

# Mapeando objetos com JPA

---

DSC AULA 2

# O que veremos?

- Nesta aula e nas próximas, vamos aprender a utilizar JPA para mapear objetos através de exemplos.


# TB\_USUARIO



The image shows a screenshot of a database schema viewer window titled "TB\_USUARIO". The window has a blue header bar with a small icon on the left and a dropdown arrow on the right. Below the header, the table structure is listed. The first field, "ID BIGINT", is preceded by a yellow lightbulb icon, indicating it is the primary key. The remaining seven fields are preceded by a teal diamond icon. At the bottom of the window, there is a grey bar labeled "Indexes" with a right-pointing arrow.

TB_USUARIO	
ID BIGINT	
TXT_CPF VARCHAR(14)	
DT_NASCIMENTO DATE	
TXT_EMAIL VARCHAR(50)	
TXT_LOGIN VARCHAR(50)	
TXT_NOME VARCHAR(255)	
TXT_SENHA VARCHAR(20)	
TXT_TIPO_USUARIO VARCHAR(20)	

Indexes ▶

  
`@Entity``@Table(name = "TB_USUARIO")``public class Usuario implements Serializable {` `@Id` `@GeneratedValue(strategy = GenerationType.IDENTITY)` `private Long id;` `//Observe o nome da coluna, que não é nullable, tem length 14 e é única.` `@Column(name = "TXT_CPF", nullable = false, length = 14, unique = true)` `private String cpf;` `@Column(name = "TXT_NOME", nullable = false, length = 255)` `private String nome;` `@Column(name = "TXT_LOGIN", nullable = false, length = 50)` `private String login;` `@Column(name = "TXT_EMAIL", nullable = false, length = 50)` `private String email;` `@Column(name = "TXT_SENHA", nullable = false, length = 20)` `private String senha;` `@Temporal(TemporalType.DATE)` `@Column(name = "DT_NASCIMENTO", nullable = true)` `private Date dataNascimento;` `@Transient` `private Integer idade;`

# Tipos Temporais

- Referem-se a datas e horários.
  - Atributos devem ser do tipo `java.util.Date` ou `java.util.Calendar`.
  - `TemporalType.DATE` (armazena dia, mês e ano)
  - `TemporalType.TIME` (armazena tempo)
  - `TemporalType.TIMESTAMP` (armazena dia, mês e ano, além de tempo)



## Métodos hashCode e equals

```
@Override
public int hashCode() {
    int hash = 0;
    hash += (id != null ? id.hashCode() : 0);
    return hash;
}
```

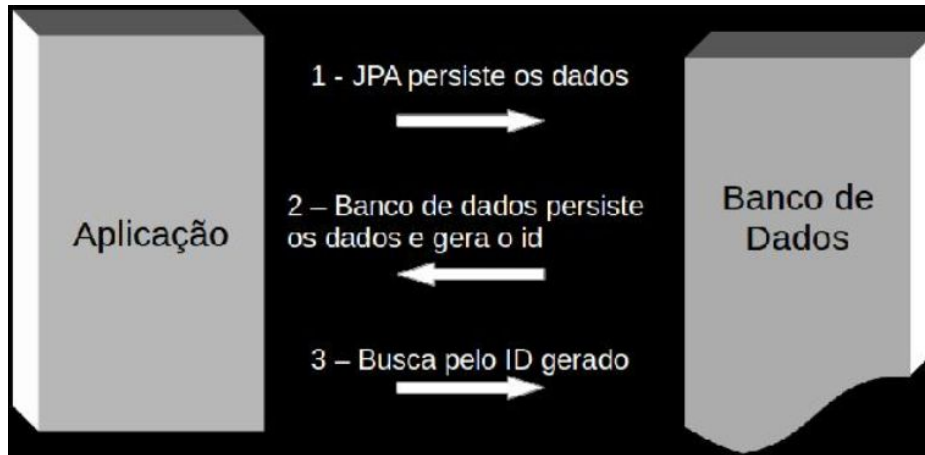
```
@Override
public boolean equals(Object object) {
    if (!(object instanceof Usuario)) {
        return false;
    }
    Usuario other = (Usuario) object;

    return !((this.id == null && other.id != null) ||
            (this.id != null && !this.id.equals(other.id)));
}
```



## Geração de Chave Primária - Identity

- O tipo de geração Identity nada mais é do que o famoso auto incremento do banco de dados. Para aqueles que trabalham com bancos como MySQL, SQLServer e Derby esse tipo de geração de id é indicado.
- Após a persistência do objeto, a implementação JPA deverá buscar pelo valor da chave primária recém-gerada.



