

PROGRAMAÇÃO PARA WEB I

JPA

Profa. Silvia Bertagnolli

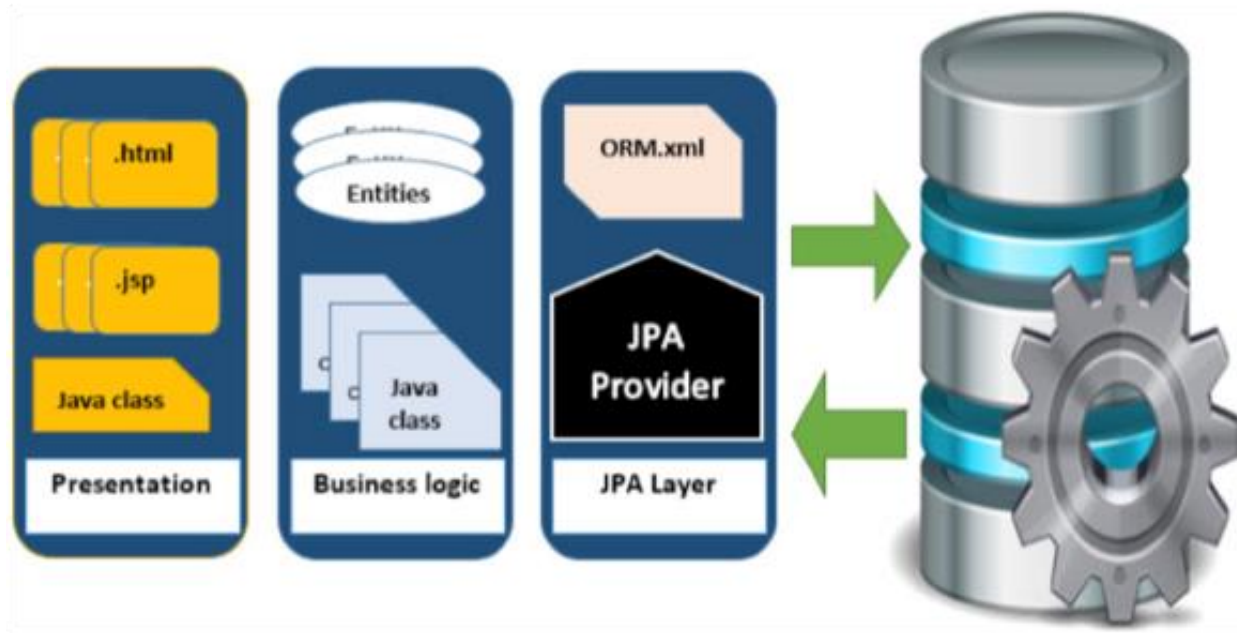
JPA x CAMADAS

PADRÃO DAO

Assim como no JDBC podemos usar o padrão DAO com o JPA

Toda a lógica de persistência ficará em uma classe DAO que é responsável por chamar os métodos `persist`, `remove`, `find`, entre outros da classe `EntityManager`

MVC COM JPA



EXEMPLO - PACKAGE DAO

PASSO 1 – CRIE A ENTIDADE

```
@Entity
public class Usuario implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idUsuario;

    //....
```

PASSO 2 – CRIE A CLASSE DAO

O padrão de projeto DAO tem como objetivo evitar a exposição da camada de persistência para outras camadas.

