Mapeando relacionamentos entre entidades com JPA

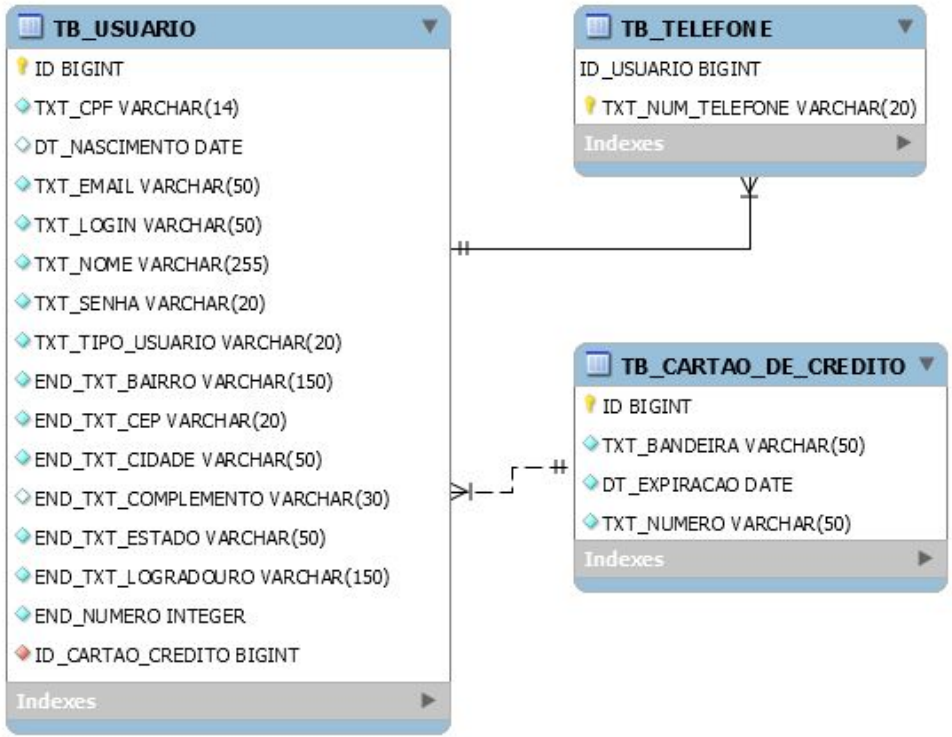
DSC AULA 3

O que veremos?

- Nesta aula e nas próximas, vamos aprender a utilizar JPA para mapear relacionamento entre entidades:
  - um para um;
  - muitos para um;
  - um para muitos;
  - muitos para muitos.
  - herança.

Relacionamento um para um

- Suponha que o usuário possuía um e somente um cartão de crédito.
- Na tabela TB_USUARIO, ID_CARTAO_CREDITO, chave estrangeira para ID de TB_CARTAO_CREDITO, é unique.



Usuario

- Observe que a entidade usuário possui um relacionamento um para um com cartão de crédito.
    - A anotação @OneToOne determina essa relação.

```
@Entity
@Table(name = "TB_USUARIO")
public class Usuario implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL, optional = false)
    @JoinColumn(name = "ID_CARTAO_CREDITO", referencedColumnName = "ID")
    private CartaoCredito cartaoCredito;
```

CartaoCredito

- Observe que adicionamos em cartão de crédito o relacionamento com a entidade Usuario.

```
@Entity
@Table(name = "TB_CARTAO_CREDITO")
public class CartaoCredito implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToOne(mappedBy = "cartaoCredito", optional = false, fetch = FetchType.LAZY)
    private Usuario usuario;
```

Usuario

- A implementação JPA não gerencia automaticamente relacionamentos bidirecionais
  - Por isso, faz-se necessário gerenciá-los automaticamente.
  - O código abaixo está na entidade Usuario.

