

Desmistificando a Programação Orientada a Objetos

Nesta palestra, vamos mergulhar nos conceitos essenciais da Programação Orientada a Objetos, transformando o complexo em algo claro e aplicável no seu dia a dia como estudante e desenvolvedor.

Palestrante: Filippo Chiarion, estudante de Engenharia de Software

Agenda: O que Vamos Aprender?

1 Por que a POO é tão importante?

Descubra por que a POO domina o mercado e como ela vai transformar seu jeito de programar.

2 Classes e Objetos

Aprenda a criar os "moldes" (Classes) e os elementos da vida real (Objetos) dentro do seu código.

3 Os 4 Pilares

Vamos desvendar os quatro conceitos fundamentais que organizam e potencializam qualquer código em POO.

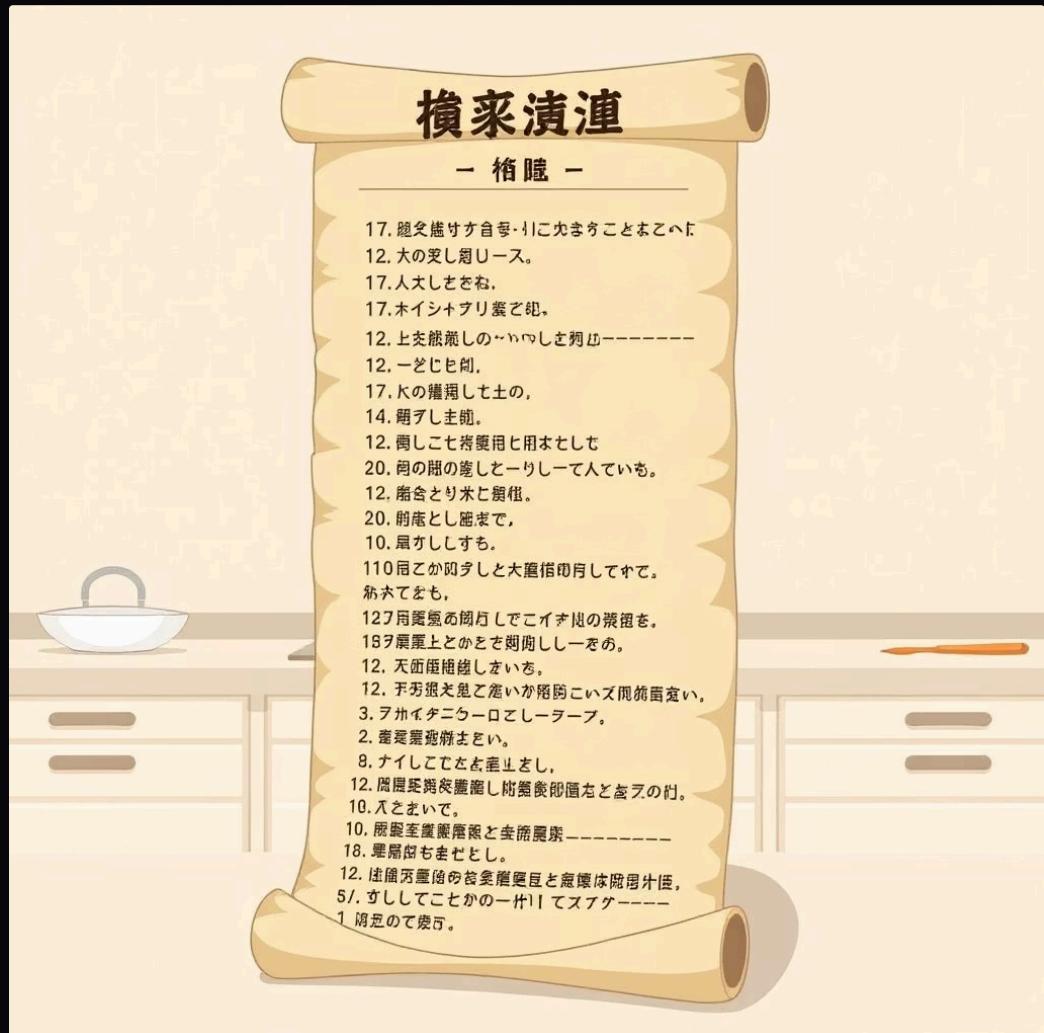
4 Vamos Praticar!