O DAO isola as operações com o banco, como criação, recuperação, atualização e exclusão, para um objeto de negócio

USUARIODAO - SALVAR

persist() é equivalente
ao INSERT INTO

```
public boolean salvar(Usuario user) {  
    try {  
        em = JPAUtil.getEntityManager();  
        em.getTransaction().begin();  
        em.persist(user);  
        em.getTransaction().commit();  
        return true;  
    } catch (RuntimeException e) {  
        if (em.getTransaction().isActive()) {  
            em.getTransaction().rollback();  
        }  
        return false;  
    }  
}
```


USUARIODAO - ATUALIZAR

merge() é equivalente
ao UPDATE

```
public boolean atualizar(Usuario user) {  
    try {  
        em = JPAUtil.getEntityManager();  
        em.getTransaction().begin();  
        em.merge(user);  
        em.getTransaction().commit();  
        return true;  
    } catch (RuntimeException e) {  
        if (em.getTransaction().isActive()) {  
            em.getTransaction().rollback();  
        }  
        return false;  
    }  
}
```

USUARIODAO- BUSCAR (POR ID)

find() é equivalente ao
SELECT FROM WHERE ID=

```
public Usuario buscarID(int id) {  
    try {  
        em = JPAUtil.getEntityManager();  
        Usuario usuario = em.find(Usuario.class, id);  
        return usuario;  
    } catch (RuntimeException e) {  
        if (em.getTransaction().isActive()) {  
            em.getTransaction().rollback();  
        }  
        return null;  
    }  
}
```

USUARIODAO - REMOVER

remove() é equivalente
ao DELETE

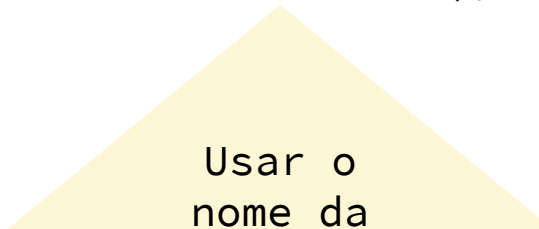
```
public boolean remover(long id) {  
    try {  
        em = JPAUtil.getEntityManager();  
        em.getTransaction().begin();  
        Usuario entity = em.find(Usuario.class, id);  
        em.remove(entity);  
        em.getTransaction().commit();  
        return true;  
    } catch (RuntimeException e) {  
        if (em.getTransaction().isActive()) {  
            em.getTransaction().rollback();  
        }  
        return false;  
    }  
}
```

USUARIODAO - BUSCAR TODOS

```
public List<Usuario> buscarTodos() {  
    try {  
        em = JPAUtil.getEntityManager();  
        TypedQuery<Usuario> query = em.createQuery("SELECT obj FROM Usuario obj",  
                                                    Usuario.class);  
  
        List<Usuario> usuarios = query.getResultList();  
        return usuarios;  
    } catch (RuntimeException e) {  
        return null;  
    }  
}
```

USUARIODAO - BUSCAR TODOS

```
public List<Usuario> buscarTodos() {  
    try {  
        em = JPAUtil.getEntityManager();  
        TypedQuery<Usuario> query = em.createQuery("SELECT obj FROM Usuario obj",  
                                                    Usuario.class);  
  
        List<Usuario> usuarios = query.getResultList();  
        return usuarios;  
    } catch (RuntimeException e) {  
        return null;  
    }  
}
```



Usar o
nome da
classe no
SQL -
**Case
sensitive**

TESTEDA0

```
public static void main(String[] args) {  
    UsuarioDAO objDAO = new UsuarioDAO();  
  
    // Cria uma nova instância de usuário e salva  
    Set<String> emails = new HashSet<String>();  
    emails.add("fulano1@mail.com");  
    emails.add("fulano2@mail.com");  
    Usuario user = new Usuario("fulano", "123456", emails);  
    user.setPerfil(Perfil.ALUNO);  
    user.setDataCadastro(new Date());  
  
    if (objDAO.salvar(user))  
        System.out.print("Usuário Fulano foi salvo!!!");  
}
```

TESTEDA01

```
System.out.println("\nLISTAR TODOS");  
for (Usuario u : objDAO.buscarTodos())  
System.out.printf(p.toString());
```

```
// Recupera o usuário e atualiza com novo identificador  
user.setIdentificador("Beltrano");  
if (objDAO.atualizar(user))  
System.out.println("Usuário Atualizado!!");
```