# Para Baixar

- O código completo do exemplo 2, acesse
  - [https://svn.riouxsvn.com/dsc-ifpe/exemplo\\_02](https://svn.riouxsvn.com/dsc-ifpe/exemplo_02)



# Tipos Enumerados

- Suponha que uma enumeração que identifica os tipos de usuários da aplicação.
- Podemos representar essa enumeração no banco de dados como String ou como int.

```
package exemplo.jpa;  
  
public enum TipoUsuario {  
    VENDEDOR, COMPRADOR, ADMIN;  
}
```



# Entidade Usuario

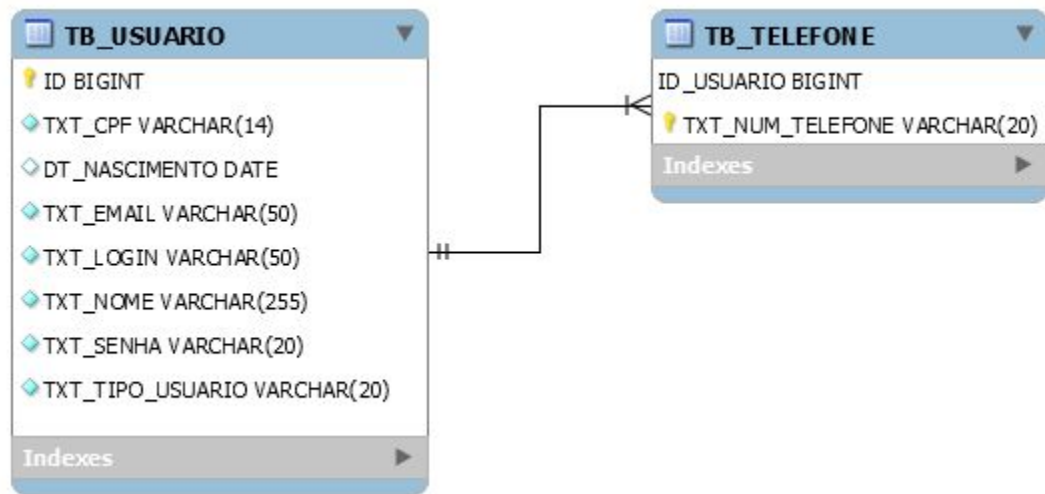
- Na entidade Usuario temos o seguinte mapeamento
- Alternativamente, utilize EnumType.ORDINAL para armazenar a enumeração como inteiro.

```
@Enumerated(EnumType.STRING)
@Column(name = "TXT_TIPO_USUARIO", nullable = false, length = 20)
private TipoUsuario tipo;
```



# Coleções

- Suponha que um usuário possui uma coleção de números de telefones, conforme pode ser visto no diagrama ER a seguir.





## Entidade Usuario

```
//O atributo será armazenado em uma tabela que representará a coleção.  
@ElementCollection  
@CollectionTable(name = "TB_TELEFONE", //nome da tabela que representa a coleção.  
                 //atributo na tabela que faz referência à chave primária de TB_USUARIO  
                 joinColumns = @JoinColumn(name = "ID_USUARIO", nullable = false))  
@Column(name = "TXT_NUM_TELEFONE", nullable = false, length = 20)  
private Collection<String> telefones;
```

# Entidade Usuario

- Observe que é utilizado um método para adicionar os telefones.
  - É necessário gerenciar as coleções, a implementação JPA não fará isso por nós.