Aplicação prática dos conceitos aprendidos para solidificar o conhecimento.

A Grande Ideia: O Mundo Antes da POO

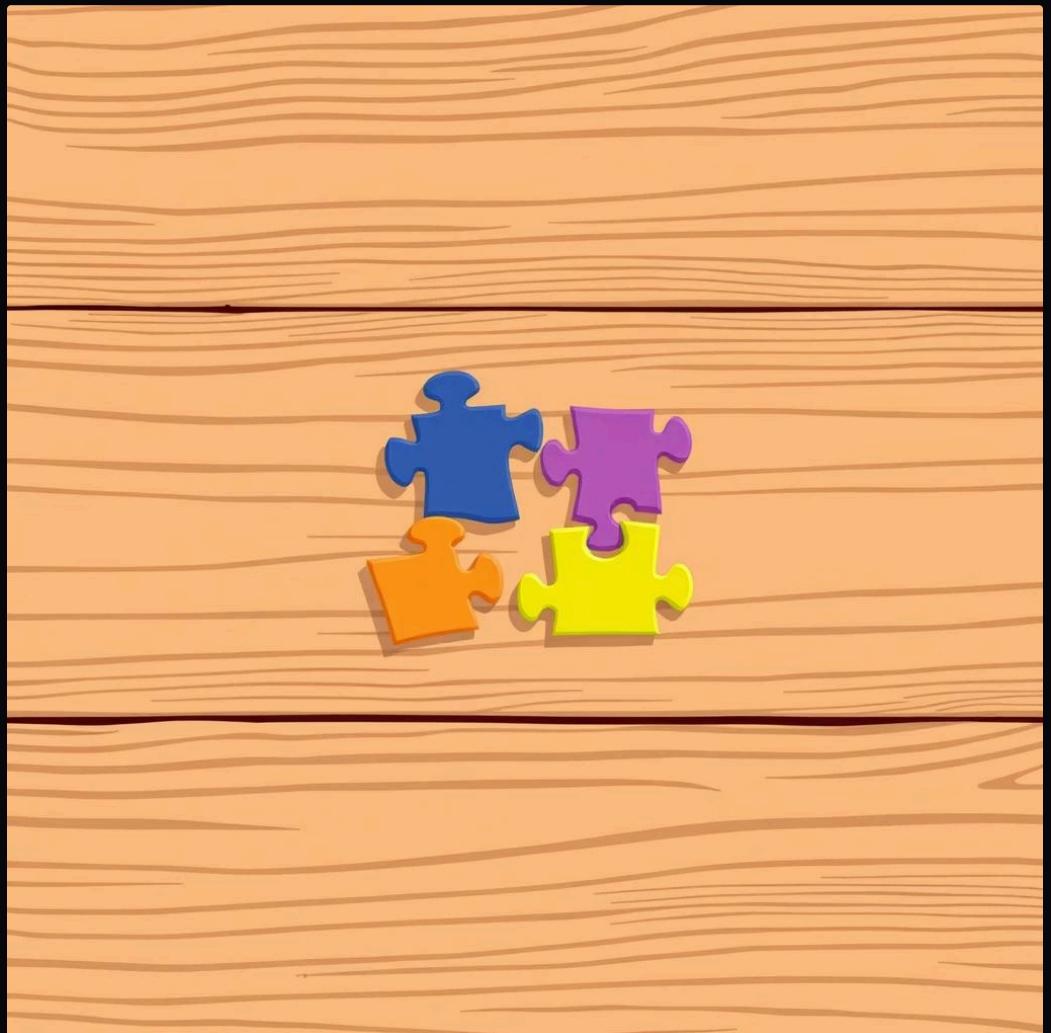
Antes da Programação Orientada a Objetos, predominava a programação procedural. Era como seguir um manual de instruções gigante para cada tarefa.

Programação Procedural



Pense em uma longa receita. É um roteiro único e sequencial, onde a ordem dos passos é rígida. Um único erro no início compromete todo o resultado final.

Com a POO



Pense em um quebra-cabeça. Cada peça é um bloco independente, projetado para se encaixar perfeitamente com os outros.

Conectando com o Mundo Real: A Necessidade da POO

Pense em um sistema real como um e-commerce: ele precisa gerenciar Clientes, Produtos e Pedidos. A POO nasceu para espelhar essa realidade, nos permitindo organizar o código com a mesma lógica do mundo real, com uma promessa clara de eficiência.



