UNIVERSIDADE FEDERAL DE LAVRAS

Técnicas de Programação Aplicada à Engenharia

GAC111 – Turmas 22A e 22B

2021-2

Encapsulamento e Visibilidade

Julio César Alves *DAC/ICET* *juliocesar.alves[a]ufla.br*

Hoje vamos ver **um dos conceitos mais importantes** da Orientação a Objetos.

Tipos Abstratos de Dados

Suponha que você tenha feito um código em C ou C++ (sem usar Orientação a Objetos) para um banco, que trate uma **conta bancária**.

Suponha que esta conta tenha um limite especial, ou seja, o cliente pode ficar devendo até um certo valor que o banco cobre a despesa por um determinado prazo.

Você definiu então **um registro (struct)** para guardar os **dados** da conta (como titular, número, saldo, limite, etc.).

E criou **funções** para as **operações** de saldo, transferência e saque. Na implementação de tais funções você tratou, por exemplo, o limite especial.

Nós chamamos de **Tipo Abstrato de Dados** (TAD), um conjunto de dados e as operações sobre eles com uma finalidade específica.

Neste nosso exemplo, podemos dizer que você definiu um TAD para representar uma conta bancária.

Encapsulamento e Structs

Qual é o problema com os **structs**?

Os outros programadores!!

Suponha agora que outro programador usará seu código (sem alterá-lo) para fazer outra parte do sistema do banco.

- Espera-se então que ele utilize as funções que você criou para realizar as operações na conta, certo?
- Mas o que impede que ele acesse os dados (a struct) diretamente, e altere os valores dos dados?
  - E, ao fazer isso, como você pode garantir que os limites da conta serão seguidos, por exemplo?

Encapsulamento e Structs

Qual é o problema com os **structs**?

Os outros programadores!!

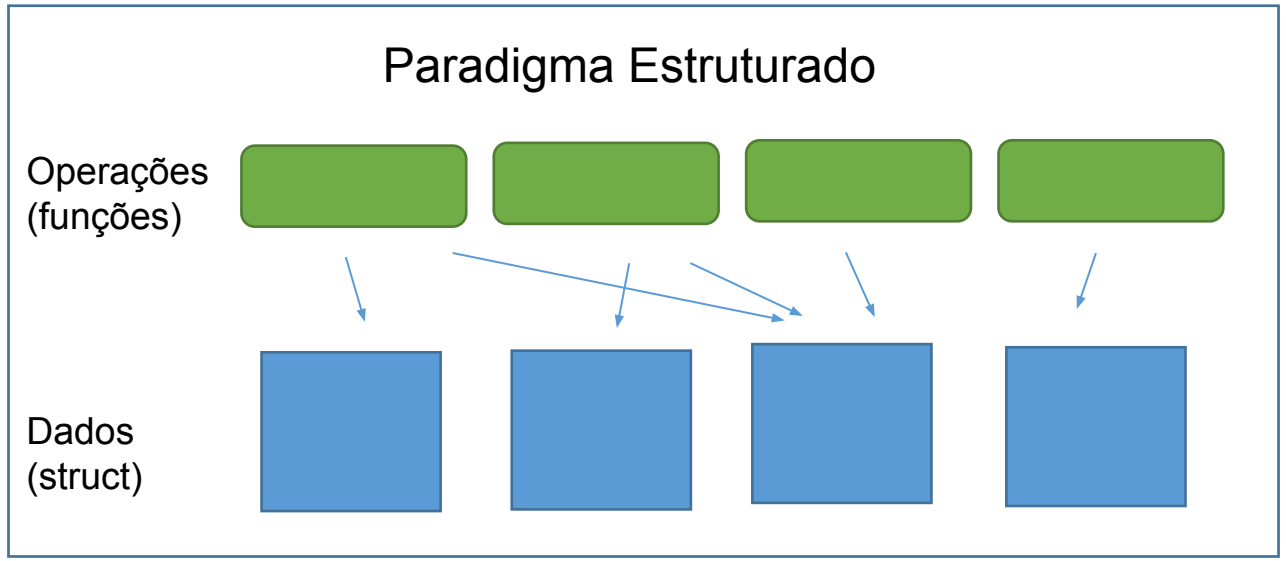
Note que uma struct não consegue proteger seus dados.

