

# ECM251 - Linguagens I

Teoria - Encapsulamento e Construtores

Prof. Murilo Zanini de Carvalho

Prof. Tiago Sanches da Silva

Antes de começar!

# Clone seu repositório do Github

- Lembre-se sempre antes de iniciar uma aula, clonar seu repositório remoto e realizar as atividades nele.
- Para cada atividade desenvolvida, criar um novo diretório.



# GitHub

Retirado de  
([https://miro.medium.com/max/4000/0\\*MZMI76wKo2FQLqG0.png](https://miro.medium.com/max/4000/0*MZMI76wKo2FQLqG0.png)), em 07/03/2021

# Modificadores de Acesso

# Modificadores de Acesso

A nossa classe Conta ainda possui algumas fragilidades. E devemos cuidar para que qualquer programa, objeto ou módulo que for utilizá-la não consiga quebrar a integridade semântica que definimos.

Como criadores da classe Conta, devemos fornecer os meios para que utilizem todas nossas funcionalidade, mas somos responsáveis caso seja possível fazer mal uso de nossa classe. Acredito que nenhum banco iria querer utilizar está classe.

Como podemos resolver isso?

# Modificadores de Acesso

Não vamos mais deixar que os “outros” mexam no saldo, somente a classe Conta, poderá mexer!

```
Conta c1 = new Conta();
```

```
c1.saldo = 1000;
```

```
c1.sacar( 200 );
```

```
c1.saldo = 1000;
```



**problem?**

# Modificadores de Acesso

Os modificadores de acesso são padrões de visibilidade de acessos às classes, atributos e métodos. Esses modificadores são palavras-chaves reservadas pelo Java, ou seja, palavras reservadas não podem ser usadas como nome de métodos, classes ou atributos.

**public**  
**private**  
**protected**

**public** Uma declaração com o modificador public pode ser acessada de qualquer lugar e por qualquer entidade que possa visualizar a classe a que ela pertence.

**private** Os membros da classe definidos como não podem ser acessados ou usados por nenhuma outra classe. Esse modificador não se aplica às classes, somente para seus métodos e atributos. Esses atributos e métodos também não podem ser visualizados pelas classes herdadas.

**protected** O modificador protected torna o membro acessível às classes do mesmo pacote ou através de herança, seus membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados.



# Modificadores de Acesso

## **default (padrão):**

A classe e/ou seus membros são acessíveis somente por classes do mesmo pacote, na sua declaração não é definido nenhum tipo de modificador, sendo este identificado pelo compilador.

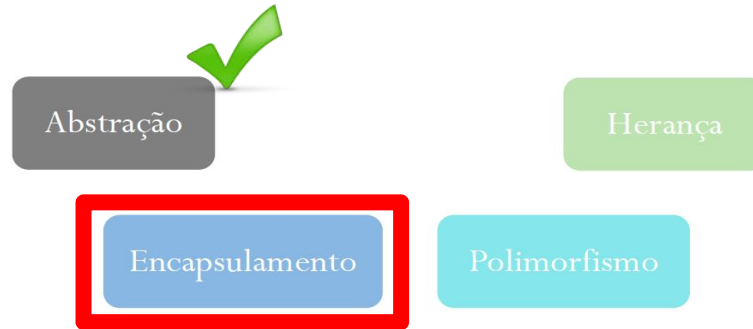
# Modificadores de Acesso

Como boas práticas (best practices) do Java, na maioria das declarações de variáveis de atributos da instância são definidos com a palavra-chave `private`, para garantir a segurança de alterações acidentais, sendo somente acessíveis através dos métodos. Essa ação tem como efeito ajudar no encapsulamento dos dados, preservando ainda mais a segurança e a aplicação de programação orientada a objetos do Java.

# Modificadores de Acesso

## Os quatro pilares da POO

Veremos em detalhes cada uma delas no momento apropriado.



# Encapsulamento

# Encapsulamento

Encapsulamento é a técnica que faz com que detalhes internos do funcionamento uma classe permaneçam ocultos para os objetos. Por conta dessa técnica, o conhecimento a respeito da implementação interna da classe é desnecessário do ponto de vista do objeto, uma vez que isso passa a ser responsabilidade dos métodos internos da classe.