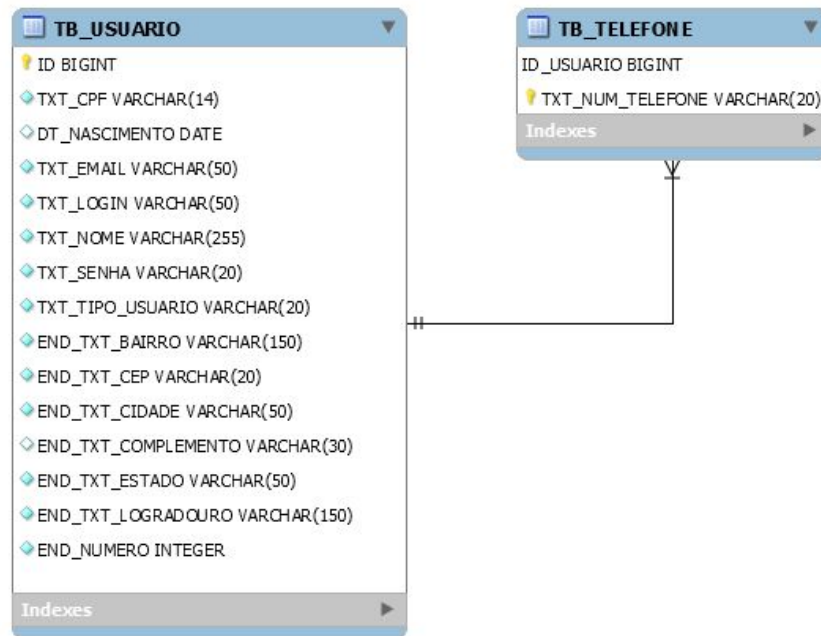
```
public Collection<String> getTelefones() {  
    return telefones;  
}
```

```
public void addTelefone(String telefone) {  
    if (telefones == null) {  
        telefones = new HashSet<>();  
    }  
    telefones.add(telefone);  
}
```



# Mapeando Mais de Uma Classe na Mesma Tabela

- Imagine agora que TB\_USUARIO contenha também dados de endereço, conforme pode ser visto no diagrama ER a seguir





# Mapeando Mais de Uma Classe na Mesma Tabela

```
/**
 * Todos os campos de Endereco serão armazenados na mesma tabela
 * que armazena os dados de Usuario.
 */
```

```
@Embeddable
```

```
public class Endereco implements Serializable {
```

```
    @Column(name = "END_TXT_LOGRADOURO")
```

```
    private String logradouro;
```

```
    @Column(name = "END_TXT_BAIRRO")
```

```
    private String bairro;
```

```
    @Column(name = "END_NUMERO")
```

```
    private Integer numero;
```

```
    @Column(name = "END_TXT_COMPLEMENTO")
```

```
    private String complemento;
```

```
    @Column(name = "END_TXT_CEP")
```

```
    private String cep;
```

```
    @Column(name = "END_TXT_CIDADE")
```

```
    private String cidade;
```

```
    @Column(name = "END_TXT_ESTADO")
```

```
    private String estado;
```

```
@Entity
```

```
@Table(name = "TB_USUARIO")
```

```
@Access(AccessType.FIELD)
```

```
public class Usuario implements Serializable {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    @Embedded
```

```
    private Endereco endereco;
```

