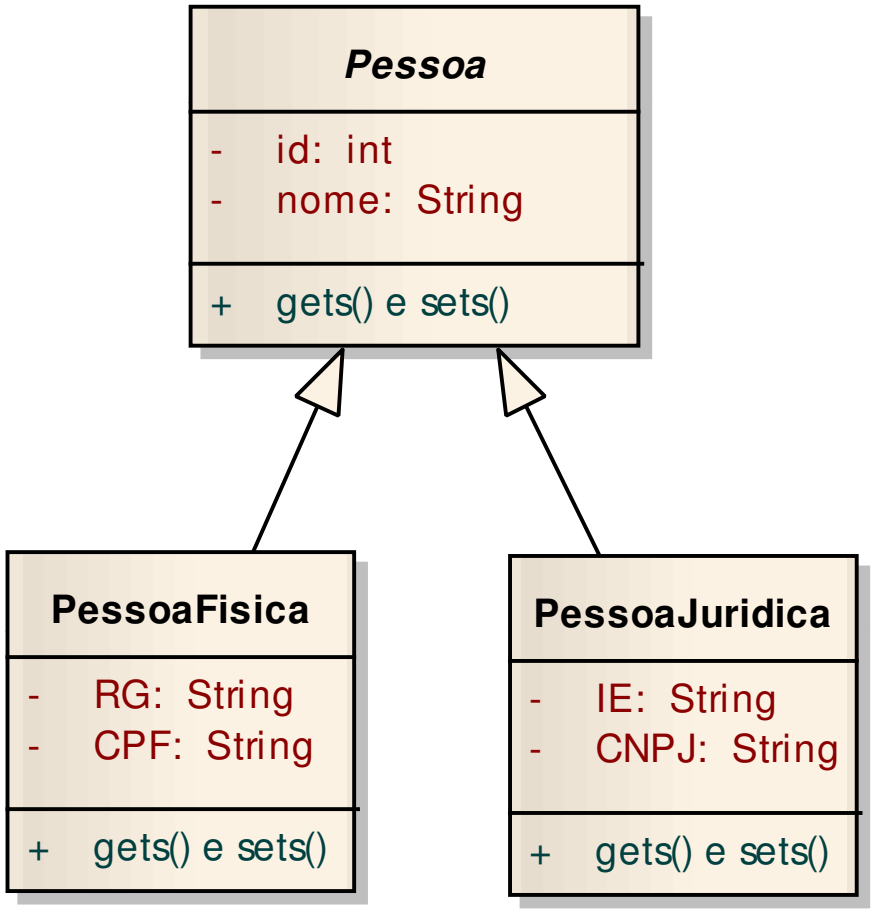
Java Persistence API (JPA)
Mapeamento de Herança

Fernando dos Santos
fernando.santos@udesc.br

UDESC
Ibirama

Herança e Banco de Dados

- Considere a seguinte situação de herança.



- Quais tabelas você teria no banco de dados?

Mapeamento de Herança

- JPA disponibiliza 3 estratégias para mapeamento de herança:
  - Tabela por **hierarquia de classes**
  - Tabela por **classe concreta**
  - Tabela por **classe e subclasse**

Tabela por hierarquia de classes

Definição

- Uma **única tabela** é criada para guardar todos os dados
  - é necessário uma **coluna discriminadora**, para identificar o tipo de dado armazenado pelo registro (pessoa física ou jurídica)