TESTE DAO1

```
System.out.println("\nLISTAR TODOS");  
for (Usuario p : objDAO.buscarTodos())  
    System.out.printf(p.toString());  
  
if (objDAO.remover(user.getIdUsuario()))  
    System.out.printf("Usuário excluído: [ID=%d] = %s\n",  
        user.getIdUsuario(),  
        "Coleção vazia? " + (objDAO.buscarTodos().isEmpty()));
```


RELACIONAMENTOS

JPA: RELACIONAMENTOS MAPEADOS

@Inheritance

@OneToOne

@OneToMany

@ManyToOne

@ManyToMany

HERANÇA

JPA: HERANÇA

@Inheritance

Estratégias:

- Uma única tabela - `InheritanceType.SINGLE_TABLE`
- Uma tabela para cada classe (JOINED) - `InheritanceType.JOINED`
- Uma tabela por classe concreta -
`InheritanceType.TABLE_PER_CLASS`

Ou ainda -> Usar herança de Mapeamento com: `MappedSuperClass`

SINGLE_TABLE

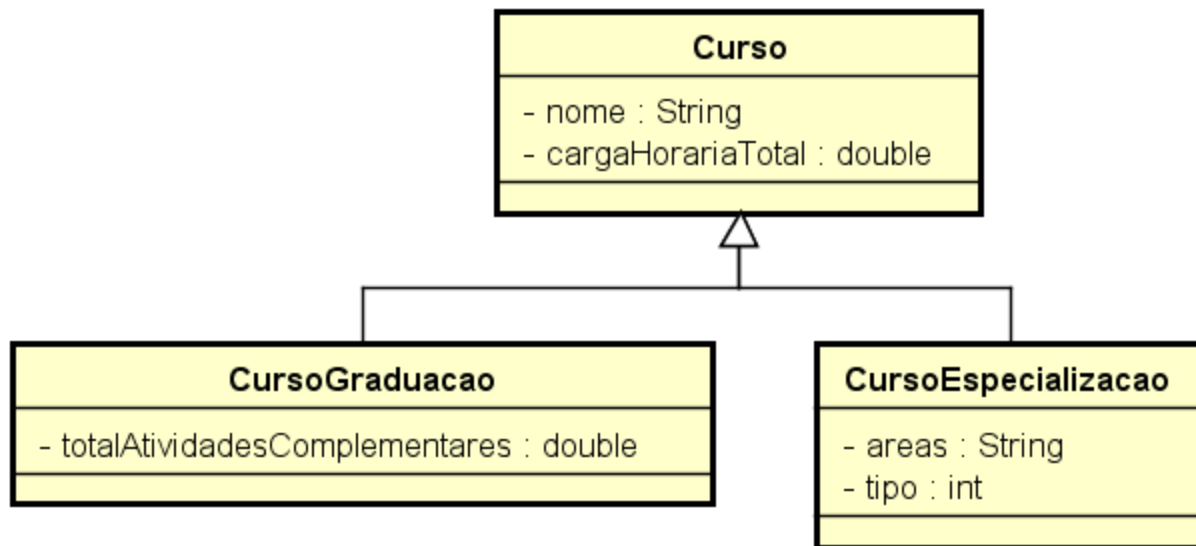
SINGLE_TABLE

Todos os atributos de todas as classes serão persistidos em uma única tabela

Os objetos das subclasses serão diferenciados uns dos outros usando um atributo designador, que é uma coluna pré-determinada mapeada com uma anotação

Estratégia de mapeamento mais rápida e simples, porém a desvantagem é que as colunas das propriedades declaradas nas subclasses precisam aceitar valores nulos

EXEMPLO SINGLE_TABLE



SINGLE_TABLE: CURSO

```
@Entity
@Inheritance (strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "tipo", length = 3,
    discriminatorType = DiscriminatorType.STRING)
public class Curso implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long idCurso;
    @Column(name = "tipo", updatable=false)
    private String tipo;
    private String nome;
    private double cargaHorariaTotal;
```