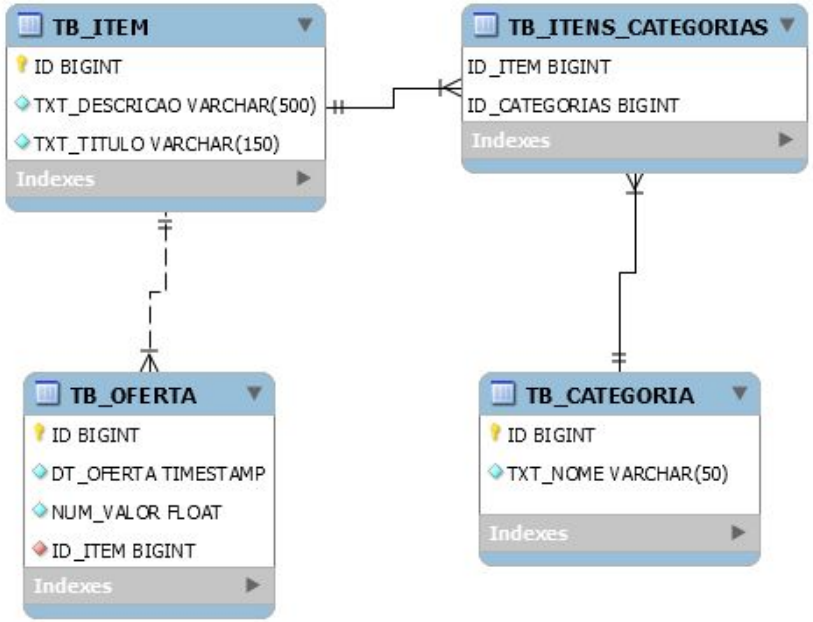
```
public void setCartaoCredito(CartaoCredito cartaoCredito) {
    this.cartaoCredito = cartaoCredito;
    this.cartaoCredito.setUsuario(this);
}
```

Para Baixar

- O código completo do exemplo 4, acesse
  - https://svn.riouxsvn.com/dsc-ifpe/exemplo_04
  - Um boa prática é ficar atento ao SQL gerado na execução do projeto

Relacionamentos muitos para um e um para muitos

- Imagine que um item possua várias ofertas, e que uma oferta se refere a apenas um único item.



Oferta

- Observe a anotação ***@ManyToOne***, representando o tipo da relação entre entidades, bem como ***@JoinColumn***, informando que ID_ITEM de TB_OFERTA é chave estrangeira de TB_ITEM.

```
@Entity
@Table(name = "TB_OFERTA")
public class Oferta implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "ID_ITEM", referencedColumnName = "ID")
    private Item item;
```

Item

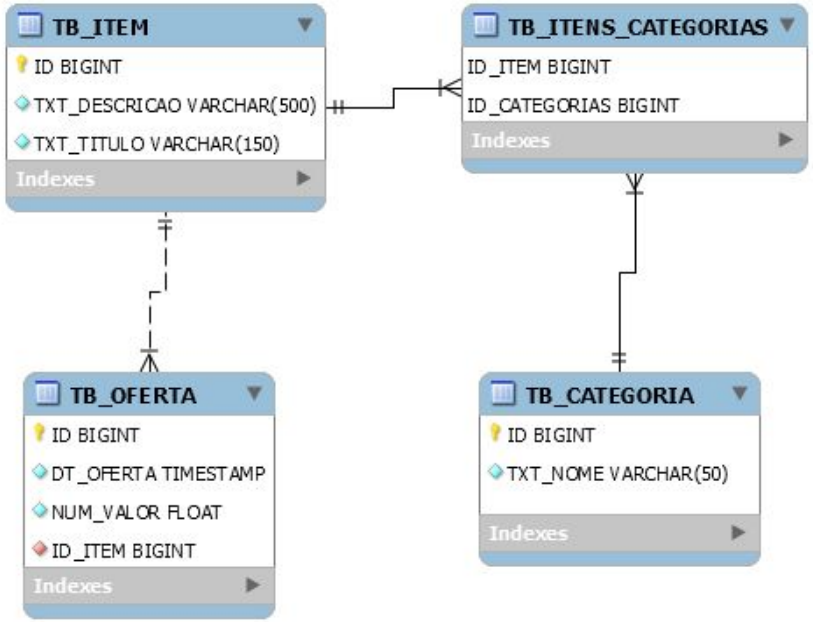
- Observe a anotação @OneToMany, com atributo mappedBy = “item”, o que significa que os detalhes do mapeamento estão do outro lado da relação.

```
@Entity
@Table(name = "TB_ITEM")
public class Item implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToMany(mappedBy = "item", fetch = FetchType.LAZY,
            cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Oferta> ofertas;
```