A ideia é encapsular, isto é, esconder todos os membros de uma classe, além de esconder como funcionam as rotinas (no caso métodos) do nosso sistema.

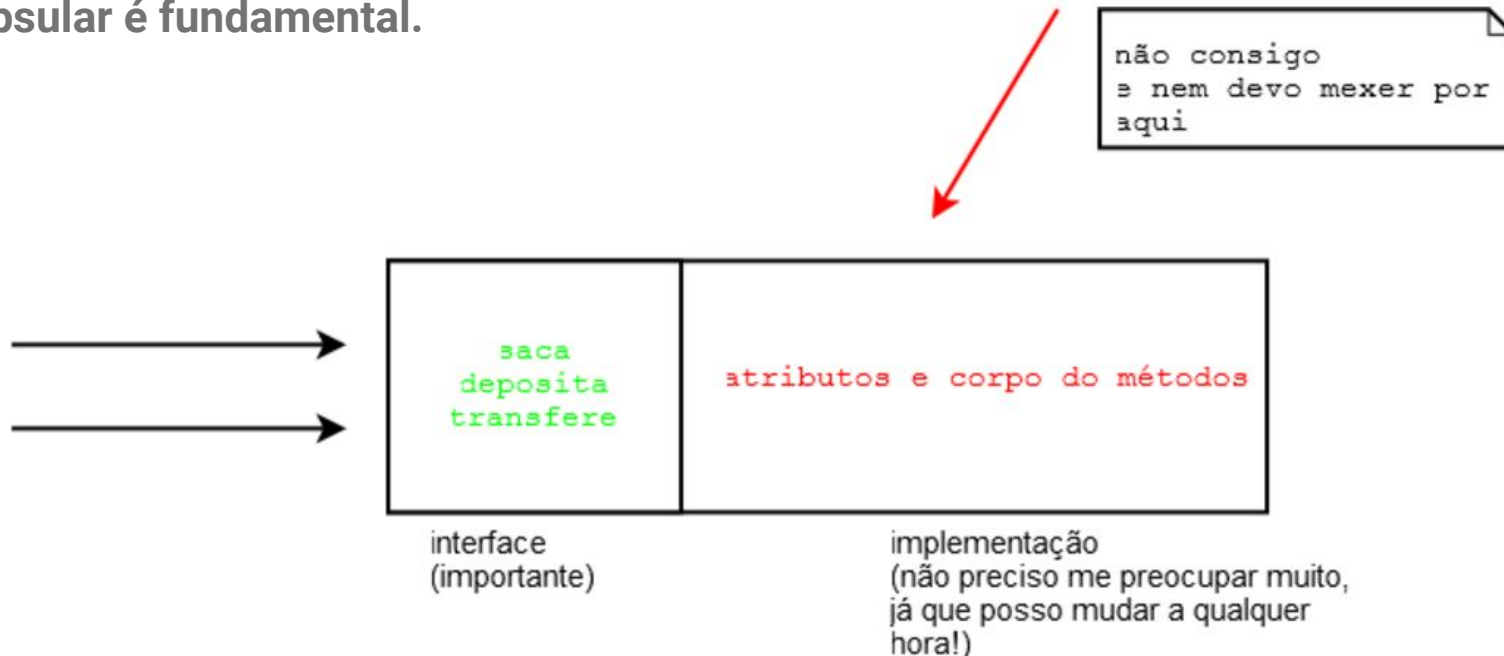
# Encapsulamento

Encapsular é fundamental para que seu sistema seja suscetível a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está encapsulada.

Como podemos melhorar o encapsulamento de nossa classe?

# Encapsulamento

Encapsular é fundamental.



O conjunto de métodos públicos de uma classe é também chamado de interface da classe, pois esta é a única maneira a qual você se comunica com objetos dessa classe.

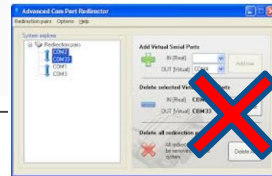
## Encapsular é fundamental.

### **Programando voltado para a interface e não para a implementação**

É sempre bom programar pensando na interface da sua classe, como seus usuários a estarão utilizando, e não somente em como ela vai funcionar.

A implementação em si, o conteúdo dos métodos, não tem tanta importância para o usuário dessa classe, uma vez que ele só precisa saber o que cada método pretende fazer, e não como ele faz, pois isto pode mudar com o tempo.