Na verdade, como o outro programador consegue acessar a struct que você definiu, ele pode alterar os dados sem usar as operações. Isso pode causar muitos problemas em grandes equipes de desenvolvimento.

Structs x Classes

Os **TADs** implementados usando **structs** possuem **operações** que **não são fortemente ligadas aos dados**.

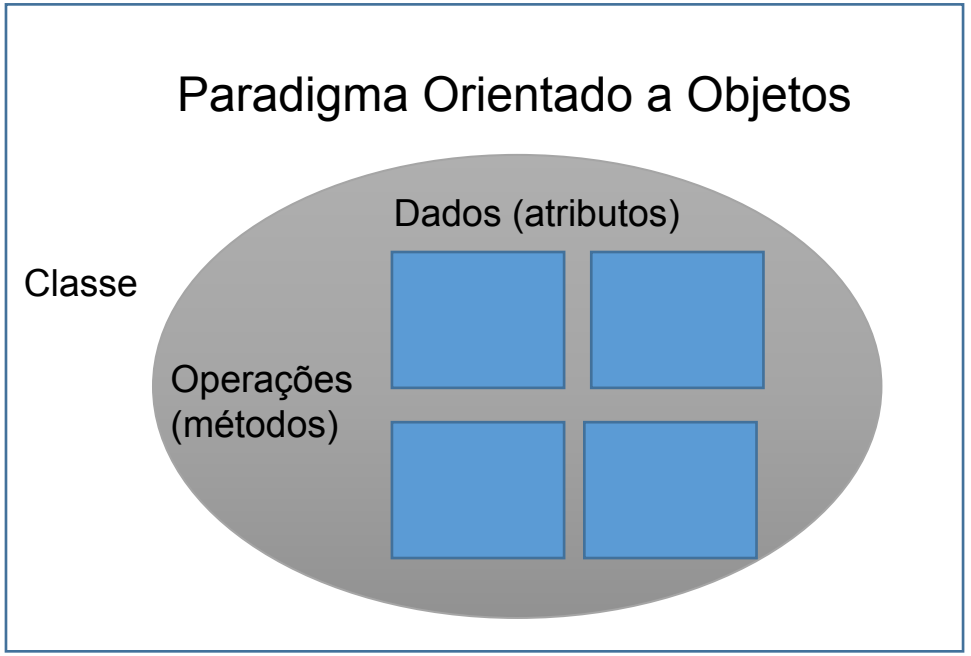
Outras funções podem ser criadas em outros lugares e acessarem os dados da struct.



Já na **Orientação a Objetos**, um **TAD é** representado por **uma classe**.

E a diferença é que as **classes podem encapsular seus dados**. Ou seja, elas não permitem que os dados (atributos) sejam acessados de fora da classe.

A **única forma de acessar ou alterar um dado é através dos métodos** definidos pela própria classe. Portanto, ela tem total controle sobre se e como os atributos podem ser alterados.



Encapsulamento

Portanto, em OO é muito importante que saibamos separar **o que** uma classe faz de **como** a classe faz o que precisa.

O que a classe faz é representado pela assinatura de seus **métodos acessíveis de fora da classe**, o que chamamos de **interface de comunicação**.

Já **como a classe faz** o que precisa, é representado pelos seus **atributos** e pela **implementação dos métodos** e isso **deve ser encapsulado** pela classe.