Reusabilidade de Código

Crie componentes que podem ser usados em diferentes partes do seu software, economizando tempo e esforço.



Organização e Clareza

Estruture seu código de forma lógica, tornando-o mais fácil de ler, entender e colaborar.



Manutenção Simplificada

Altere ou corrija partes do sistema sem afetar outras, reduzindo a complexidade de manutenção.

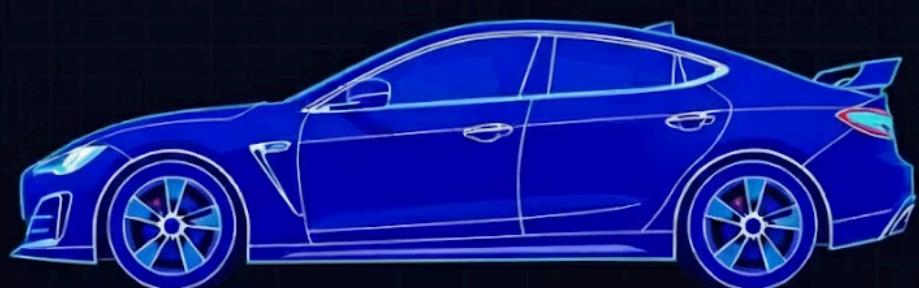
Classes

Pense em uma **Classe** como a planta baixa de uma casa. Ela define as características (o que a casa tem) e os comportamentos (o que a casa faz), mas não é a casa em si.

- **Atributos:** São as características de um objeto (cor, marca, modelo, ano).
- **Métodos:** São as ações que um objeto pode realizar (ligar(), acelerar(), frear()).

Nossa Primeira Classe em Java:

A Planta Baixa (O Molde)



Atributos:

cor
marca
modelo
ano

Métodos

ligar()
acelerar()
frear()
desligar()

Nossa Primeira Classe em Java

```
class Carro {  
    String cor;  
    String marca;  
    String modelo;  
    int ano;  
  
    public Carro() {}  
  
    void ligar() {  
        System.out.println("Carro ligado!");  
    }  
  
    void acelerar() {  
        System.out.println("Carro acelerando!");  
    }  
  
    void frear() {  
        System.out.println("Carro freando!");  
    }  
  
    void desligar() {  
        System.out.println("Carro desligado!");  
    }  
}
```

Objetos

Se a Classe é a planta, o **Objeto** é a casa construída. É uma instância real e concreta daquela planta, com suas próprias características e estados.

Criando Objetos em Java:

O Molde que Ganhou Vida

Objetos: A Construção de Instâncias



Atributos
cor: Azul
marca: Volkswagen
modelo: Fusca
ano: 1970

Métodos
ligar()

Atributos
cor: Vermelho
marca: Ferrari
modelo: 458 Italia
ano: 2015

Métodos
acelerar()

Dando Vida com new

```
public class Garagem {  
  
    public static void main(String[] args) {  
        Carro fusca = new Carro(); // Criando um objeto 'fusca'  
        fusca.cor = "Azul";  
        fusca.marca = "Volkswagen";  
        fusca.modelo = "Fusca";  
        fusca.ano = 1970;  
        System.out.println("Meu Fusca: " + fusca.cor + " " +  
        fusca.modelo);  
        fusca.ligar();  
  
        Carro ferrari = new Carro(); // Criando outro objeto 'ferrari'  
        ferrari.cor = "Vermelho";  
        ferrari.marca = "Ferrari";  
        ferrari.modelo = "458 Italia";  
        ferrari.ano = 2015;  
        System.out.println("Minha Ferrari: " + ferrari.cor + " " +  
        ferrari.modelo);  
        ferrari.acelerar();  
    }  
}
```

Construtores: Dando Vida aos Objetos

O que é um Construtor?

A sua principal função é inicializar os atributos do objeto com valores iniciais.

Pense no construtor como o "manual de montagem" de um objeto. É um método especial chamado automaticamente quando usamos a palavra `new`, garantindo que o objeto nasça pronto para o uso, com seus atributos essenciais já definidos.

Para que serve o `this`?

A palavra-chave `this` é usada dentro de uma classe para se referir ao próprio objeto. Ela resolve a ambiguidade quando o nome de um parâmetro é o mesmo do atributo da classe, deixando claro que estamos atribuindo o valor do parâmetro ao atributo do objeto.

