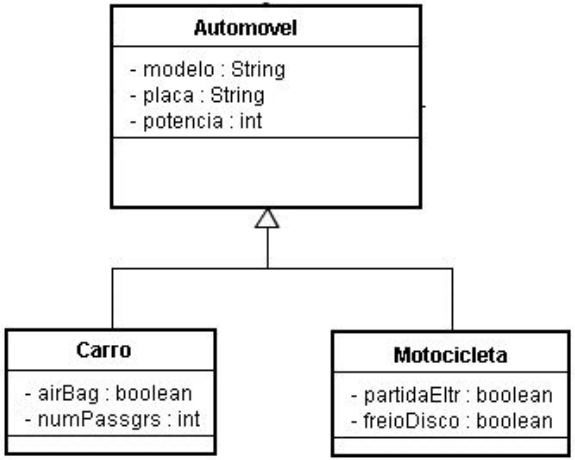
Banco de Dados: 2 tabelas

CARRO
MODELO
PLACA
POTENCIA
AIRBAG
NUMPASSGRS

MOTOCICLETA
MODELO
PLACA
POTENCIA
PARTIDAELTR
FREIODISCO

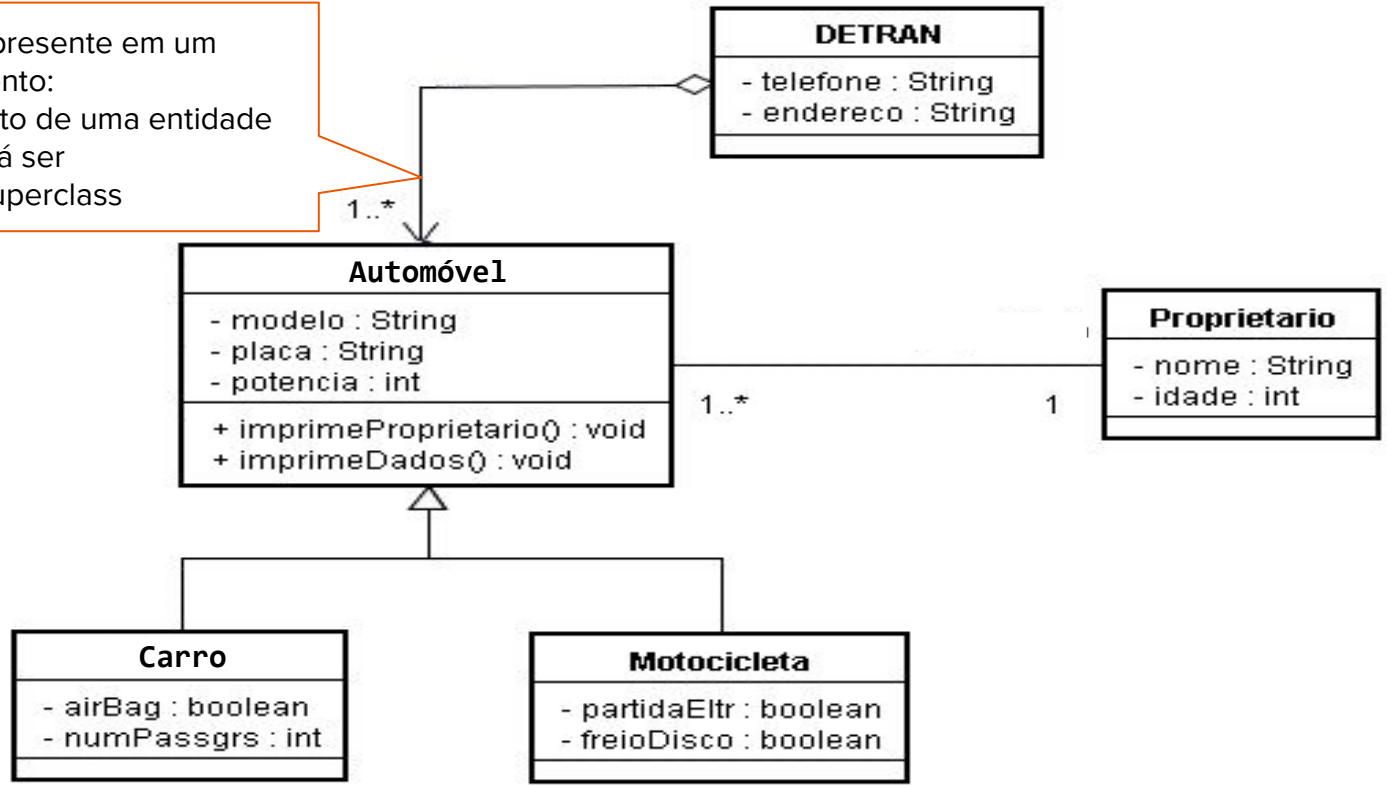
Usando @MappedSuperclass

- Define que o objeto da superclasse não é persistente, mas possui atributos persistentes
  - ➢ Não existe a tabela Automóvel
- Superclasse pode ser tanto abstrata como concreta
  - ➢ Porém, é uma boa prática tê-la como abstrata
  - Se é @MappedSuperclass, NÃO poderá ser @Entity
  - Não poderá ser atributo de uma classe que seja @Entity

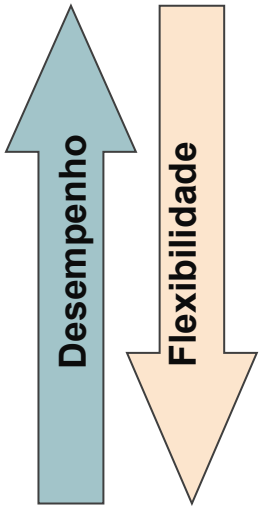