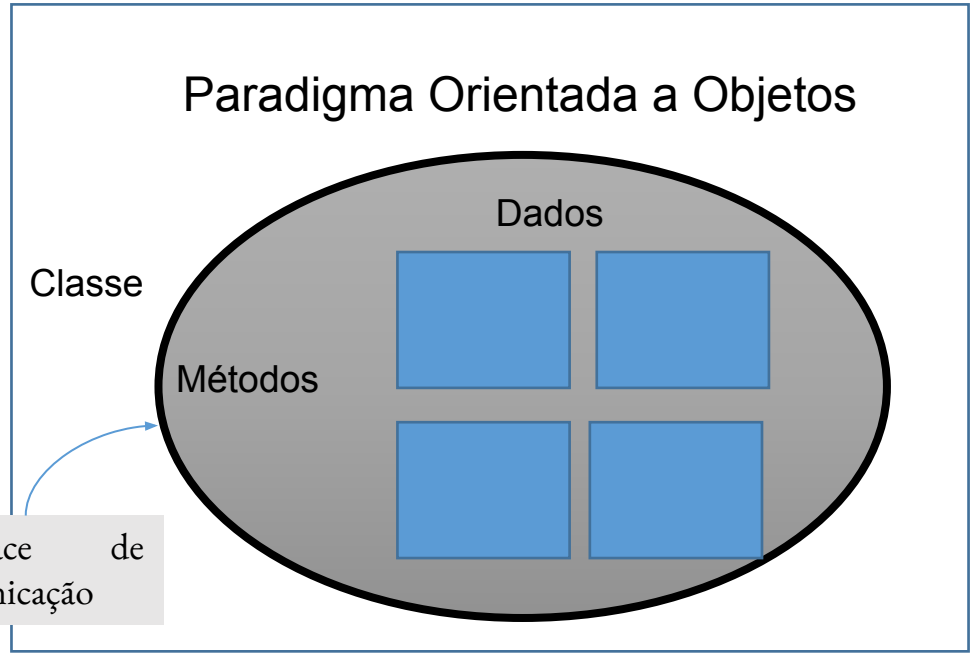
No nosso exemplo de conta bancária, você poderia então definir uma **classe** para representar uma conta e fornecer para o outro programador apenas as assinaturas dos métodos desta classe.

- Veja que você não precisa fornecer os dados (antes o outro programador tinha acesso à struct, mas agora ele não tem acesso aos atributos da classe).

Dessa forma, sem mexer no código da sua classe, o outro programador não terá como alterar os dados da conta, e será obrigado a usar os métodos que você definiu para lidar com a conta bancária.

Interface de comunicação

Reforçando, nós chamamos de **interface de comunicação** de uma classe os seus métodos que podem ser chamados de fora da classe.



Esse nome se refere ao fato de que nós conseguimos comunicar com um objeto (enviar mensagens para ele) somente através de seus métodos.

Nós dizemos então que o **encapsulamento** garante que o único **meio de acessar** ou alterar **dados** de uma classe é **através** da sua **interface de comunicação**.

Interface de comunicação

Um bom exemplo que ilustra o encapsulamento é um telefone fixo comum.

Você usa o telefone através de sua interface (os botões) e não precisa saber como é o funcionamento interno.

A mesma analogia pode ser feita com um caixa eletrônico.

Você usa as opções que aparecem pra você (interface) mas não precisa saber como isso funciona internamente. E você só tem acesso aos dados que a interface disponibiliza.

Fiquei curioso com o significado da palavra cápsula!

cáp·su·la

(latim *capsula, -ae*, pequena caixa)