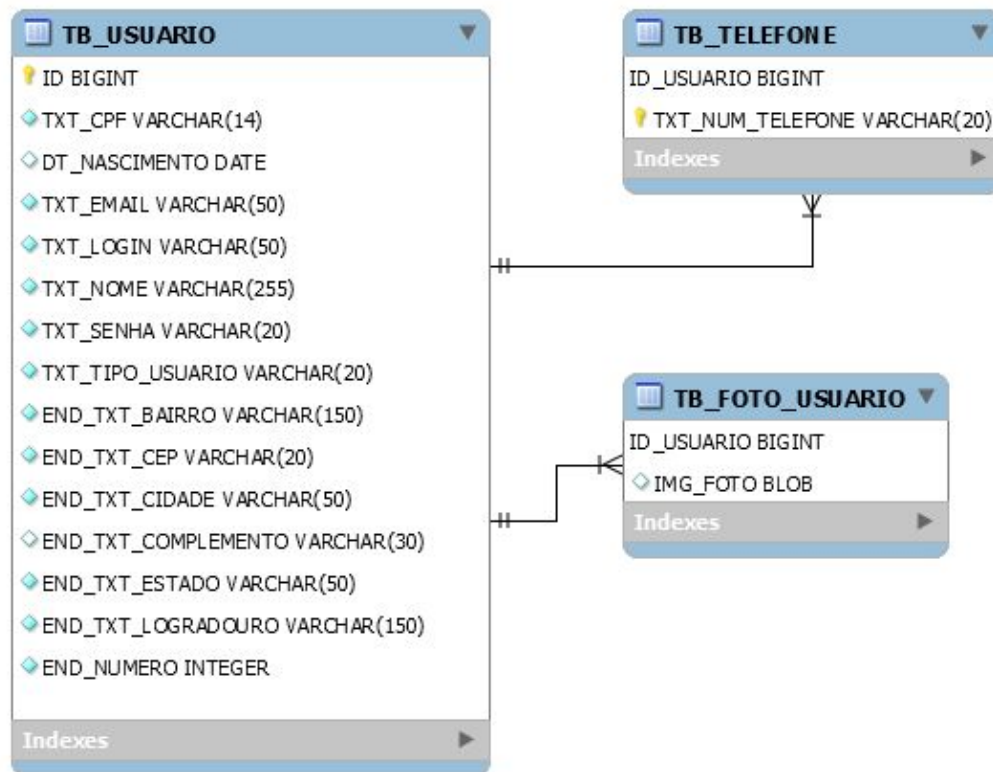
# Mapeando Mais de Uma Tabela na Mesma Classe

- Em sistemas legados é comum encontrar situações em que duas ou mais tabelas possam representar uma entidade.
- Pode ser ainda que seja tomada a decisão de armazenar uma imagem em uma tabela à parte, por questões de desempenho (carregar sempre a imagem pode ser custoso e desnecessário).





# Mapeando Mais de Uma Tabela na Mesma Classe





# Mapeando Mais de Uma Tabela na Mesma Classe

```
@Entity
@Table(name = "TB_USUARIO")
@SecondaryTable(name = "TB_FOTO_USUARIO",
    pkJoinColumns = {
        @PrimaryKeyJoinColumn(name = "ID_USUARIO")}
)
@Access(AccessType.FIELD)
public class Usuario implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Basic(fetch = FetchType.LAZY)
    @Column(name = "IMG_FOTO", table = "TB_FOTO_USUARIO", nullable = true)
    private byte[] foto;
```



## Classe TesteJPA

A classe TesteJPA agora realiza a persistência e a consulta do usuário persistido.

```
public static void main(String[] args) throws IOException {  
    persistirUsuario();  
    consultarUsuario(1L);  
}
```



## consultarUsuario

```
private static void consultarUsuario(long l) {  
    EntityManager em = emf.createEntityManager();  
  
    //A primeira consulta vai recuperar apenas os dados de TB_USUARIO, excetuando IMG_FOTO.  
    Usuario usuario = em.find(Usuario.class, l);  
    System.out.println(usuario.getNome());  
    //usuario.getTelefones().iterator() vai provocar uma consulta à tabela TB_TELEFONE.  
    System.out.println(usuario.getTelefones().iterator().next());  
    //usuario.getFoto().length vai as informações de TB_USUARIO, incluindo IMG_FOTO.  
    System.out.println(usuario.getFoto().length);  
    em.close();  
}
```



## consultarUsuario

a execução de **usuario = em.find(Usuario.class, l)** gera a seguinte query SQL, sem carregar os dados da foto e dos telefones:

```
SELECT
  t0.ID, t0.TXT_CPF, t0.DT_NASCIMENTO, t0.TXT_EMAIL,
  t0.TXT_LOGIN, t0.TXT_NOME, t0.TXT_SENHA,
  t0.TXT_TIPO_USUARIO, t0.END_TXT_BAIRRO,
  t0.END_TXT_CEP, t0.END_TXT_CIDADE,
  t0.END_TXT_COMPLEMENTO, t0.END_TXT_ESTADO,
  t0.END_TXT_LOGRADOURO, t0.END_NUMERO
FROM
  TB_USUARIO t0, TB_FOTO_USUARIO t1
WHERE
  ((t0.ID = ?) AND (t1.ID_USUARIO = t0.ID))
```