SINGLE_TABLE: CURSO

@Entity

@Inheritance (strategy = InheritanceType.SINGLE_TABLE)

**@DiscriminatorColumn(name = "tipo", length = 3,
discriminatorType = DiscriminatorType.STRING)**

public class Curso implements Serializable {

@DiscriminatorColumn

informa qual o nome da
coluna que armazenará a
entidade “dona” de uma
determinada linha no BD

SINGLE_TABLE: CURSOGRADUACAO

@Entity

@DiscriminatorValue(value = "CG")

```
public class CursoGraduacao extends Curso implements  
Serializable {
```

@DiscriminatorValue
identifica cada classe
na tabela do BD
CursoGraduacao será
identificado por CG

TESTE

```
CursoGraduacao c1 = new CursoGraduacao();  
c1. setTotalAtividadesComplementares(67.44);  
c1.setCargaHorariaTotal(1000.0);  
c1.setNome("Curso XXX");  
new CursoGraduacaoDAO().salvar(c1);
```

NO BD (SINGLE_TABLE)...

```
new CursoGraduacaoDAO().salvar(c1);
```

@Entity

@DiscriminatorValue(value = "CG")

public class CursoGraduacao extends Curso
implements Serializable

+ Opções

	IDCURSO	tipo	CARGAHORARIATOTAL	NOME	TOTALATIVIDADESCOMPLEMENTARES	AREA	CLASSIFICACAO
<input type="checkbox"/> Editar Copiar Apagar	1	CG	1000	Curso XXX	67.44	NULL	NULL

Tabela Curso: área e classificação null,
porque esse é um curso de graduação

SINGLE_TABLE: VANTAGENS

Dados centralizados – em uma única tabela

Fácil de entender – facilita a extração de dados via SQL

Bom desempenho – a consulta é realizada em uma única tabela

SINGLE_TABLE: DESVANTAGENS

A tabela de mapeamento é criada uma coluna para cada campo das classes que fazem parte da hierarquia. Assim, quanto mais atributos maior será a tabela mapeada, e poderão aparecer várias colunas vazias/nulas

Se uma entidade do tipo da subclasse não pode ter campos nulos uma mensagem do BD será exibida

JOINED

JOINED

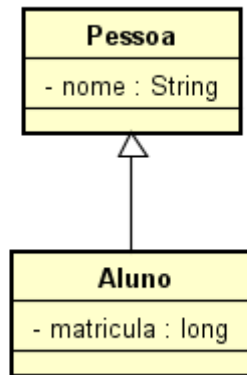
Cria uma tabela diferente para cada classe da hierarquia

Cada tabela só possui o estado declarado na própria classe

Cada classe da hierarquia é representada por uma tabela correspondente

Cada tabela que representa uma subclasse contém apenas os campos definidos na subclasse (não contém os herdados das superclasses) e colunas chaves primárias que servem como chaves estrangeiras para as chaves primárias da tabela da superclasse

EXEMPLO JOINED



JOINED: PESSOA

@Entity

@Inheritance (strategy = InheritanceType.JOINED)

```
public class Pessoa implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Column(name="idPessoa")  
    private Long idPessoa;  
    private String nome;  
  
    public Pessoa() {}  
    public Pessoa(Long id, String nome) {  
        this.id = id;  
        this.nome = nome;  
    }  
}
```

JOINED: ALUNO

@Entity

@PrimaryKeyJoinColumn(name = "idAluno", referencedColumnName = "idPessoa")

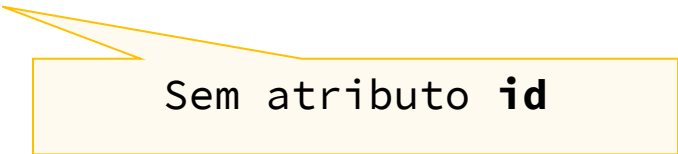
public class Aluno extends Pessoa implements Serializable {

private static final long serialVersionUID = 1L;

private long matricula;

public long getMatricula() {
 return matricula;
}

public void setMatricula(long matricula) {
 this.matricula = matricula;
}



Sem atributo **id**

NO BD (JOINED)...