Relacionamentos muitos para muitos

- Imagine que um item possua várias categorias e uma categorias esteja ligada a vários itens.



Item

- O item possui uma lista de categorias, observe a anotação ***@ManyToMany***.
- É necessário informar a tabela que liga itens e categorias, no caso, TB_ITENS_CATEGORIAS. Observe também que é necessário informar como é o *join* entre essas tabelas.

```
@Entity
@Table(name = "TB_ITEM")
public class Item implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "TB_ITENS_CATEGORIAS", joinColumns = {
        @JoinColumn(name = "ID_ITEM")},
            inverseJoinColumns = {
                @JoinColumn(name = "ID_CATEGORIA")})
    private List<Categoria> categorias;
```

Categoria

- Categoria não foi mapeada com a lista de itens, mas poderia ser.

```
@Entity
@Table(name = "TB_CATEGORIA")
public class Categoria implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "TXT_NOME", length = 50, nullable = false)
    private String nome;
```

Categoria

- Caso o relacionamento seja bidirecional, o mapeamento pode ser feito como ao lado.
  - Observe que não há detalhes de campos e tabelas, porque isso já foi feito na entidade Item.

```
@Entity
@Table(name = "TB_CATEGORIA")
public class Categoria implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "TXT_NOME", length = 50, nullable = false)
    private String nome;
    @ManyToOne(fetch = FetchType.LAZY, optional = true)
    @JoinColumn(name = "ID_CATEGORIA_MAE", referencedColumnName = "ID")
    private Categoria mae;
    @ManyToMany(mappedBy = "categorias")
    private List<Item> itens;
```

Categoria

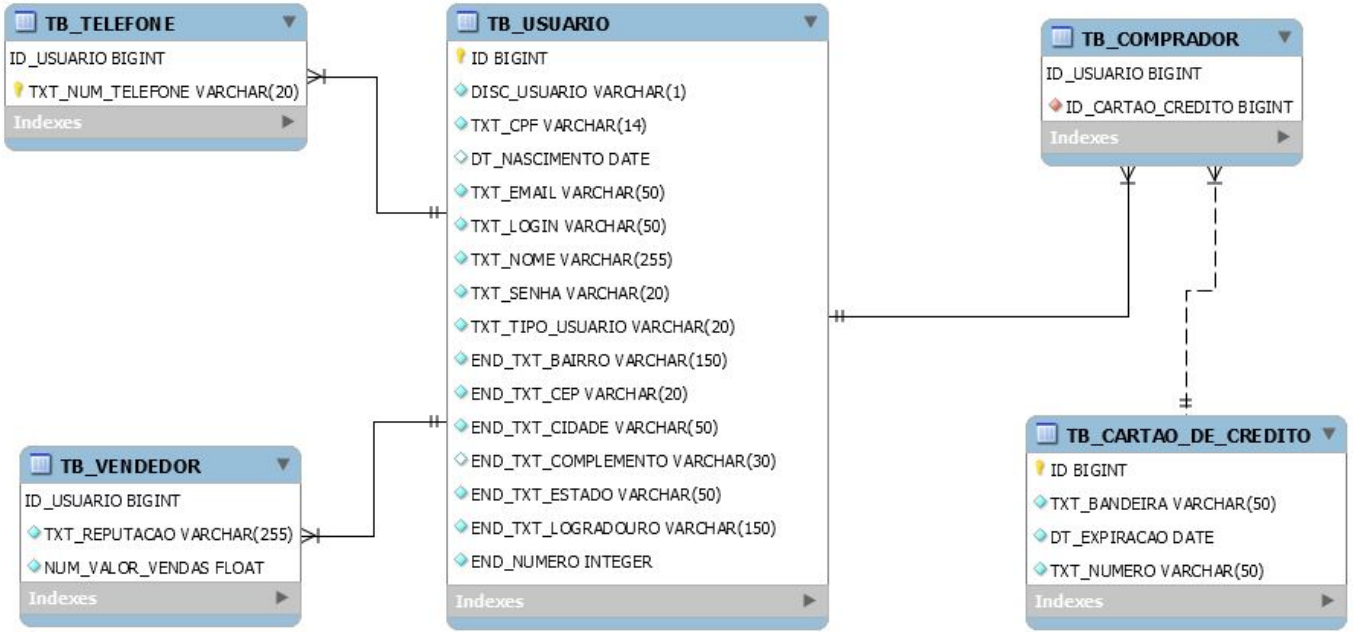
TB_CATEGORIA
ID
TXT_NOME
ID_CATEGORIA_MAE