Classe Carro com Construtor:

```
class Carro {  
    String cor;  
    String marca;  
    String modelo;  
    int ano;  
  
    // Construtor  
    public Carro(String cor, String marca, String modelo, int ano) {  
        this.cor = cor;  
        this.marca = marca;  
        this.modelo = modelo;  
        this.ano = ano;  
    }  
  
    void ligar() {  
        System.out.println("Carro ligado!");  
    }  
  
    void acelerar() {  
        System.out.println("Carro acelerando!");  
    }  
}
```

Criando um Objeto de Forma Direta:

```
public class Garagem {  
    public static void main(String[] args) {  
        Carro ferrari = new Carro("Vermelho", "Ferrari", "458 Italia",  
2015);  
        System.out.println("Minha Ferrari: " + ferrari.cor + " " +  
ferrari.modelo);  
        ferrari.acelerar();  
    }  
}
```

Os 4 Pilares da POO

Encapsulamento

Herança

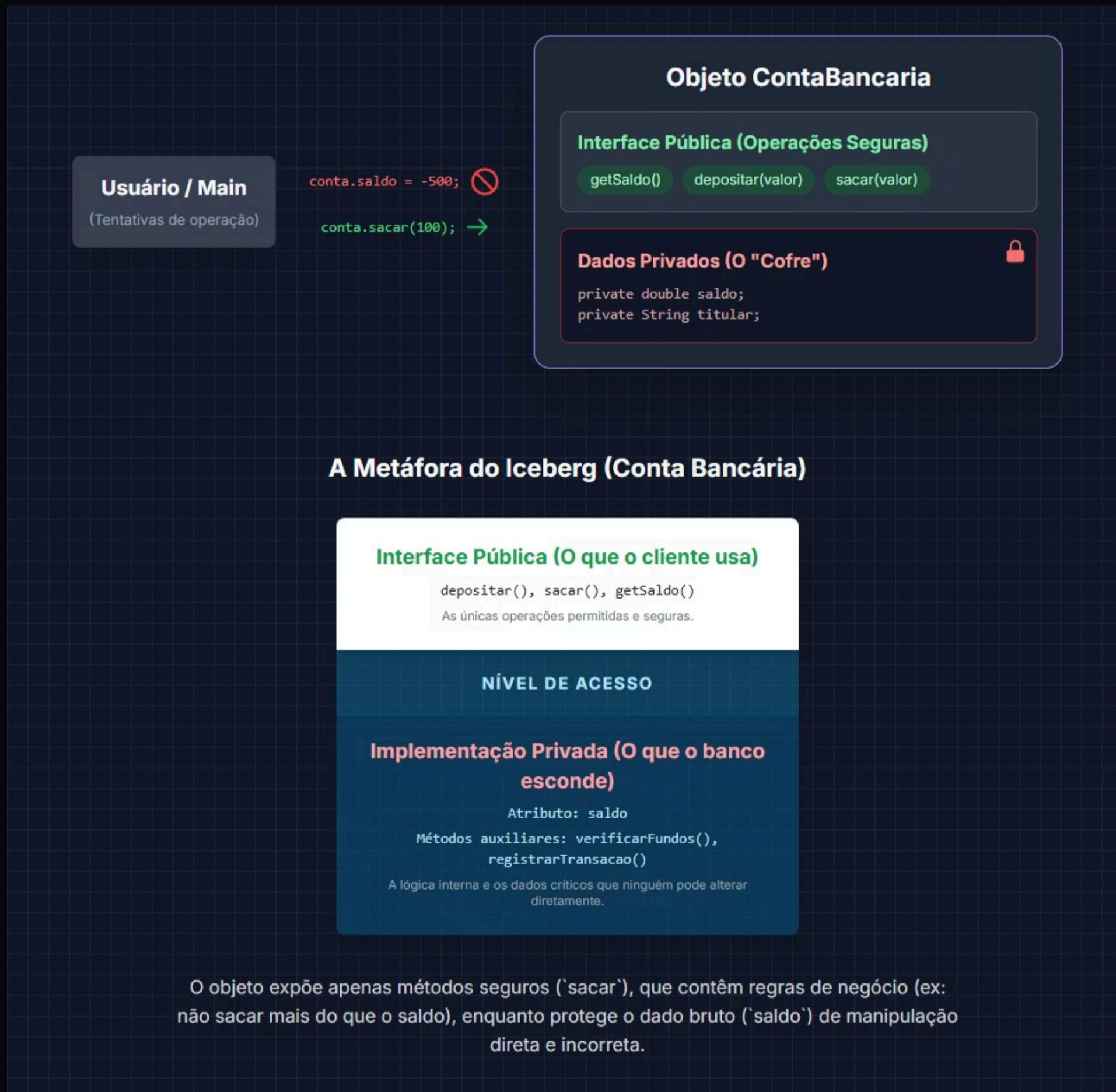
Polimorfismo

Abstração

Pilar 1: Encapsulamento

"Proteger dados sensíveis e garantir a integridade." O encapsulamento funciona como o cofre de um banco: protege os dados internos (como o saldo), expondo apenas operações seguras e controladas.

Encapsulamento na Prática (Java):



Controle de Acesso: public vs. private

O encapsulamento é reforçado pelos modificadores de acesso, que definem "quem" pode ver ou usar atributos e métodos de uma classe.

Public (Público)

- O que é?** É como a porta da frente da sua casa. Qualquer um pode ver e acessar.
- Onde tem acesso?** Em qualquer lugar do projeto (dentro da própria classe, em outras classes, etc.).
- Uso comum:** Métodos (como getters e setters), que são a interface segura com o mundo exterior.

Private (Privado)

- O que é?** É como um cofre dentro do seu quarto. Só quem está dentro do quarto (a própria classe) pode acessar.
- Onde tem acesso?** Apenas dentro da própria classe onde foi declarado.
- Uso comum:** Atributos (como cor, ano), para proteger os dados do objeto e garantir a integridade.

Controle o Acesso aos Dados: Getters e Setters

1. O que são Getters e Setters?