+ Opções

	idAluno	MATRICULA
<input type="checkbox"/> Editar Copiar Apagar	1	1234

Tabela Aluno

+ Opções

	idPessoa	DTYPE	NOME
<input type="checkbox"/> Editar Copiar Apagar	1	Aluno	Fulano

Tabela Pessoa

JOINED: VANTAGENS

Uma tabela por entidade

Comparando a hierarquia de classe ao relacionamento entre as tabelas geradas, observa-se que essa estratégia é a mais próxima do modelo de classes que a mapeia – segue o modelo OO

JOINED: DESVANTAGENS

Insert mais custoso, é necessário realizar o insert em duas tabelas e não em uma única

Observe que para recuperar o estado persistente, deve-se realizar um ou mais Joins

É o mais lento, mas resulta no esquema de base de dados mais normalizado de todos os modelos de mapeamento de herança

TABLE_PER_CLASS

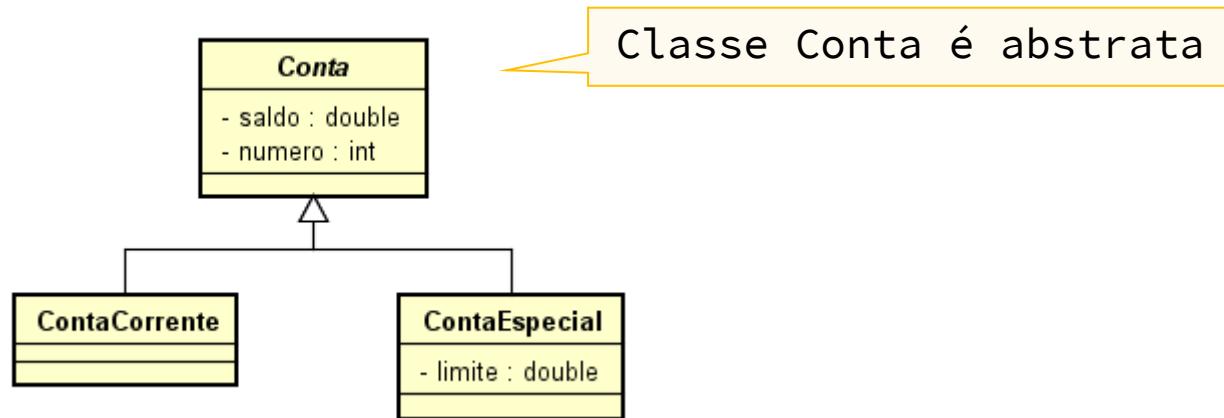
TABLE_PER_CLASS

Cria uma tabela diferente para cada classe concreta da hierarquia, como na estratégia joined

Os atributos da classe abstrata fazem parte da tabela correspondente à classe concreta

Observe que cada tabela inclui **todo** o estado de uma instância da classe correspondente - todas as propriedades das instâncias de uma classe, incluindo as herdadas, são mapeadas para colunas da tabela correspondente à classe

EXEMPLO TABLE__PER__CLASS



TABLE_PER_CLASS: CONTA

@Entity

@Inheritance (strategy = InheritanceType.TABLE_PER_CLASS)

public abstract class Conta implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

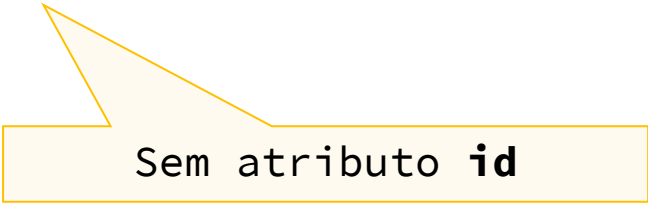
private Long idConta;

private double saldo;

private int numero;