Para Baixar

- O código completo do exemplo 5, acesse
  - https://svn.riouxsvn.com/dsc-ifpe/exemplo_05/
  - Um boa prática é ficar atento ao SQL gerado na execução do projeto

Herança

- Nosso exemplo de Herança é uma superclasse Usuário com duas subclasses Comprador e Vendedor



Usuario

- A anotação ***@Inheritance*** informa a estratégia de herança adotada. No nosso caso *JOINED*, pois as tabelas da superclasse e subclasses se relacionam através de chave estrangeira.
  - ***@DiscriminatorColumn*** é o campo da tabela da superclasse (no exemplo, TB_USUARIO) que determina a que classe da hierarquia se refere determinado registro. Mais à frente, veremos que as instâncias de Comprador terão o valor 'C', enquanto as de Vendedor, 'V'.

```
@Entity
@Table(name = "TB_USUARIO")
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "DISC_USUARIO",
        discriminatorType = DiscriminatorType.STRING, length = 1)
public abstract class Usuario implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected Long id;
```

Comprador

- Observem a anotação ***@DiscriminatorValue***, informando que todo comprador terá o valor ‘C’ no campo DISC_USUARIO da tabela TB_USUARIO
- Observem que é preciso, ainda, especificar como será o join com a chave primária da superclasse, através da anotação ***@PrimaryKeyJoinColumn***

```
@Entity
@Table(name="TB_COMPRADOR")
@DiscriminatorValue(value = "C")
@PrimaryKeyJoinColumn(name="ID_USUARIO", referencedColumnName = "ID")
public class Comprador extends Usuario implements Serializable {
    @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL, optional = false)
    @JoinColumn(name = "ID_CARTAO_CREDITO", referencedColumnName = "ID")
    private CartaoCredito cartaoCredito;
```

Vendedor

- De maneira similar, temos a subclasse Vendedor, que possui valor ‘V’ para ***@DiscriminatorValue***.

```
@Entity
@Table(name="TB_VENDEDOR")
@DiscriminatorValue(value = "V")
@PrimaryKeyJoinColumn(name="ID_USUARIO", referencedColumnName = "ID")
public class Vendedor extends Usuario implements Serializable {
    @Column(name = "NUM_VALOR_VENDAS")
    private Double valorVendas;
    @Column(name = "TXT_REPUTACAO")
    private String reputacao;
```

Para Baixar

- O código completo do exemplo 6, acesse
  - https://svn.riouxsvn.com/dsc-ifpe/exemplo_06/
  - Um boa prática é ficar atento ao SQL gerado na execução do projeto