Curso Presencial Programação Fullstack

Aula 09

Prof. MSc. Kelson | Senior Software Engineer

PROFKELSON.DEV

BORA CODAR?

PROFKELSON.DEV
BORA CODAR?

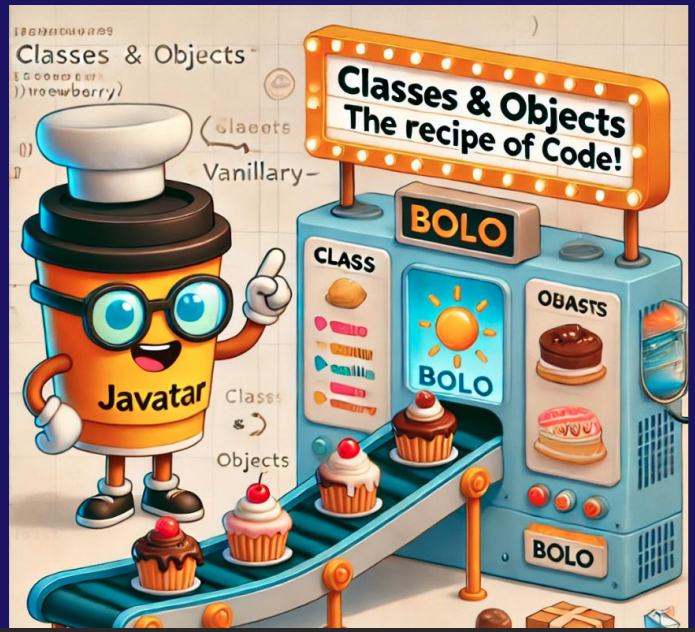
Java Básico Pré-Spring

- Bem-vindos ao Mundo do Java! ☕🚀
  O que você vai ver hoje?
  📌 1. **Programação Orientada a Objetos (POO)** – Classes, objetos, herança e mais! (Sim, vai ter bolo 🎂)
  📌 2. **Estruturas de Controle & Coleções** – Listas, mapas, laços de repetição e organização dos dados! 🔄📦
  📌 3. **Tratamento de Exceções** – Como lidar com erros sem pânico! ⚠️💥
  📌 4. **Sintaxe & Boas Práticas** – Como organizar seu código e evitar gambiarras! 🤓✅

Classes e Objetos - A Receita do Código! 🎂☕

💡 Pense em uma classe como uma receita de bolo!

- Ela não é um bolo, mas tem tudo que precisamos para fazer um. 🍰
- Define ingredientes (atributos) e modo de preparo (métodos).
- Sem a receita, não conseguimos fazer o bolo!




```java
public class Bolo {
    String sabor; // Ingrediente
    int tamanho;  // Tamanho do bolo

    void assar() { // Modo de preparo
        System.out.println("Assando um bolo de " + sabor + " de " + tamanho + " cm!");
    }
}
```

Classes e Objetos - A Receita do Código! 🎂☕

O que é um Objeto? 🎭

- Criamos objetos a partir de classes, como fazemos bolos com receitas.
- Cada objeto pode ter características diferentes.


```
Assando um bolo de Chocolate de 20 cm!
```

```java
Bolo boloChocolate = new Bolo();
boloChocolate.sabor = "Chocolate";
boloChocolate.tamanho = 20;
boloChocolate.assar();
```