Mapeamento de Hierarquia de Classes

Automóvel presente em um relacionamento:
- Será atributo de uma entidade
- Não poderá ser @MappedSuperclass



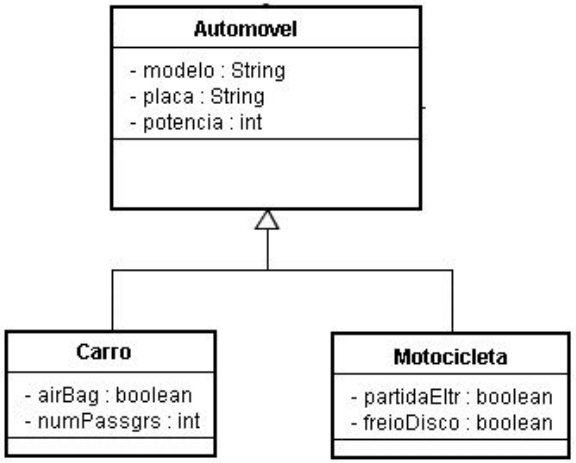
Mapeamento de Hierarquia de Classes



1. Existem 3 estratégias de mapeamento de hierarquia
    a. Uma tabela para toda a hierarquia de classes
    b. Uma tabela por classe concreta
    c. Uma tabela por subclasse

Estratégia 1: SINGLE_TABLE

- Uma tabela única para toda a hierarquia
- Junta todos os atributos em uma única tabela
- Acrescenta um novo campo a tabela para identificar o tipo do objeto