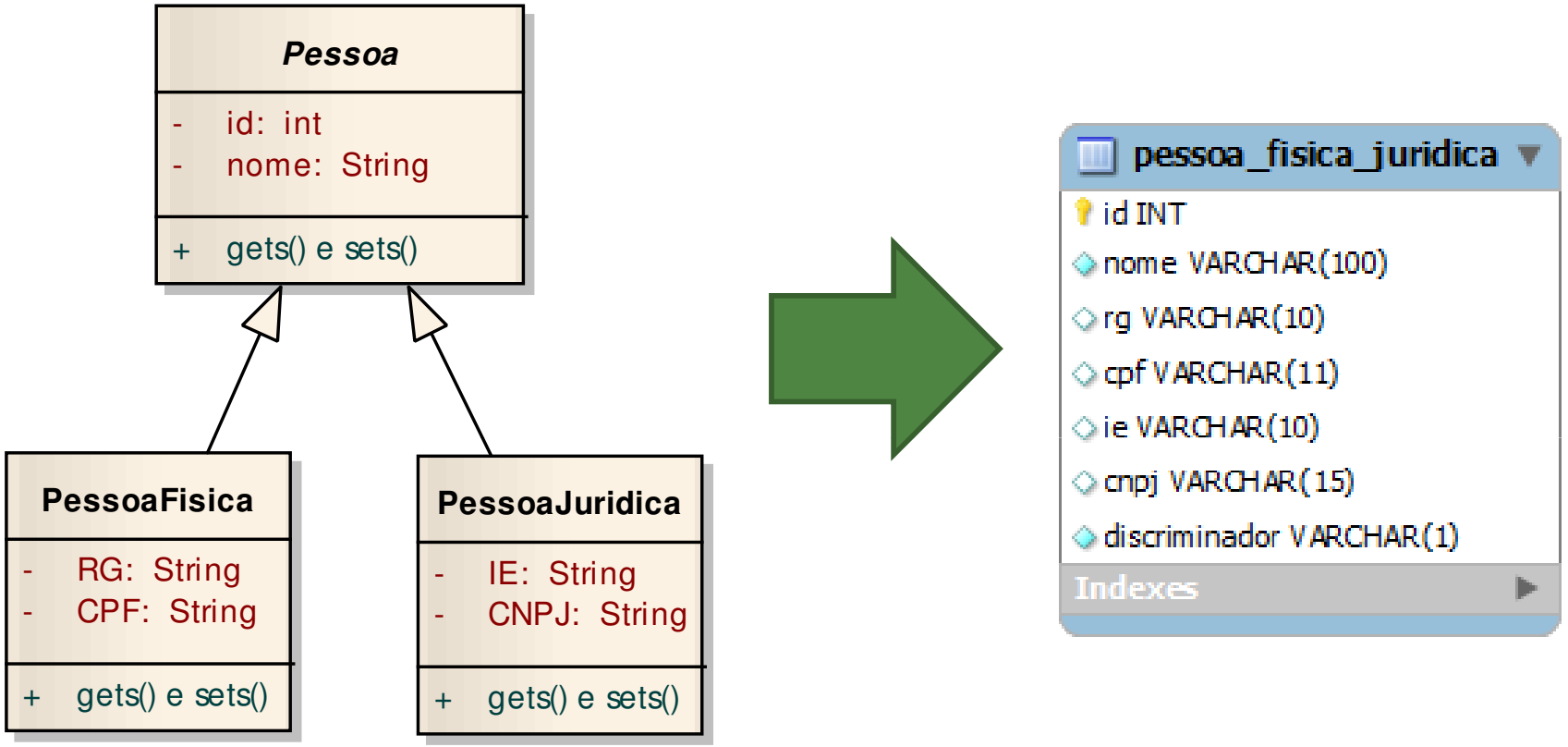


Tabela por hierarquia de classes
Mapeamento JPA

- Mapeamento na **superclasse**

```
@Entity
@Table (name="pessoa_fisica_juridica")
@Inheritance (strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn ( name="discriminador",
                       discriminatorType= DiscriminatorType.STRING)
public abstract class Pessoa implements Serializable {
    @Id
    @GeneratedValue
    private int id;

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

    // métodos get/set
}
```