São métodos públicos que atuam como "portões" controlados para acessar e modificar os atributos privados de uma classe. Eles garantem que a interação com os dados internos do objeto ocorra de forma segura e validada, respeitando o princípio do encapsulamento.

2. Getter - Para LEITURA (buscar/obter valores)

O Getter (ou método "get") permite que outras classes **leiam** o valor de um atributo privado de forma controlada. Geralmente, retorna o valor do atributo sem alterá-lo.

```
public class ContaBancaria {  
    private double saldo;  
  
    // ... construtor e outros métodos ...  
  
    // Getter para saldo  
    public double getSaldo() {  
        return saldo;  
    }  
}
```

3. Setter - Para ESCRITA (alterar/definir valores)

O Setter (ou método "set") permite que outras classes **modifiquem** o valor de um atributo privado. É aqui que podemos adicionar regras de validação ou lógica de negócio antes de permitir a alteração.

```
public class ContaBancaria {  
    private double saldo;  
  
    // ... construtor e outros métodos ...  
  
    // Setter para saldo (com validação)  
    public void depositar(double valor) {  
        if (valor > 0) {  
            this.saldo += valor;  
            System.out.println("Depósito de " + valor + " realizado. Novo saldo: " + this.saldo);  
        } else {  
            System.out.println("Valor de depósito inválido.");  
        }  
    }  
}
```

4. Por que usar Getters e Setters em vez de Acesso Direto?

- **Segurança dos Dados:** Impede modificações indesejadas ou diretas nos atributos privados.
- **Validação:** Permitem adicionar lógica para verificar a integridade dos dados (ex: não permitir saldo negativo).
- **Flexibilidade:** A lógica interna pode mudar sem afetar o código externo que usa a classe.
- **Manutenibilidade:** Facilita a manutenção e evolução do código.

ⓘ Dica de Produtividade no IntelliJ: Use Alt + Insert para gerar construtores, getters e setters automaticamente!

Pilar 2: Herança

1

Herança: Reutilizando Código (DNA)

"Uma classe filho herda características e ações de uma classe pai."