## consultarUsuario

```
@Basic(fetch = FetchType.LAZY)
@Column(name = "IMG_FOTO", table = "TB_FOTO_USUARIO", nullable = true)
private byte[] foto;
```

Lembrem-se do mapeamento acima. Ele diz que o atributo foto só deve ser carregado se necessário (**@Basic(fetch = FetchType.LAZY)**). Logo, no momento de selecionar o Usuario, a foto não é carregada.

## consultarUsuario

```
@ElementCollection
@CollectionTable(name = "TB_TELEFONE",
    joinColumns = @JoinColumn(name = "ID_USUARIO", nullable = false))
@Column(name = "TXT_NUM_TELEFONE", nullable = false, length = 20)
private Collection<String> telefones;
```

Lembrem-se do mapeamento acima. Ele diz que o atributo telefones só deve ser carregado se necessário. Logo, no momento de selecionar o Usuario, os telefones não serão carregados. Este é o comportamento padrão para **@ElementCollection**.



## consultarUsuario

a execução de **usuario.getTelefones().iterator().next()** gera a seguinte query SQL:

```
SELECT t0.TXT_NUM_TELEFONE FROM TB_TELEFONE t0 WHERE  
(t0.ID_USUARIO = ?)
```





## consultarUsuario

a execução de **usuario.getFoto().length** gera a seguinte query SQL:

```
SELECT
  t0.ID, t1.ID_USUARIO, t0.TXT_CPF, t0.DT_NASCIMENTO,
  t0.TXT_EMAIL, t1.IMG_FOTO, t0.TXT_LOGIN, t0.TXT_NOME,
  t0.TXT_SENHA, t0.TXT_TIPO_USUARIO, t0.END_TXT_BAIRRO,
  t0.END_TXT_CEP, t0.END_TXT_CIDADE,
  t0.END_TXT_COMPLEMENTO, t0.END_TXT_ESTADO,
  t0.END_TXT_LOGRADOURO, t0.END_NUMERO
FROM
  TB_USUARIO t0, TB_FOTO_USUARIO t1
WHERE
  ((t0.ID = ?) AND (t1.ID_USUARIO = t0.ID))
```

## Altere o código e verifique a mudança no SQL gerado

```
@Basic(fetch = FetchType.EAGER)
@Column(name = "IMG_FOTO", table = "TB_FOTO_USUARIO", nullable = true)
private byte[] foto;
@Embedded
```

***FetchType.EAGER*** vai fazer com que a foto seja carregada sempre que o usuário for carregado. Experimente tirar também o ***@Basic*** e veja que o comportamento será o mesmo.

## Altere o código e verifique a mudança no SQL gerado

```
@ElementCollection(fetch = FetchType.EAGER)
@CollectionTable(name = "TB_TELEFONE",
    joinColumns = @JoinColumn(name = "ID_USUARIO", nullable = false))
@Column(name = "TXT_NUM_TELEFONE", nullable = false, length = 20)
private Collection<String> telefones;
```

***@ElementCollection(fetch = FetchType.EAGER)*** vai fazer com que os telefones sejam carregados sempre que o usuário for carregado.

# Para Baixar

- O código completo do exemplo 3, acesse
  - [https://svn.riouxsvn.com/dsc-ifpe/exemplo\\_03](https://svn.riouxsvn.com/dsc-ifpe/exemplo_03)
  - Verifique o arquivo persistence.xml

# Configure o projeto exemplo\_03 no Netbeans

- Para o carregamento preguiçoso funcionar, é necessário, em **Project Properties**, ir na categoria **Run** e adicionar o jar do eclipselink, em **VM Options**.
- Além disso, configurar, em **Working Directory**, o caminho do jar do eclipselink (no caso pode ser o caminho onde o maven armazena o arquivo).
- -javaagent:eclipselink-4.0.4.jar