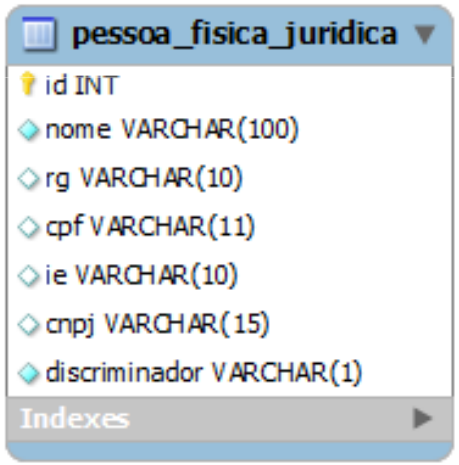


Tabela por hierarquia de classes
Mapeamento JPA

- Mapeamento nas **subclasses**

```
@Entity
@DiscriminatorValue ("F")
public class PessoaFisica extends Pessoa{
    @Column(name="rg")
    private String RG;
    @Column(name="cpf")
    private String CPF;

    // gets e sets...
}
```

```
@Entity
@DiscriminatorValue ("J")
public class PessoaJuridica extends Pessoa {
    @Column(name="ie")
    private String IE;
    @Column(name="cnpj")
    private String CNPJ;

    // gets e sets...

}
```

Tabela por hierarquia de classes

Vantagens e desvantagens

- **Vantagens**
  - é a mais simples de implementar (em Java e no BD)
  - ótima performance em consultas, pois só há uma tabela.

- **Desvantagens**
  - não é normalizada
    - desperdício de espaço com colunas que ficarão nulas no banco.
  - não permite uso da restrição **not null**
    - ex: CPF not null

Tabela por classe concreta

Definição

- Uma **tabela** é criada para cada **classe concreta**
  - classe concreta é aquela que não é abstrata
- Os campos comuns são duplicados em cada tabela.

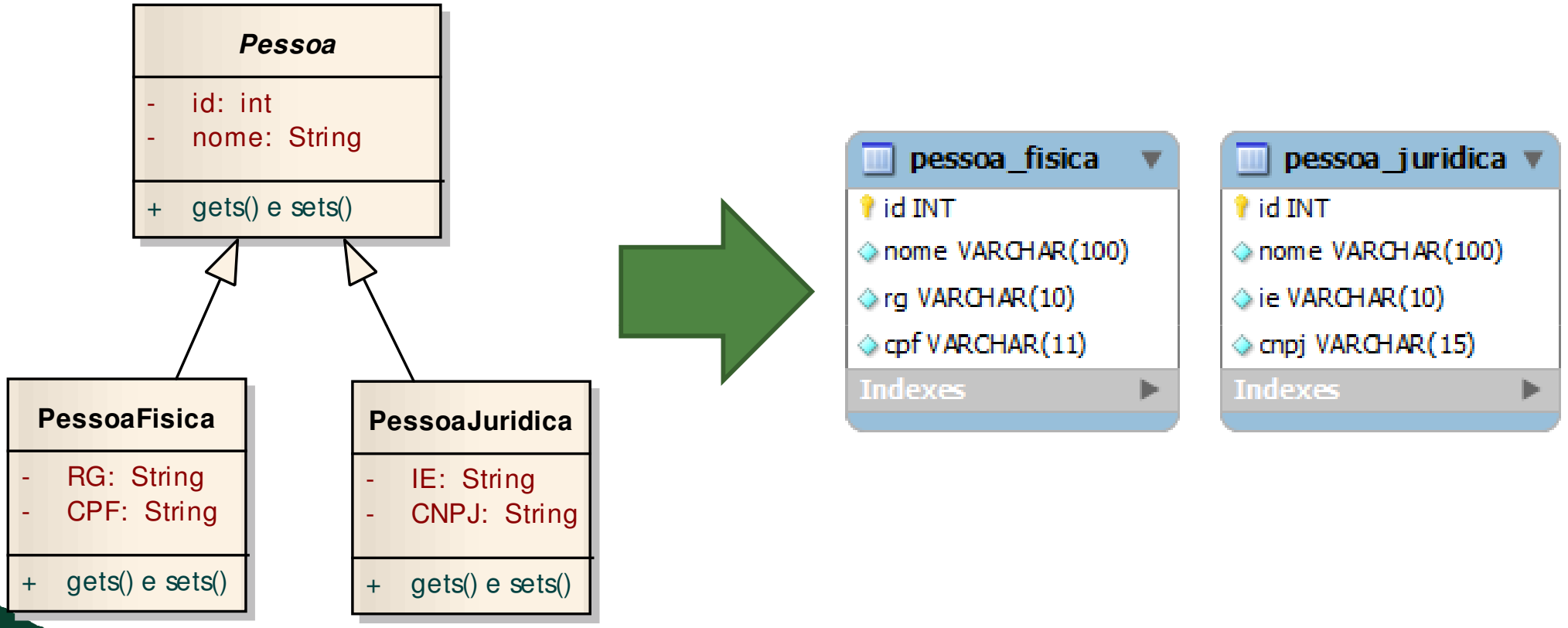
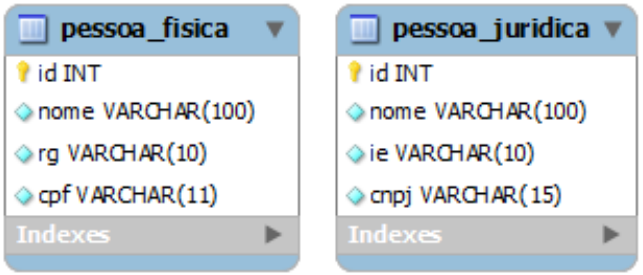