Banco de Dados

AUTOMÓVEL
MODELO
PLACA*
POTENCIA
AIRBAG
NUMPASSGRS
PARTIDAELTR
FREIODISCO
DTYPE

Estratégia 1: SINGLE_TABLE

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Automovel {}
```

```
@Entity
public class Carro {}
```

```
@Entity
public class Motocicleta {}
```

Estratégia 1: SINGLE_TABLE

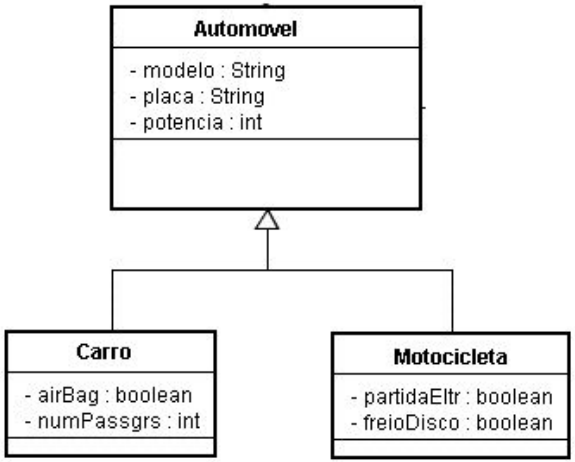
- Propriedades de uma subclasse específica não podem conter o atributo nullable=false: @Column(nullable=false)

AUTOMOVEL

PLACA *	DTYPE	MODELO	POTENCIA	AIRBAG	NUMPASSAG	PARTIDAELTR	FREIODISCO
PTO1212	CARRO	PALIO FIRE	1.0	N	5	NULL	NULL
PFA123	MOTOCICLETA	CG 150	200	NULL	NULL	S	S

Estratégia 2: TABLE_PER_CLASS

- Uma tabela para cada classe concreta: Carro e Motocicleta



Banco de Dados

CARRO	MOTOCICLETA
MODELO	MODELO
PLACA	PLACA
POTENCIA	POTENCIA
AIRBAG	PARTIDAELTR
NUMPASSGRS	FREIODISCO

Estratégia 2: TABLE_PER_CLASS

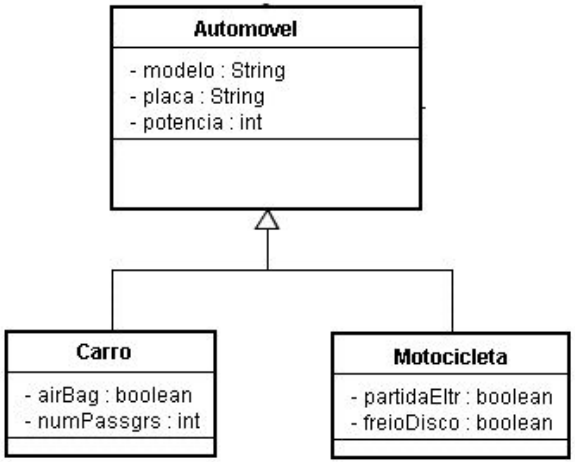
```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Automovel {}
```

```
@Entity
public class Carro {}
```

```
@Entity
public class Motocicleta {}
```

Estratégia 3: JOINED

- Uma tabela para cada classe (abstrata ou concreta): Automóvel, Carro e Motocicleta



Banco de Dados

AUTOMÓVEL
MODELO
PLACA*
POTENCIA
DTYPE

CARRO
PLACA*
AIRBAG
NUMPASSGRS

MOTOCICLETA
PLACA*
PARTIDAELTR
FREIODISCO

Estratégia 3: JOINED

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Automovel {}
```

```
@Entity
public class Carro {}
```

```
@Entity
public class Motocicleta {}
```

Painel geral sobre as três estratégias de herança

	SINGLE_TABLE: TABELA ÚNICA	**TABLE_PER_CLASS**: TABELA POR CLASSE CONCRETA	**JOINED**: TABELA POR CLASSE
Banco de dados	•Uma tabela para todas as classes •Não poderá usar not-null • Tabela aumenta quando mais subclasses são adicionadas.	•Uma tabela para cada classe concreta	•Uma tabela para a classe pai (campos comuns) • Uma tabela para cada subclasse (apenas os campos específicos)
Coluna de discriminador: DTYPE	Sim	Não	Sim
SQL para recuperação da hierarquia	SELECT simples	SELECT para cada subclasse ou UNION de SELECTs	SELECT com JOIN
SQL para inserir e atualizar	Um único INSERT ou UPDATE	INSERT ou UPDATE para cada subclasse	Vários INSERT ou UPDATE: um para cada classe envolvida
Relacionamento e consultas polimórficas	Suporta	Não suporta	Suporta
Especificação no JPA	Obrigatório	Opcional	Obrigatório