Essa frase vem do livro Design Patterns, de Eric Gamma et al. Um livro cultuado no meio da orientação a objetos.





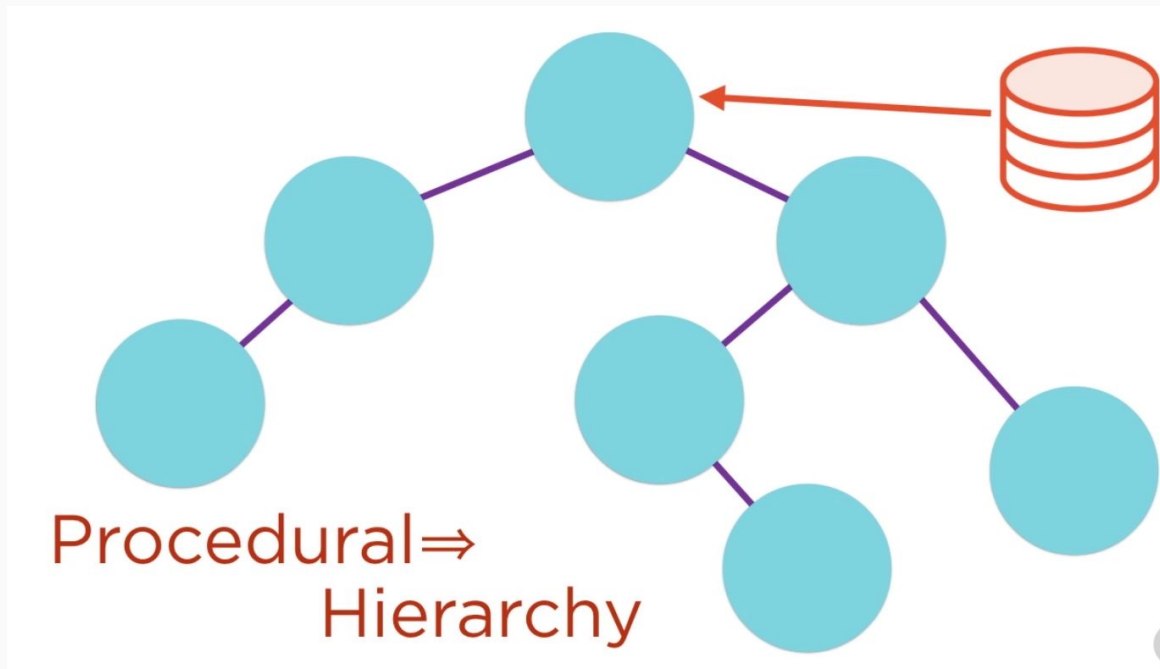
## **Encapsular é fundamental.**

Comece a se preocupar em como os outros objetos/classes (usuários) usarão a sua classe.

Sempre que vamos acessar um objeto, utilizamos sua interface. Existem diversas analogias fáceis no mundo real:

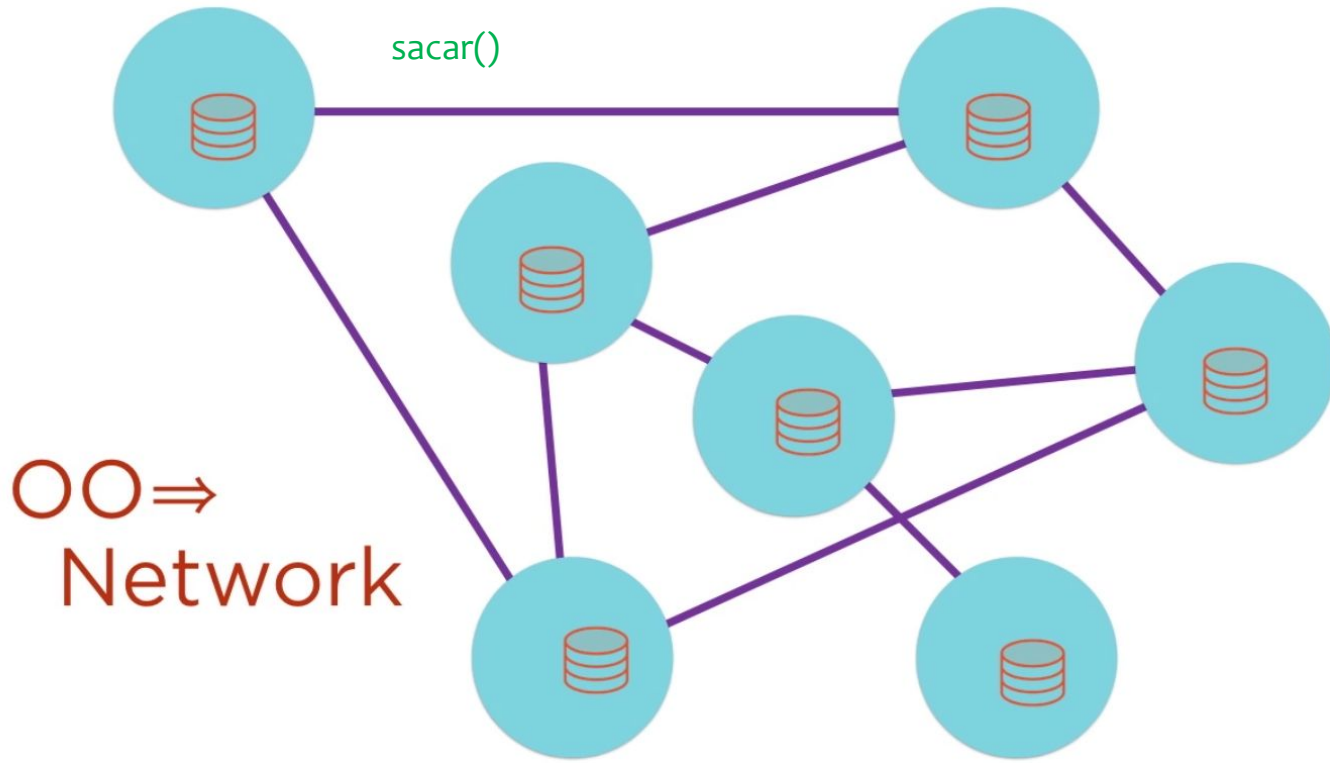
Quando você dirige um carro, o que te importa são os pedais e o volante (interface) e não o motor que você está usando (implementação). É claro que um motor diferente pode te dar melhores resultados, mas o que ele faz é o mesmo que um motor menos potente, a diferença está em como ele faz. Para trocar um carro a álcool para um a gasolina você não precisa reaprender a dirigir! (trocar a implementação dos métodos não precisa mudar a interface, fazendo com que as outras classes continuem usando eles da mesma maneira).

O modelo procedural é ***data centric***, e o fluxo de dados é muito importante. O sistema possui acesso a um conjunto de dados e todos os módulos alteram esses dados de alguma forma.



# Encapsulamento

O OO não é ***data centric***, parece um ***rede***. Cada objeto será responsável por gerenciar o seu próprio conjunto de dados.



# Encapsulamento

```
public class Cliente {  
  
    private String nome;  
    private String endereco;  
    private String cpf;  
    private int idade;  
  
    public void mudarCPF(String cpf) {  
        if (validarCPF(cpf)) {  
            this.cpf = cpf;  
        } else {  
            System.out.println("Não foi possível alterar o CPF.");  
        }  
    }  
  
    private boolean validarCPF(String cpf) {  
        // série de regras aqui, falha caso não seja válido  
    }  
}
```

E o dia que você não precisar verificar o CPF de quem tem mais de 60 anos?

# Encapsulamento

```
public void mudarCPF(String cpf) {  
    if (this.idade <= 60) {  
        validarCPF(cpf);  
    }  
    this.cpf = cpf;  
}
```

Qual é a modificação necessária no usuário dessa classe?

# Modificadores de Acesso em Conjunto com o Encapsulamento

# Modificadores de Acesso em Conjunto com o Encapsulamento

## Tabela dos modificadores de acesso

	private	default	protected	public
mesma classe	sim	sim	sim	sim
mesmo pacote	não	sim	sim	sim
pacotes diferentes (subclasses)	não	não	sim	sim
pacotes diferentes (sem subclasses)	não	não	não	sim

# Getters e Setters



# Getters e Setters

O modificador `private` faz com que ninguém consiga modificar, nem mesmo ler, o atributo em questão.