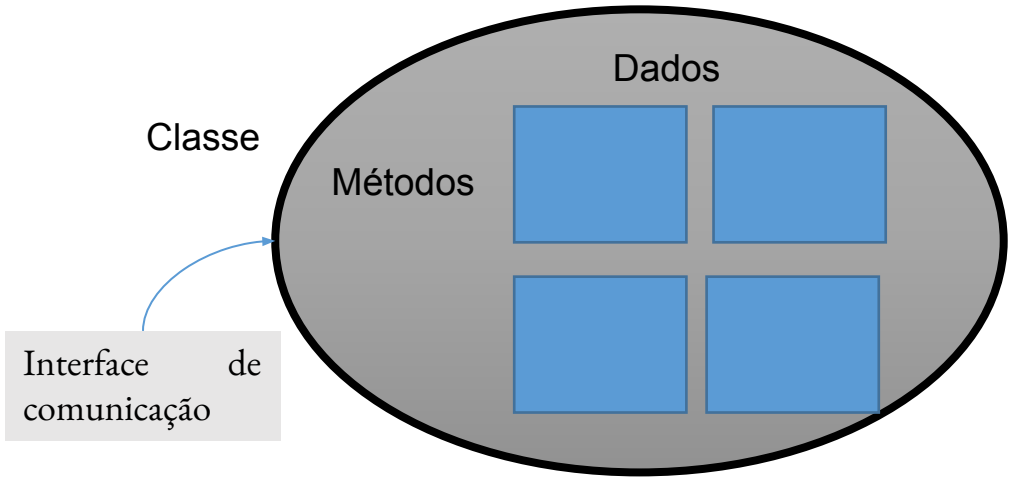
substantivo feminino

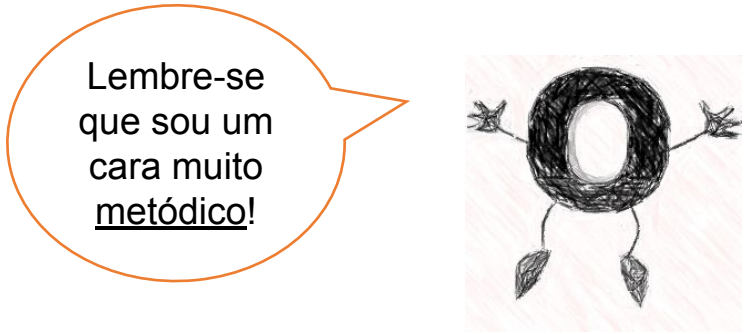
1. Pequeno recipiente.
2. Invólucro metálico que cobre a rolha e uma parte do gargalo de uma garrafa.
3. [Portugal] Tampa de garrafa.
4. Glóbulo gelatinoso ou pequeno recipiente ingerível em que se encerra um medicamento.
5. [Botânica] Invólucro de certas sementes.
6. [Anatomia] Membrana que envolve certas estruturas anatômicas (ex.: *cápsula articular; cápsula do cristalino*).

O **encapsulamento** garante o **princípio da ocultação**.

Ele diz que devemos expor apenas a interface de comunicação entre os objetos.

Interface de comunicação





Como a **única forma de se comunicar com um objeto** e saber seus dados é **através dos métodos**, isso traz **duas vantagens** muito importantes.

A primeira é que se você mantiver a interface de comunicação (assinaturas dos métodos) **você pode alterar a representação interna dos dados, sem afetar o resto do sistema**.

Por exemplo, você pode alterar a forma como você guarda os dados da conta bancária, que não vai causar nenhum efeito colateral em quem usa a classe.

A outra é que para alguém **conseguir usar um objeto** de uma classe que você definiu, **basta fornecer a interface de comunicação**, ou seja, as assinaturas dos métodos.

Uma coisa é o conceito em si (ex: encapsulamento).