TABLE__PER__CLASS: CONTA CORRENTE

```
@Entity  
public class ContaCorrente extends Conta implements Serializable  
{  
  
    private static final long serialVersionUID = 1L;  
  
}
```

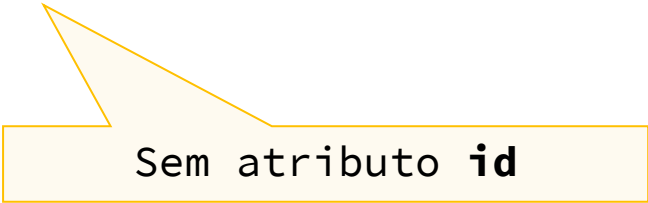


Sem atributo **id**

TABLE__PER__CLASS: CONTAESPECIAL

@Entity




```
public class ContaEspecial extends Conta implements Serializable {  
    private static final long serialVersionUID = 1L;  
    private double limite;  
    //...  
}
```



Sem atributo **id**




NO BD (TABLE_PER_CLASS)...

+ Opções

	IDCONTA	NUMERO	SALDO
<input type="checkbox"/>  Editar  Copiar  Apagar	1	123	200

```
ContaCorrente c1 = new ContaCorrente();  
c1.setNumero(123);  
c1.setSaldo(200.0);
```

+ Opções

	IDCONTA	LIMITE	NUMERO	SALDO
<input type="checkbox"/>  Editar  Copiar  Apagar	1	300	123	200

```
ContaEspecial c2 = new ContaEspecial();  
c2.setNumero(123);  
c2.setSaldo(200.0);  
c2.setLimite(300.0);
```

```
SELECT * FROM `conta`
```

IDCONTA	NUMERO	SALDO
---------	--------	-------

Tabela Conta está vazia,
pois foram salvos objetos
do tipo ContaCorrente e
ContaEspecial - classe
abstrata não pode ser
instanciada

TABLE__PER__CLASS

Vantagem: Essa estratégia é muito eficiente para todas as operações sobre instâncias de classes conhecidas

Desvantagem: Colunas repetidas nas classes filhas

@MAPPEDSUPERCLASS

@MappedSuperclass

Com essa estratégia, herda-se o mapeamento de colunas de uma superclasse anotada com **@MappedSuperclass** que **não é entidade**, ou seja, não tem a anotação **@Entity** ou **@Table**

Essa classe não tem uma relação direta com uma tabela, logo não pode ser usada em consultas

Boa prática: deixar a classe com **@MappedSuperclass** definida como abstrata

@MappedClass: CURSO

@MappedSuperclass

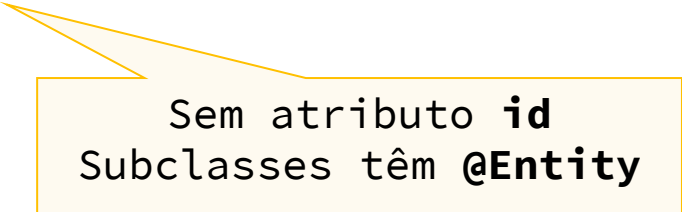
```
public abstract class Curso implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    @Column(name = "id_curso")  
    private Long id;  
    private String nome;  
    private double cargaHorariaTotal;  
}
```

@MappedSuperclass: CursoPosGraduacao

@Entity

```
public class CursoPosGraduacao extends Curso implements  
Serializable {
```

```
    private static final long serialVersionUID = 1L;  
    private String area;  
    private String classificacao;
```



Sem atributo **id**
Subclasses têm **@Entity**

EXERCÍCIOS

**Fazer os
exercícios 7 a 9
da lista
disponível no
Moodle!**
