

Despertar do Código

JAVA E A FORÇA OOP



Alisson Rangel



O.O. os 4 pilares com JAVA - ALISSON RANGEL

Despertando para o Mundo Orientado a Objetos

O que é POO e por que todo desenvolvedor Java precisa dominá-la

A Programação Orientada a Objetos (POO) é uma forma de estruturar programas pensando em objetos do mundo real. Em vez de trabalhar com funções soltas e dados separados, unimos tudo em classes que representam entidades com atributos (características) e métodos (ações).

Em Java, praticamente tudo gira em torno de objetos. Aprender a pensar em termos de classe, objeto, estado e comportamento é essencial para criar aplicações robustas, escaláveis e organizadas.

Agora, vamos entender os 4 pilares que formam a base da Orientação a Objetos.



O.O. os 4 pilares com JAVA - ALISSON RANGEL

01

Encapsulamento

**Esconda os detalhes e controle o
acesso aos dados**



O.O. os 4 pilares com JAVA - ALISSON RANGEL

Encapsulamento

Protegendo o Coração dos Seus Objetos

Encapsulamento é o pilar que trata da proteção e controle dos dados de um objeto. A ideia é simples: não permitir que qualquer parte do sistema acesse e altere diretamente os dados internos de um objeto. Para isso, usamos os modificadores `private`, `public` e os famosos `getters` e `setters`.



Encapsulamento

Exemplo: Sistema de alunos

```
Java - Encapsulamento

public class Aluno {
    private String nome;
    private double nota;

    public Aluno(String nome) {
        this.nome = nome;
    }

    public void setNota(double nota) {
        if (nota ≥ 0 && nota ≤ 10) {
            this.nota = nota;
        }
    }

    public double getNota() {
        return this.nota;
    }

    public String getNome() {
        return this.nome;
    }
}
```

Por que isso importa?

Se nota fosse pública, qualquer parte do sistema poderia colocar um valor inválido, como -3 ou 20. Com o encapsulamento, protegemos a integridade dos dados e centralizamos a lógica de validação dentro do próprio objeto.

02

Herança

**Evite duplicação e modele
relações do mundo real**



O.O. os 4 pilares com JAVA - ALISSON RANGEL

Herança

Reutilizando Comportamentos de Forma Inteligente

A herança permite que uma classe herde atributos e comportamentos de outra. Isso representa uma relação “é um” — por exemplo, um Cachorro é um Animal, um Administrador é um Usuário. Com herança, conseguimos reaproveitar código e organizar hierarquias de classes.



Herança

Exemplo: Usuários de um sistema web

```
Java - Herança

public class Usuario {
    String nome;
    public Usuario(String nome) {
        this.nome = nome;
    }
    public void login() {
        System.out.println(nome + " fez login.");
    }
}

public class Cliente extends Usuario {
    public Cliente(String nome) {
        super(nome);
    }
    public void comprarProduto() {
        System.out.println(nome + " comprou um produto.");
    }
}

public class Administrador extends Usuario {
    public Administrador(String nome) {
        super(nome);
    }
    public void acessarPainel() {
        System.out.println(nome + " acessou o painel.");
    }
}
```

Benefício:

Com herança, Cliente e Administrador reaproveitam o método login() da classe Usuario, sem duplicar código. Isso facilita a manutenção e torna o sistema mais lógico.

03

Polimorfismo

**Escreva código genérico e ganhe
flexibilidade**



O.O. os 4 pilares com JAVA - ALISSON RANGEL

Polimorfismo

Um Método, Muitos Comportamentos

Polimorfismo é a capacidade de um mesmo método se comportar de maneiras diferentes, dependendo do objeto que o invoca. Isso permite que você escreva código mais flexível e reutilizável.

Por exemplo, você pode ter uma classe `Animal` com um método `falar()`, e classes `Cachorro` e `Gato` que implementam esse método de forma diferente. Quando você chama `falar()` em um objeto `Animal`, o Java executa a versão correta de acordo com o tipo real do objeto. Isso é útil em listas de objetos diversos, onde você quer tratá-los de forma genérica. Em Java, isso é possível com herança e interfaces. Com o polimorfismo, seu código fica mais organizado e preparado para mudanças.

Polimorfismo

Exemplo:

```
Java - Polimorfismo

public class Animal {
    public void falar() {
        System.out.println("O animal faz um som.");
    }
}

public class Cachorro extends Animal {
    @Override
    public void falar() {
        System.out.println("O cachorro late: Au Au!");
    }
}

public class Gato extends Animal {
    @Override
    public void falar() {
        System.out.println("O gato mia: Miau!");
    }
}
```

Benefício:

O mesmo método falar() se comporta de forma diferente, dependendo do tipo real do objeto. Isso permite criar código genérico e modular, fácil de manter e escalar.

04

Abstração

**Modele comportamentos
comuns sem se preocupar com o
“como”**



Abstração

O Essencial Sem os Detalhes

Abstração é sobre ocultar a complexidade e expor apenas o que é necessário. Com ela, criamos classes e interfaces que definem comportamentos, mas não se preocupam com a implementação naquele momento.



Abstração

Exemplo: Sistema de pagamentos

```
Java - Polimorfismo

// Interface que representa um comportamento genérico de pagamento
public interface MetodoPagamento {
    // Método abstrato: só define o que deve ser feito, não como
    void pagar(double valor);
}

// Classe que implementa o pagamento com cartão
public class CartaoCredito implements MetodoPagamento {
    @Override
    public void pagar(double valor) {
        System.out.println("Pagando R$" + valor + " Cartão de Crédito.");
    }
}

// Classe que implementa o pagamento via Pix
public class Pix implements MetodoPagamento {
    @Override
    public void pagar(double valor) {
        System.out.println("Pagando R$" + valor + " via Pix.");
    }
}

// Classe que realiza o processo de pagamento, sem se importar com a forma
public class Checkout {
    public void processarPagamento(MetodoPagamento metodo, double valor) {
        // Aqui usamos a abstração: só chamamos "pagar", sem saber os detalhes
        metodo.pagar(valor);
    }
}

// Classe principal para executar o código
public class Main {
    public static void main(String[] args) {
        MetodoPagamento pagamento1 = new CartaoCredito();
        MetodoPagamento pagamento2 = new Pix();
        Checkout checkout = new Checkout();

        // Pagamento com cartão
        checkout.processarPagamento(pagamento1, 150.00);
        // Pagamento com Pix
        checkout.processarPagamento(pagamento2, 75.00);
    }
}
```


Abstração

O que esse exemplo mostra:

A interface MetodoPagamento define o que precisa ser feito (pagar), mas não define como.

As classes CartaoCredito e Pix implementam os detalhes específicos.

A classe Checkout usa abstração para processar qualquer tipo de pagamento, sem se preocupar com os detalhes. Isso garante baixo acoplamento e alta extensibilidade.

Essa separação torna o sistema mais flexível, extensível e fácil de manter.

Agradecimentos

Este ebook foi desenvolvido por Alisson Rangel como parte de uma atividade de um Bootcamp da DIO com a tutoria do especialista Felipe Aguiar.
Agradeço a DIO pela a oportunidade e ao Felipe Aguiar pela ministração do curso.



Obrigado

Esse Ebook foi gerado por IA, e diagramado por humano.

O passo a passo se encontra no meu Github.



<https://github.com/alissonrangel/prompts-recipe-to-create-a-ebook>

Autor: Alisson Rangel

<https://github.com/alissonrangel>

<https://www.linkedin.com/in/alisson-rangel/>



O.O. os 4 pilares com JAVA - ALISSON RANGEL