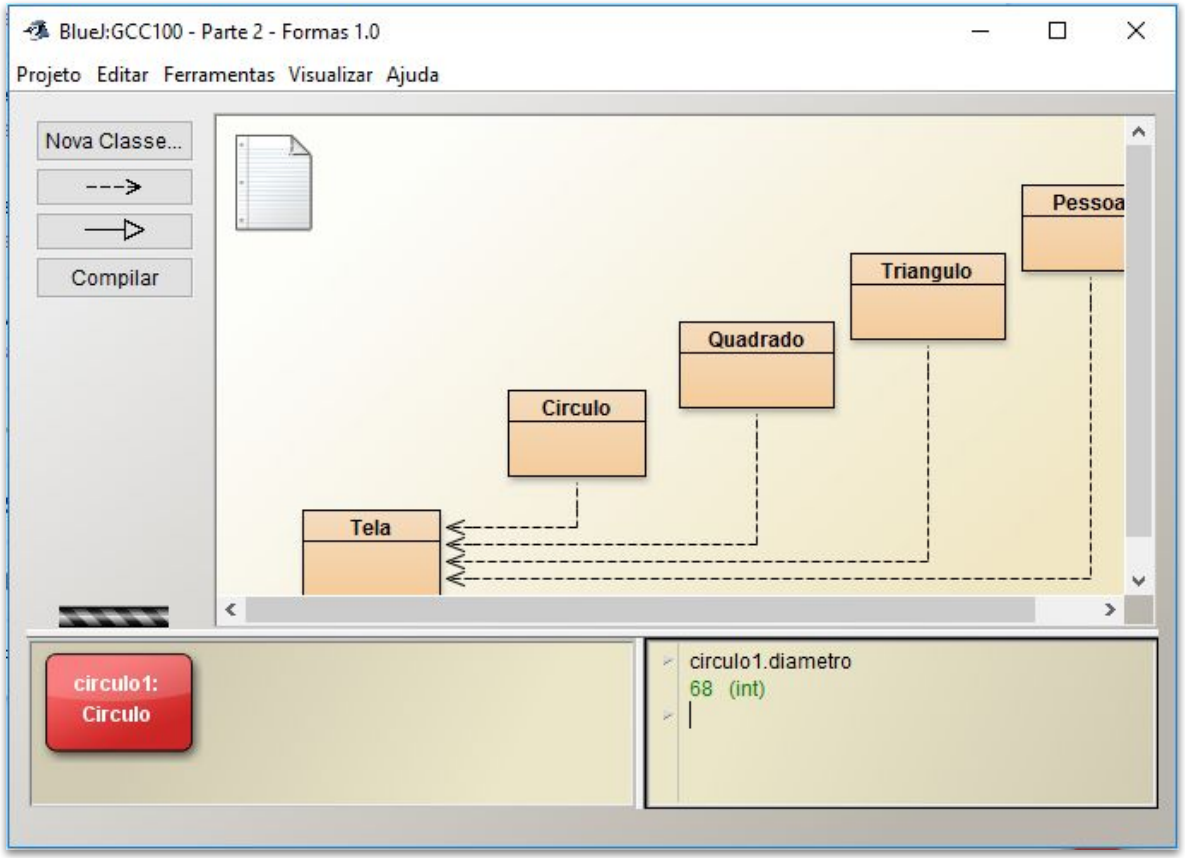
Outra coisa é como cada linguagem de programação adota o conceito.

Vamos ver agora a linguagem Java nos permite usar o conceito de encapsulamento.

Encapsulamento em Java

Nós poderíamos imaginar que, já que a Orientação a Objetos define que os dados têm que estar encapsulados, os atributos de uma classe em Java seriam inacessíveis fora dela.

Mas no nosso exemplo de formas geométricas, vemos que nós conseguimos acessar o valor de um atributo mesmo fora da classe.



Encapsulamento em Java: **`private`**

É que para encapsular o atributo em uma classe Java, nós precisamos dizer isso explicitamente.

Nós usamos a palavra-chave **`private`** para indicar que os atributos são privados da classe. Ou seja, ninguém tem acesso a eles de fora da classe.

```
class Circulo
{
    int diametro;
    int posX;
    int posY;
    String cor;
    boolean estaVisivel;
```

```
public class Circulo
{
    private int diametro;
    private int posX;
    private int posY;
    private String cor;
    private boolean estaVisivel;
```

Ao usar a palavra-chave `private` nós garantimos que a única forma de saber o valor dos atributos ou alterá-los, de fora da classe, é através dos métodos da classe.

Orientação a Objetos real em Java

```
class Jogador {
    String nome;
    float altura;
    int habilidadeDrible;

    void chutar(int direcao) {
        // ... algum código
    }

    void correr() {
        // ... algum código
    }
}
```

Obs: a classe acima fere alguns princípios de OO, mas eles ainda serão vistos.

Em todas as classes apresentadas até agora, nos slides e exemplos, foi colada a observação (caixa laranja) de que a classe feria alguns princípios de OO.

As classes feriam justamente o princípio do encapsulamento por não deixar os atributos privados.

Lembre-se: **todos os atributos devem sempre ser privados**!

Apesar de Java ser uma linguagem Orientada a Objetos, o simples fato de fazer um programa em Java não é garantia que seu programa realmente segue os conceitos de Orientação a Objetos.

Visibilidade de atributos e métodos

Além da palavra-chave `private` existe também a **`public`** que, como era de se esperar define exatamente o contrário (que algo é acessível de fora da classe).