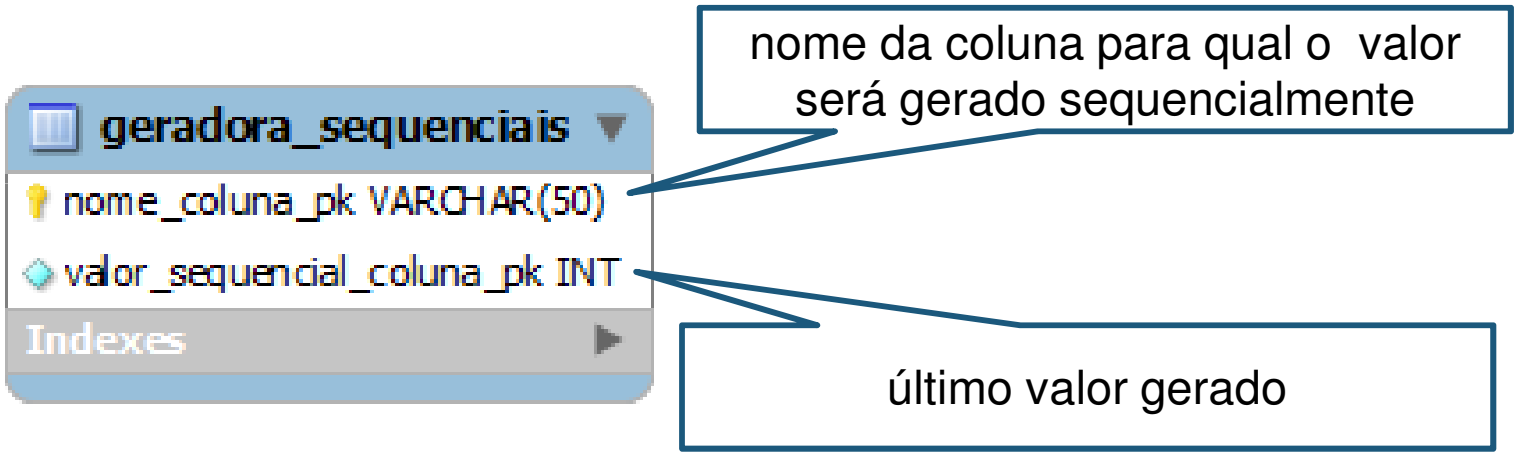
Tabela por classe concreta

Geração de PK com campo auto increment



- Cada tabela possui seu próprio campo PK:

- Como garantir **PK única** entre pessoas físicas e jurídicas?
  - auto-increment **não atende** esta necessidade
- **Soluções**
  - usar **tabela geradora de sequenciais**



  - usar **SEQUENCES** (não disponível no MySQL)