Herança: Reutilizando Código

Classe Pai: Animal

Atributos: nome, idade, peso

Métodos: comer(), dormir()



extends

Classe Filha: Cachorro

Atributos:

- nome, idade, peso (de Animal)
- raça

Métodos:

- comer(), dormir() (de Animal)
- latir(), farejar()



Classe Filha: Gato

Atributos:

- nome, idade, peso (de Animal)
- corDoPelo

Métodos:

- comer(), dormir() (de Animal)
- miar(), arranhar()



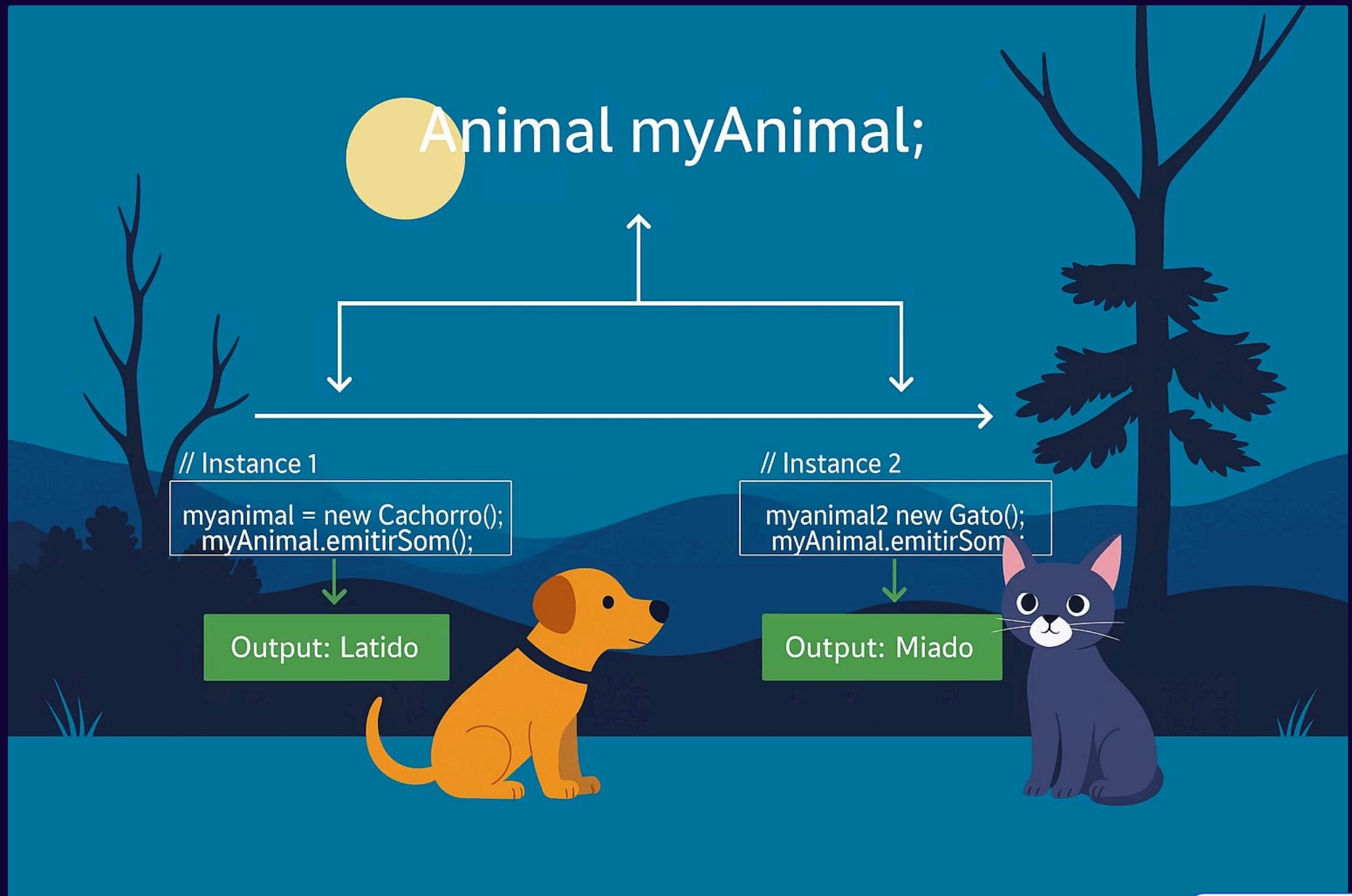
Membros herdados de **Animal** são mostrados em cinza, enquanto os **novos atributos e métodos**, específicos de cada classe, são destacados em verde.

Pilar 3: Polimorfismo

1

Polimorfismo: Muitas Formas, Uma Ação

Polimorfismo é a capacidade de um objeto ser referenciado de múltiplas formas. No nosso exemplo, tanto um Cachorro quanto um Gato podem ser tratados como um Animal. Isso nos permite escrever códigos mais flexíveis, onde a mesma ação `emitirSom()` pode ter comportamentos completamente diferentes dependendo do objeto que a executa.