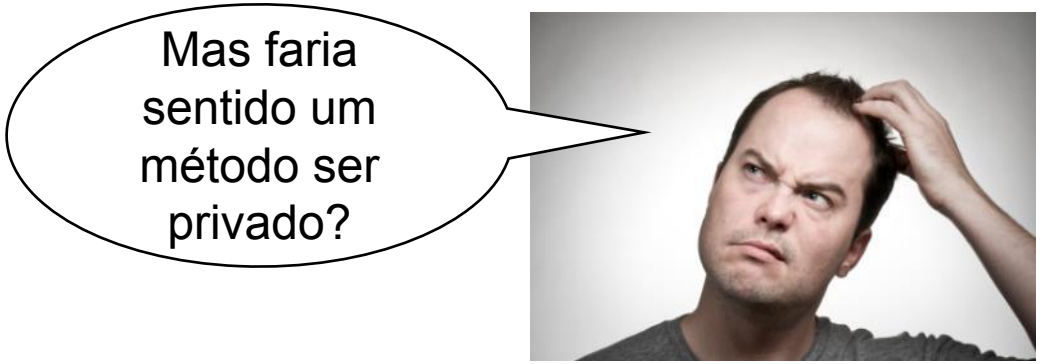
Nós dizemos que esses modificadores definem a **visibilidade** de um atributo ou de um método.

E, exatamente, eles também são usados para métodos!

```
/**
 * Deixa o círculo visível.
 */
public void exibir()
{
    estaVisivel = true;
    desenhar();
}
```

Portanto, todos os métodos que possam ser chamados de fora da classe precisam ter o modificador **`public`**.

Métodos Privados



Os **métodos privados** (ou internos) são muito **úteis**, principalmente, por duas razões:

- Para tornar o **código mais legível e facilitar a manutenção**, quando o código de um **método** fica muito **grande**, o ideal é **dividi-lo em métodos menores**. E esses métodos menores não devem estar acessíveis fora da classe.
- O segundo ponto é que muitas vezes dois métodos públicos possuem um trecho idêntico. O ideal é separar esse trecho em um **método privado e reutilizá-lo** nos dois métodos públicos.

Classe, por favor, não avacalhe comigo, gosto de ser organizado!

Em resumo:

- Atributos → `private`
- Apenas métodos podem ser `public`

Programador! Ei é você mesmo!

Não vá criar métodos com mais de um propósito e nem muito grandes.

E lembre-se: nomes sugestivos e comentários, por favor.

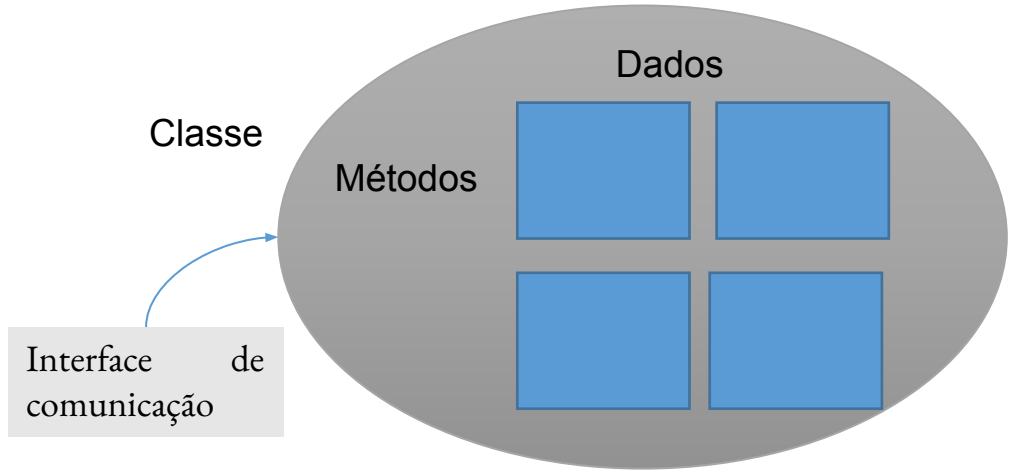
Idealmente, um método deve ter:

- Uma única missão, resolver um só problema.
- De 1 a 15 linhas.
- Ter um nome adequado, sugestivo (já foi falado do *camelCase*?).
- Ter comentários.