Classes e Objetos – A Receita do Código! 🎂☕

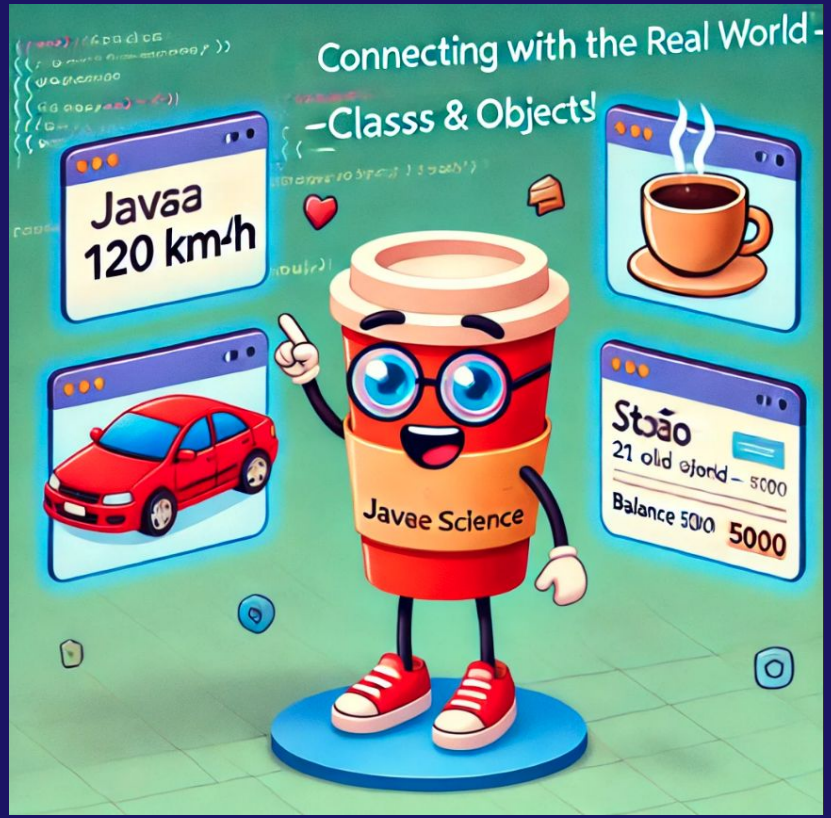
Conectando com o Mundo Real 🌎

📌 Uma classe pode representar:

✅ Um Carro (com cor, modelo, velocidade). 🚗

✅ Um Aluno (com nome, idade, curso). 🎓

✅ Uma Conta Bancária (com saldo, agência, titular). 🏦

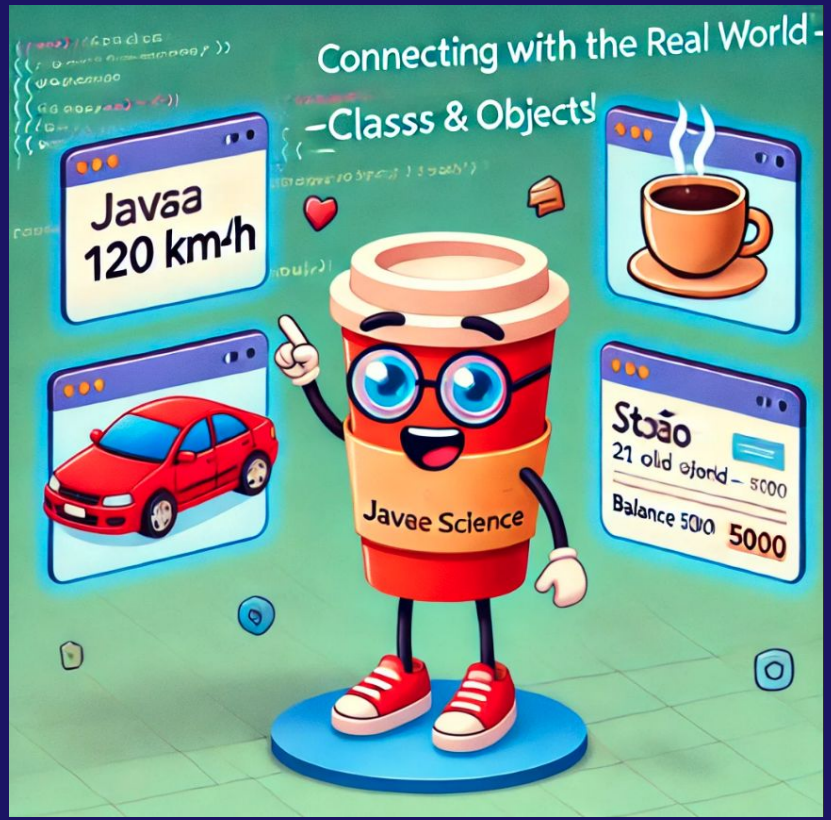


Classes e Objetos - A Receita do Código! 🎂☕

Conectando com o Mundo Real 🌎

📌 Os objetos são os exemplos reais dessas classes!