Tabela por classe concreta
Mapeamento JPA

- Mapeamento na **superclasse**

```
@Entity
@Inheritance ( strategy = InheritanceType.TABLE_PER_CLASS )
public abstract class Pessoa implements Serializable {

   @TableGenerator ( name="SEQUENCIA_PESSOA",
                     table="geradora_sequenciais",
                     pkColumnName="nome_coluna_pk",
                     valueColumnName="valor_sequencial_coluna_pk“ )

   @ Id
   @GeneratedValue ( strategy= GenerationType.TABLE,
                     generator="SEQUENCIA_PESSOA“ )
    private int id;
    @Column(name="nome")
    private String nome;
    // gets e sets...
}
```

mapeamento da tabela geradora de sequenciais

Tabela por classe concreta
Mapeamento JPA

- Mapeamento nas **subclasses**
  - Não é necessário usar anotações diferenciadas

```
@Entity
@Table(name="pessoa_fisica")
public class PessoaFisica extends Pessoa {
    @Column(name="rg")
    private String RG;
    @Column(name="cpf")
    private String CPF;

    // gets e sets..
}
```

```
@Entity
@Table(name="pessoa_juridica")
public class PessoaJuridica extends Pessoa {
    @Column(name="ie")
    private String IE;
    @Column(name="cnpj")
    private String CNPJ;

    // gets e sets..
}
```

Tabela por classe concreta
Vantagens e desvantagens

- **Vantagens**
  - permite uso da restrição **not null**

- **Desvantagens**
  - modelo relacional ainda não é normalizado
    - duplicação de colunas em pessoa física e pessoa jurídica
  - chaves primárias sequenciais requerem tratamento diferenciado
  - desempenho pior que tabela única por hierarquia de classes

Tabela por classe e subclasses

Definição

- Uma **tabela** é criada para cada classe e subclasse
- As tabelas das subclasses possuem **chave estrangeira** para a tabela da classe pai