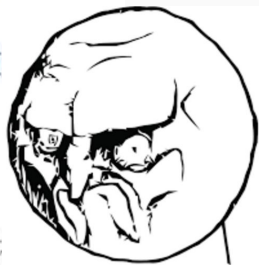
Apesar de termos o método para exibir as informações sobre a conta, pode ser que em um determinado projeto, queiramos que o usuário da classe possa construir sua própria mensagem, para isso ele terá de ter acesso ao valor do atributo, mas devemos fazer de uma forma a manter o atributo privado.

O que fazer? Criar métodos de acesso!

# Getters e Setters

Vamos então criar um método, digamos pegaSaldo, para realizar essa simples tarefa:

```
class Conta {  
  
    private double saldo;  
  
    // outros atributos omitidos  
  
    public double pegaSaldo() {  
        return this.saldo;  
    }  
  
    // deposita() e saca() omit  
  
}  
  
public static void main(String[] args) {  
    Conta minhaConta = new Conta();  
    minhaConta.depositar(1000);  
    System.out.println("Saldo: " + minhaConta.pegaSaldo());  
}
```



# Getters e Setters

Para permitir o acesso aos atributos (já que eles são **private**) de uma maneira controlada, a prática mais comum é criar dois métodos, um que **retorna o valor** e outro que **muda o valor**.

A convenção para esses métodos é de colocar a palavra get ou set antes do nome do atributo. Apenas um exemplo, manteremos o único modo de acesso ao saldo de nossa classe através dos métodos sacar e depositar, onde colocamos lógica para proteger a semântica.

```
public double getSaldo() {  
    return this.saldo;  
}  
  
public void setSaldo(double saldo) {  
    this.saldo = saldo;  
}
```

É uma má prática criar uma classe e, logo em seguida, criar getters e setters para **todos** seus atributos. Você só deve criar um getter ou setter se tiver a real **necessidade**. Repare que nesse exemplo `setSaldo` não deveria ter sido criado, já que queremos que todos usem `depositar()` e `sacar()`.

Outro detalhe importante, um método `getX` não necessariamente retorna o valor de um atributo que chama X do objeto em questão. Isso é interessante para o encapsulamento. Imagine a situação: queremos que o banco sempre mostre como saldo o valor do limite somado ao saldo (uma prática comum dos bancos que costuma iludir seus clientes).

```
private double limite; // adicionando um limite a conta

public double getSaldo() {
    return this.saldo + this.limite;
}
```

# Getters e Setters

O código nem possibilita a chamada do método **getLimite()**, ele não existe. E nem deve existir enquanto não houver essa necessidade.

O método **getSaldo()** não devolve simplesmente o saldo... e sim o que queremos que seja mostrado como se fosse o saldo.

Utilizar **getters** e **setters** não só ajuda você a proteger seus atributos, como também possibilita ter de mudar algo em um só lugar... chamamos isso de **encapsulamento**, pois esconde a maneira como os objetos guardam seus dados. É uma prática muito importante.

# Construtores

# Construtores

Construtores são métodos (**CUIDADO**) especiais chamados pelo sistema no momento da **criação de um objeto**. Eles não possuem valor de retorno, porque você não pode chamar um construtor para um objeto, você só usa o construtor no momento da inicialização do objeto.

Quando usamos a palavra chave **new**, estamos construindo um objeto. Sempre quando o **new** é chamado, ele executa o construtor da classe.

Construtor é como se fosse um método que possui o mesmo nome da classe.

```
class Conta {  
  
    int numero;  
    double saldo;  
  
    // construtor  
    public Conta(parametros) {  
        // faça coisas  
    }  
  
}
```



## O construtor default

Até agora, as nossas classes não possuíam nenhum construtor. Então como é que era possível dar `new`, se todo `new` chama um construtor **obrigatoriamente**?

Quando você não declara nenhum construtor na sua classe, o Java cria um para você. Esse construtor é o **construtor default**, ele não recebe nenhum argumento e o corpo dele é vazio.

A partir do momento que você declara um construtor, o construtor default não é mais fornecido.

```
public static void main(String[] args) {  
    Conta minhaConta = new Conta();  
    minhaConta.depositar(1000);  
}
```

# Construtores

Para o que eu posso utilizar um construtor?

Construtores representam uma oportunidade de inicializar seus dados de forma organizada.

# Perguntas?