- Um carro vermelho, modelo sedan, 120 km/h.
- Um aluno João, 21 anos, cursando Ciência da Computação.
- Uma conta bancária com saldo de R$ 5000.



Classes e Objetos - A Receita do Código! 🎂☕

Como a Receita Ganha Vida? 🏭✨

1️⃣ A classe (receita) é criada! 📜

2️⃣ A receita entra na máquina (instanciação de objeto)! 🔄

3️⃣ Os bolos saem prontos (objetos são criados)! 🎂🎂🎂

```java
// Definição da classe Bolo (Receita)
public class Bolo {
    String sabor;
    int tamanho;

    // Construtor para definir os atributos ao criar um objeto
    public Bolo(String sabor, int tamanho) {
        this.sabor = sabor;
        this.tamanho = tamanho;
    }

    // Método para exibir os detalhes do bolo
    public void exibirDetalhes() {
        System.out.println("Bolo de " + sabor + " com " + tamanho + " cm pronto!");
    }
}
```

Classes e Objetos - A Receita do Código! 🎂☕

Como a Receita Ganha Vida? 🏭✨

1️⃣ A classe (receita) é criada! 📜

2️⃣ A receita entra na máquina (instanciação de objeto)! 🔄

3️⃣ Os bolos saem prontos (objetos são criados)! 🎂🎂🎂

```java
// Criando uma "máquina" que gera bolos (Fábrica de Objetos)
public class MaquinaDeBolos {
    public Bolo fabricarBolo(String sabor, int tamanho) {
        System.out.println("Preparando um bolo de " + sabor + "...");
        return new Bolo(sabor, tamanho);
    }
}

```

Classes e Objetos - A Receita do Código! 🎂☕

Como a Receita Ganha Vida? 🏭✨

1. A classe (receita) é criada! 📜
2. A receita entra na máquina (instanciação de objeto)! 🔄
3. Os bolos saem prontos (objetos são criados)! 🎂🎂🎂

```java
public class Padaria {
    public static void main(String[] args) {
        // Criando uma máquina de bolos
        MaquinaDeBolos maquina = new MaquinaDeBolos();

        // Instanciando objetos (criando bolos)
        Bolo boloChocolate = maquina.fabricarBolo("Chocolate", 20);
        Bolo boloBaunilha = maquina.fabricarBolo("Baunilha", 15);
        Bolo boloMorango = maquina.fabricarBolo("Morango", 25);

        // Exibindo os bolos criados
        boloChocolate.exibirDetalhes();
        boloBaunilha.exibirDetalhes();
        boloMorango.exibirDetalhes();
    }
}
```

```
Preparando um bolo de Chocolate...
Preparando um bolo de Baunilha...
Preparando um bolo de Morango...
Bolo de Chocolate com 20 cm pronto!
Bolo de Baunilha com 15 cm pronto!
Bolo de Morango com 25 cm pronto!
```

Mini Projeto 1 (Com o prof)

Você foi contratado para desenvolver um sistema simples para uma padaria. A padaria vende diferentes tipos de bolos, e cada bolo tem um sabor, tamanho e preço. O sistema deve permitir:

✅ Criar bolos com diferentes características.
✅ Exibir os detalhes dos bolos criados.
✅ Simular uma máquina que fabrica bolos.

💡 Dica: Pense na classe Bolo como a receita e nos objetos como os bolos sendo preparados!

Mini Projeto 1.1 (Sozinho(a))

Seu desafio agora é criar um sistema para uma Loja de Sorvetes. A loja vende diferentes tipos de sorvete, cada um com sabor, tamanho e preço.

O sistema deve permitir:

✅ Criar sorvetes com diferentes características.
✅ Exibir os detalhes dos sorvetes criados.
✅ Simular uma máquina que fabrica os sorvetes.

📝 Dica: O funcionamento deve ser parecido com a Padaria, mas agora com sorvetes! 😋

```
Café protegido! Você só pode acessá-lo pelos métodos certos.
```

Os 4 Pilares da Programação Orientada a Objetos ☕🚀

1. **Encapsulamento** – Protegendo o Café! ☕🔒

📌 O que é?