Métodos públicos = interface de comunicação

Agora que sabemos que os métodos podem ser públicos e privados, vamos completar a definição de interface de comunicação.

Nós chamamos de **interface de comunicação** o conjunto de **métodos públicos** de uma classe.



Métodos `get` e `set`

O que fazemos quando alguém precisa consultar o valor de um atributo, já que ele não pode ser acessado de fora da classe?

Nós criamos métodos para retornar os valores dos atributos.

- É uma prática comum, especialmente em Java, criar métodos **`get`** e **`set`** para consultar e modificar, respectivamente, os valores de um atributo.

```
private String nome;
private int velocidade;

public int getVelocidade() {
    return velocidade;
}
```

Métodos `get` e `set` vs. Atributos públicos

Não estou vendo nenhuma diferença entre o atributo ser público ou ter **um get e um set** que permite alterar e acessar o atributo.

Suponha que uma classe `Circulo` tenha o método `setDiametro`. O método simplesmente altera o valor de um atributo `diametro`.

Suponha então que a classe `Circulo` é utilizada em diversos lugares em um grande sistema e existem diversas chamadas ao método `setDiametro`.

Mas então você percebe que da forma como está, alguém poderia alterar o diâmetro do círculo para um valor negativo. O que você precisa fazer para corrigir o problema? Em quantos lugares o código precisa ser alterado?

Você precisa alterar apenas no método `setDiametro`.

Mas se fosse um atributo público seria necessário tratar em todos os lugares onde ele é alterado, e o pior, alguém poderia não lembrar de fazer isso quando precisasse fazer uma nova alteração do atributo.

- E, por fim, a responsabilidade de garantir valores válidos para seus atributos é da própria classe!

Métodos `get` e `set` vs. Atributos públicos

Mas no meu caso, não preciso de nenhuma verificação então vou deixar o atributo público mesmo.

ERRADO!

Essa é uma questão de boa prática de programação.

Mesmo que você não precise fazer uma validação agora, pode ser que no futuro você descubra que precisa

Vocês não sabem que diferença isso faz em um sistema complexo de uma grande empresa.

Métodos get e set

ERRADO!

Sou seu gerente, preciso de uma classe conta bancária que guarde o saldo e permita ao cliente movimentar a conta, depositar, sacar, saber o saldo, etc.

Agora que aprendi que precisa de `get` e `set` ficou fácil! É só criar isso pra todos os atributos.

Em um sistema bancário pode existir uma classe para representar uma conta bancária e essa classe provavelmente terá um atributo saldo.

Na vida real, como o saldo de uma conta é alterado?

Fazendo saques e depósitos, certo?

Da mesma forma, em uma classe conta devem existir os métodos `sacar` e `depositar`.

- Veja que assim não faz sentido existir um método `setSaldo`.

A Palavra-chave `this`

Nós podemos fazer um método de uma classe para mudar o valor de um atributo usando um parâmetro com o mesmo nome do atributo.

Mas não podemos simplesmente fazer `velocidade = velocidade`, pois o compilador usaria o nome de escopo mais próximo, ou seja, o parâmetro.

```
class Carro {
    // ... outros atributos
    private int velocidade;

    // ... outros métodos

    public void setVelocidade(int velocidade) {
        this.velocidade = velocidade;
    }
}
```

Para fazer isso nós usamos a palavra-chave **`this`**.

- Ela nada mais é que uma referência (“ponteiro”) para o próprio objeto que teve o método chamado.

Um erro comum é fazer `velocidade = this.velocidade.`

A falha de encapsulamento mais comum em OO

Suponha que uma classe `Casa` é formada por uma parede, uma janela e um telhado, sendo estes objetos das classes `Quadrado` e `Triangulo`.