A Anotação @Override: Garantindo a Sobrescrita Correta

A anotação `@Override` é uma ferramenta de segurança que garante que você está realmente sobrescrevendo um método da classe pai. Ela funciona como um "verificador" que:

- Confirma que o método existe na classe pai
- Verifica se a assinatura do método está correta
- Previne erros de digitação ou mudanças acidentais
- Torna o código mais legível e autodocumentado

Exemplo prático em Java:

```
class Animal {  
    public void emitirSom() {  
        System.out.println("Som genérico");  
    }  
}  
  
class Cachorro extends Animal {  
    @Override  
    public void emitirSom() {  
        System.out.println("Au au!");  
    }  
}
```

Por que usar `@Override`?

- **Segurança:** O compilador avisa se algo estiver errado
- **Clareza:** Deixa explícito que é uma sobrescrita intencional
- **Manutenção:** Facilita futuras alterações no código

Pilar 4: Abstração

Abstração: Escondendo os Detalhes, Expondo o Essencial

A Simplicidade do Controle Remoto

A Abstração foca no essencial: expor apenas as funcionalidades necessárias e esconder toda a complexidade interna.

O controle remoto é o exemplo perfeito. Ele oferece uma **interface simples** (os botões) para que você possa interagir com um sistema complexo (a TV), sem que precise entender sua eletrônica.

Em resumo, a abstração permite que você saiba **o que fazer**, sem precisar saber **como é feito**.



1

Parte 1: Abstração com abstract class

Uma **classe abstrata** é um "molde" híbrido. Ela pode ter métodos concretos (com código pronto, como `checarBateria()`) que são compartilhados pelos filhos, e métodos abstratos (sem código, como `ligar()`) que os filhos são **obrigados** a implementar.

```
public abstract class ControleRemotoAbstrato {  
  
    protected String modelo;  
  
    public ControleRemotoAbstrato(String modelo) {  
        this.modelo = modelo;  
    }  
  
    // --- MÉTODO CONCRETO ---  
    // Este método já vem pronto e é compartilhado por todos os filhos.  
    public void checarBateria() {  
        System.out.println("Bateria do modelo " + modelo + " está ok!");  
    }  
  
    // --- MÉTODOS ABSTRATOS ---  
    public abstract void ligar();  
    public abstract void desligar();  
}
```

```
public class ControleParaTV extends ControleRemotoAbstrato {  
  
    public ControleParaTV(String modelo) {  
        super(modelo); // chama o construtor da classe pai  
    }  
  
    // --- IMPLEMENTAÇÃO OBRIGATÓRIA ---  
    @Override  
    public void ligar() {  
        System.out.println("Ligando a TV com o controle " + this.modelo);  
    }  
  
    @Override  
    public void desligar() {  
        System.out.println("Desligando a TV...");  
    }  
}
```

Parte 2: Abstração com interface

Uma **interface** é um contrato 100% abstrato. Ela apenas define "o que" uma classe deve fazer (os métodos), mas nunca "como". É a ferramenta ideal quando classes muito diferentes precisam compartilhar as mesmas habilidades (os "botões" do controle remoto).

```
public interface ControleRemoto {  
  
    // Todos os métodos em uma interface são abstratos por padrão.  
    // Eles apenas definem "o que" fazer.  
    void ligar();  
    void desligar();  
}
```

```
// A classe Televisao "assina" o contrato  
public class Televisao implements ControleRemoto {  
  
    private boolean ligada;  
    private String modelo;  
  
    public Televisao(String modelo) {  
        this.modelo = modelo;  
        this.ligada = false;  
    }  
  
    @Override  
    public void ligar() {  
        this.ligada = true;  
        System.out.println("TV " + modelo + " ligada.");  
    }  
  
    @Override  
    public void desligar() {  
        this.ligada = false;  
        System.out.println("TV " + modelo + " desligada.");  
    }  
}
```

Muito Obrigado!

Filippo Chiarion

Estudante de Engenharia de Software

Vamos nos conectar?



LinkedIn

Filippo Chiarion



Email

filippochiarion02@gmail.com



GitHub

FilippoChiarion

Repositório GitHub contendo todos os códigos da aula