Escolha da estratégia

- A primeira estratégia é a mais simples mas restringe a utilização de not-null
  - Utilizado quando a principal diferença entre as diferentes subclasses é seu comportamento, possuindo quase o mesmo conjunto de propriedades
- A segunda resolve o problema do not-null e da diferença entre o conjunto de propriedades
  - Mas apresenta problema com associação polimórfica
- A terceira é a mais flexível
  - Permite associações com classes abstratas
  - Mas necessita de mais joins para recuperar dados

Em suma: NENHUMA das estratégias é a melhor

- Cada uma tem vantagens e desvantagens
- A escolha depende do sistema

Parte 4:
API de Consulta JPQL

https://thoughts-on-java.org/jpql/

JPQL

- Java Persistence Query Language
  - Permite a criação de consultas usando o modelo de entidades
    - Uso dos nomes de classes e atributos nas consultas (case-sensitive)
  - Agrega todo o conhecimento que já temos de SQL
  - Possibilita a criação de consultas dinâmicas

Consultas

1. SELECT p FROM Pessoa p
   1.1. p ⇒ referência para os objetos Pessoa
   1.2. Pessoa ⇒ Classe mapeada com @Entity
2. SELECT p FROM Pessoa p WHERE p.nome = :n
   2.1. p.nome ⇒ restrição
3. SELECT distinct c FROM Pessoa p JOIN p.cidade c
4. SELECT p.nome FROM Pessoa p WHERE p.cidade.nome LIKE %:n%

Caso de estudo

```
Pessoa{
    int codigo
    String nome
    int idade
}
```

```
Funcionario extends Pessoa {
Departamento depart
    Collection<Dependente> dependentes
    double salario
}
```

```
Dependente extends Pessoa {
    Funcionario func
}
```

```
Departamento{
    int codigo
    String nome
    Funcionario chefe
    Collection<Funcionario> funcs
}
```

Exemplos de Consultas usando JPQL

1. SELECT f FROM Funcionario f WHERE f.nome = ‘Joaquim’
2. SELECT f FROM Funcionario f WHERE f.nome like ‘Joa%’ and f.salario > 2000
3. SELECT f FROM Funcionario f WHERE f.dependentes is empty
4. SELECT f FROM Funcionario f WHERE f.datanascimento is null
5. SELECT f FROM Funcionario f WHERE f.departamento.codigo = 10 AND f.departamento.chefe.dependentes is empty

Funções

- TRIM (LEADING | TRAILING | BOTH)
  - Remover espaços em branco
- LOWER: Converter para minúsculas
- UPPER: Converter para maiúsculas
- LENGTH: Quantidade de caracteres
- CURRENT_DATE: Data corrente
- CURRENT_TIME: Hora corrente
- CURRENT_TIMESTAMP: Timestamp corrente

Agrupamentos

- Utilizado para aplicar funções num grupo de dados
  a. COUNT, MAX, MIN, AVG, SUM, HAVING
- Exemplos:
  a. SELECT MAX(salario) from Funcionario
  b. SELECT f.departamento.nome, COUNT(*)
     FROM Funcionario f
     GROUP BY f.departamento.nome
  c. SELECT f.departamento.nome, SUM(f.salario)
     FROM Funcionario f
     WHERE length(f.dependentes) > 1
     GROUP BY f.departamento.nome
     HAVING SUM(f.salario) > 20000

Referências...

- https://thoughts-on-java.org/jpql/
- https://www.luis.blog.br/modelagem-conceitual-modelo-conceitual-de-dados.html
- Livro EJB3 em Açao
  - Autores: Debu Panda, Reza Rahman, Derek Lane
- Livro Java Persistence with Hibernate
  - Edição revisada do livro Hibernate in Action
  - Autores: Christian Bauer, Gavin King
- http://en.wikibooks.org/wiki/Java_Persistence/Print_version
- http://www.hibernate.org/
  - Documentação e Javadoc
  - Fórum de discussões e dúvidas
- JSR 220: Enterprise JavaBeans Version 3.0
    - Para as Annotations do pacote javax.persistence