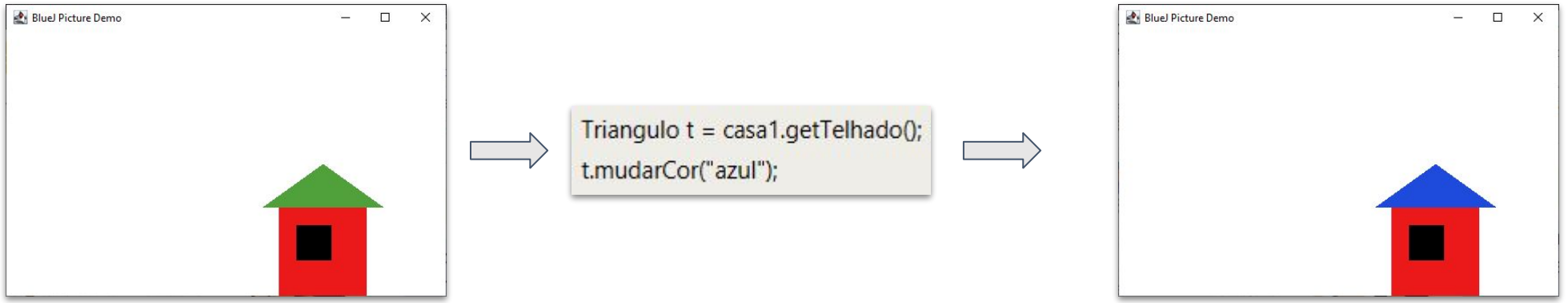
Suponha ainda que exista um método `get` para retornar o telhado da casa.

```
83      public Triangulo getTelhado()
84      {
85          return telhado;
86      }
```

O que acontece se de fora da classe `Casa`, eu declarar uma variável que recebe o retorno do método `getTelhado` e chame o método `mudarCor` usando essa variável?

Isso vai alterar a cor do telhado da casa?

A falha de encapsulamento mais comum em OO



Sim, vai alterar a cor do telhado da casa. Isso acontece porque o método `getTelhado` retorna uma referência (endereço) do telhado da casa.

Mas qual é o problema disso?

Nós aprendemos com o conceito de encapsulamento que a classe deve ser a única capaz de alterar os seus dados. Dessa forma, se uma casa possui um telhado, os dados desse telhado não deveriam ser alterados fora da classe `Casa`.

Clone: cópias de objetos

Mas vai ter caso que outra classe precisa ter o objeto telhado apenas para consultar informações, por exemplo. O que pode ser feito então?

Nesses casos deve ser retornada uma cópia do objeto (telhado) e não a referência para o objeto em si. A linguagem Java padroniza um nome de método para retornar uma cópia para o objeto.

Mais pra frente veremos como usar as facilidades da linguagem Java para criar uma cópia de um objeto através de um método *clone*.

Mas se você precisasse implementar um método desse sem conhecer o clone, como você faria com o que já conhece da linguagem?

Há casos em que o desempenho é importante e evitamos cópias de objetos.

- Veremos no futuro outras estratégias para evitar a cópia e ainda respeitar o encapsulamento.

Eu não alterei meu atributo eu juro!

Então você só pode tê-lo deixado público!

Não, não... eu prestei atenção na aula de POO.! Tem o método `get` certinho

Então como você me explica que seu atributo foi alterado?

Alguém conhecia a identidade dele ☹

Lembre-se que é necessário se **dedicar** fora das aulas!!

Exercício – Conta Bancária

Considere um sistema bancário. Dentre as várias classes desse sistema existe uma classe que define o tipo de dado `Conta`. Implemente essa classe em Java (e um programa para testar a classe) considerando a seguinte descrição fornecida por um usuário do sistema:

"Cada conta possui um número de identificação e um saldo. É importante ter opções para saque e depósito em uma conta e para transferência entre contas. Em uma operação de saque, o saldo é decrementado de acordo com o valor sacado. Não é permitido sacar um valor maior do que o saldo. Em uma operação de depósito, o saldo é incrementado de acordo com o valor depositado. E em uma operação de transferência entre contas, o saldo da conta de origem é decrementado de acordo com o valor transferido, se o saldo for suficiente, e o saldo da conta de destino é incrementado de acordo com o valor transferido ."

Depois crie uma classe que tenha o método main e que teste a sua classe `Conta`.