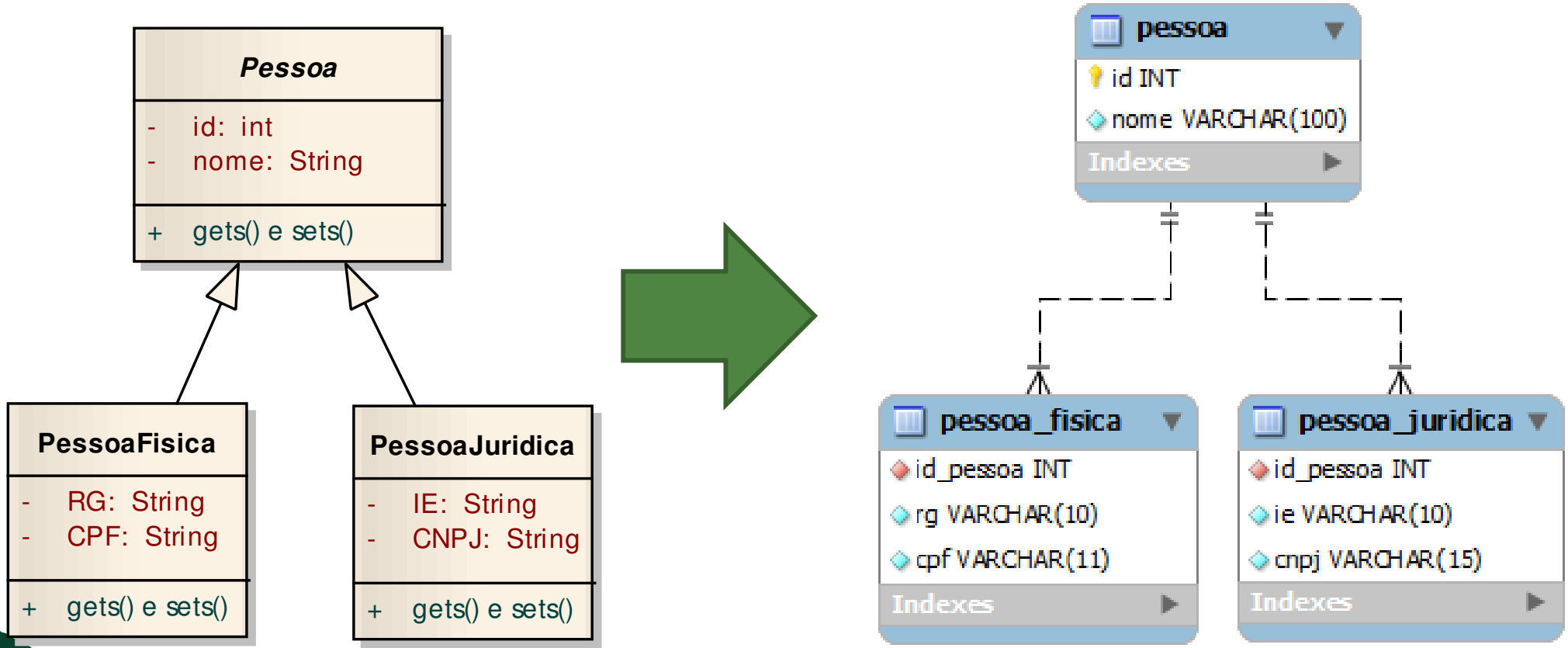


Tabela por classe e subclasses
Mapeamento JPA

- Mapeamento na **superclasse**

```
@Entity
@Inheritance ( strategy = InheritanceType.JOINED )
public abstract class Pessoa implements Serializable {
    @Id
    @GeneratedValue
    private int id;
    @Column(name="nome")
    private String nome;

    // gets e sets...

}
```

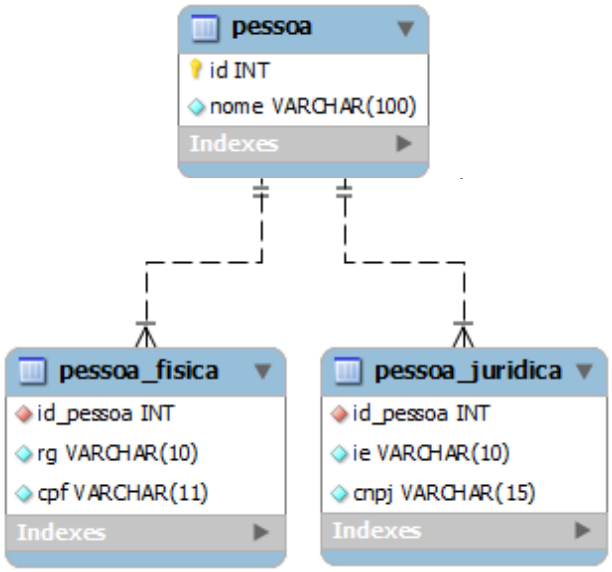


Tabela por classe e subclasses

Mapeamento JPA

- Mapeamento nas **subclasses**

```
@Entity
@Table(name="pessoa_fisica")
@PrimaryKeyJoinColumn (name="id_pessoa")
public class PessoaFisica extends Pessoa {
    @Column(name="rg")
    private String RG;
    @Column(name="cpf")
    private String CPF;

    // gets e sets...
}
```

```
@Entity
@Table(name="pessoa_juridica")
@PrimaryKeyJoinColumn (name="id_pessoa")
public class PessoaJuridica extends Pessoa {
    @Column(name="ie")
    private String IE;
    @Column(name="cnpj")
    private String CNPJ;

    // gets e sets...
}
```

Tabela por classe e subclasses

Vantagens e desvantagens

- **Vantagens**
  - modelo relacional 100% normalizado

- **Desvantagens**
  - desempenho pior que tabela única por hierarquia de classes

Mapeamento de **Herança**

Manipulações (1)

- Criar pessoa física **(1)** e jurídica **(2)** e persistir **(3)**;
  - A manipulação é simples, igual a como já se vem utilizando.

```
PessoaFisica pf = new PessoaFisica(); // (1)
pf.setNome("Homer Simpson");
pf.setRG("12345");
pf.setCPF("123456789");

PessoaJuridica pj = new PessoaJuridica(); // (2)
pj.setNome("CEAVI");
pj.setIE("13579");
pj.setCNPJ("10246812345");

em.getTransaction().begin();
em.persist(pf); // (3)
em.persist(pj);
em.getTransaction().commit();
```

Mapeamento de **Herança**

Manipulações (2)

- Consultas polimórficas
  - Buscar todas as pessoas, independente de tipo (física ou juridica)

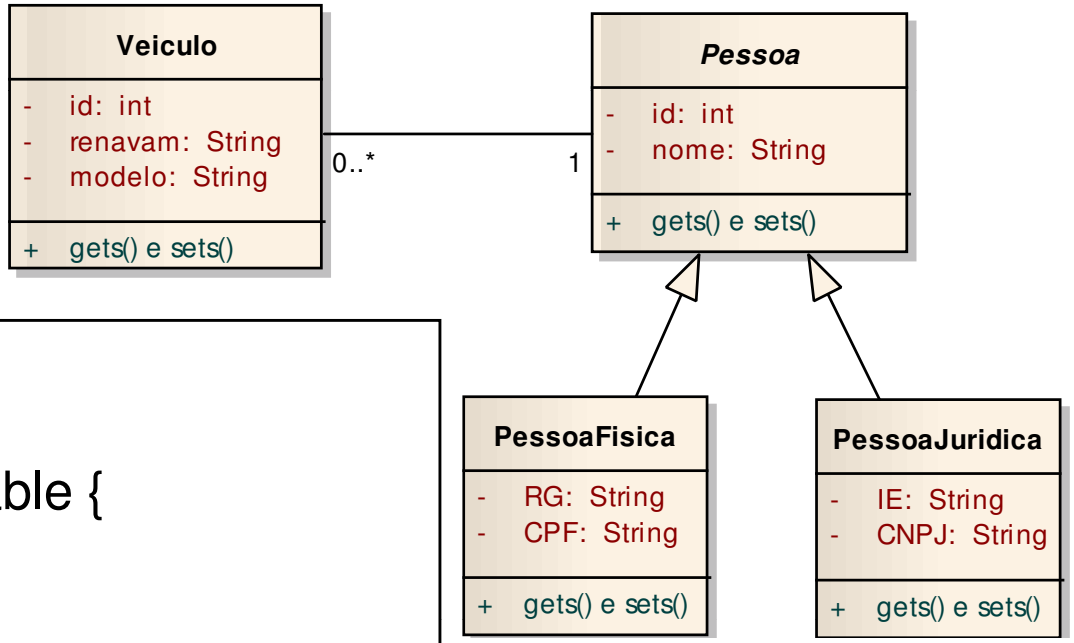
```
Query cons = em.createQuery ("select p from Pessoa p");
List<Pessoa> pessoas = cons.getResultList();
for (Pessoa umaPessoa : pessoas){
    System.out.println(umaPessoa);
 }
```

  - Buscar somente pessoas físicas
    - Fazer a consulta sobre a entidade filha

```
Query cons = em.createQuery ("select pf from PessoaFisica pf");
List<Pessoa> pessoas = cons.getResultList();
for (Pessoa umaPessoa : pessoas){
    System.out.println(umaPessoa);
 }
```

Associações polimórficas
Mapeamento

- É possível fazer associações polimórficas entre as entidades.




```
@Entity
@Table(name="veiculo")
public class Veiculo implements Serializable {
    // outros atributos...

    @ManyToOne
    @JoinColumn ( name="id_pessoa", nullable=false )
    private Pessoa proprietario;

    // gets e sets...
}
```

Associações polimórficas
Consultas

- Buscar todos os veículos
  - Os proprietários são carregados automáticamente, pelo mapeamento.

```
Query cons = em.createQuery ("select v from Veiculo v");
List<Veiculo> veiculos = cons.getResultList();
for (Veiculo umVeiculo : veiculos){
    System.out.println(umVeiculo);
 }
```

- Buscar todos os veículos em que proprietário é **Pessoa Física**

```
Query cons = em.createQuery ("select  v
                                from Veiculo v
                               where v.proprietario = PessoaFisica ");
List<Veiculo> veiculos = cons.getResultList();
for (Veiculo umVeiculo : veiculos){
    System.out.println(umVeiculo);
 }
```

Bibliografia

- BURKE, Bill; MONSON-HAEFEL, Richard. **Enterprise JavaBeans 3.0**. 5.ed. São Paulo: Prentice Hall, 2007. 538 p.