- Encapsulamento protege os dados de um objeto e controla seu acesso.
- Imagine que os atributos da classe são o café dentro do copo e a tampa do café impede o acesso direto!
- Apenas métodos específicos (getters e setters) podem abrir essa tampa e permitir alterações.

```java
public class Cafe {
    private String tipo;  // O café está encapsulado!
    private double temperatura;

    // Construtor
    public Cafe(String tipo, double temperatura) {
        this.tipo = tipo;
        this.temperatura = temperatura;
    }

    // Método GETTER para acessar o tipo de café
    public String getTipo() {
        return tipo;
    }

    // Método SETTER para modificar a temperatura com segurança
    public void setTemperatura(double temperatura) {
        if (temperatura > 0) {
            this.temperatura = temperatura;
        }
    }
}
```

```
Servindo um delicioso Café com Leite
```

Os 4 Pilares da Programação Orientada a Objetos ☕🚀

🧬 2. **Herança** – O Sabor é Passado Adiante! ☕➡️

📌 O que é?

- Herança permite que uma classe "filha" reutilize código de uma classe "pai".
- Assim como diferentes tipos de café herdam o sabor do café original, as classes filhas herdam os métodos e atributos da classe pai!

```java
// Classe pai (Café)
public class Cafe {
    String tipo = "Café Puro";

    public void servir() {
        System.out.println("Servindo um delicioso " + tipo);
    }
}

// Classe filha (Café com leite)
public class CafeComLeite extends Cafe {
    public CafeComLeite() {
        tipo = "Café com Leite";
    }
}

// Classe principal
public class Cafeteria {
    public static void main(String[] args) {
        CafeComLeite meuCafe = new CafeComLeite();
        meuCafe.servir(); // Método herdado da classe pai
    }
}
```

Os 4 Pilares da Programação Orientada a Objetos

🎭 3. **Polimorfismo** – Escolha sua Bebida! 🍵☕🥤

📌 O que é?

- Polimorfismo permite que objetos diferentes respondam de maneiras diferentes ao mesmo método!
- Assim como você pode pedir uma bebida e escolher entre café, chá ou suco, um mesmo método pode ter várias implementações!

```
Preparando um café!
Preparando um chá!
```

```java
// Classe Pai
public class Bebida {
    public void preparar() {
        System.out.println("Preparando uma bebida...");
    }
}

// Classes Filhas (Polimorfismo)
public class Cafe extends Bebida {
    @Override
    public void preparar() {
        System.out.println("Preparando um café!");
    }
}

public class Cha extends Bebida {
    @Override
    public void preparar() {
        System.out.println("Preparando um chá!");
    }
}

// Classe Principal
public class Pedido {
    public static void main(String[] args) {
        Bebida minhaBebida1 = new Cafe();
        Bebida minhaBebida2 = new Cha();

        minhaBebida1.preparar(); // Saída: Preparando um café!
        minhaBebida2.preparar(); // Saída: Preparando um chá!
    }
}
```

Os 4 Pilares da Programação Orientada a Objetos ☕🚀

🎩 4. **Abstração** – O Mistério do Café!
🎭☕

📌 O que é?

- Abstração oculta detalhes complexos e mostra apenas o necessário!
- Você bebe um café sem precisar saber como ele foi feito, apenas usa um método para prepará-lo!

```
// Classe abstrata
abstract class MaquinaDeCafe {
    // Método abstrato (cada tipo de café implementa do seu jeito)
    abstract void prepararCafe();

    // Método comum para todas as máquinas
    public void iniciar() {
        System.out.println("Ligando a máquina de café...");
    }
}

// Classe filha implementando a abstração
class Expresso extends MaquinaDeCafe {
    @Override
    public void prepararCafe() {
        System.out.println("Preparando um café expresso!");
    }
}

// Classe Principal
public class Cafeteria {
    public static void main(String[] args) {
        Expresso minhaMaquina = new Expresso();
        minhaMaquina.iniciar();
        minhaMaquina.prepararCafe();
    }
}
```

```
css

Ligando a máquina de café...
Preparando um café expresso!
```

Os 4 Pilares da Programação Orientada a Objetos ☕🚀

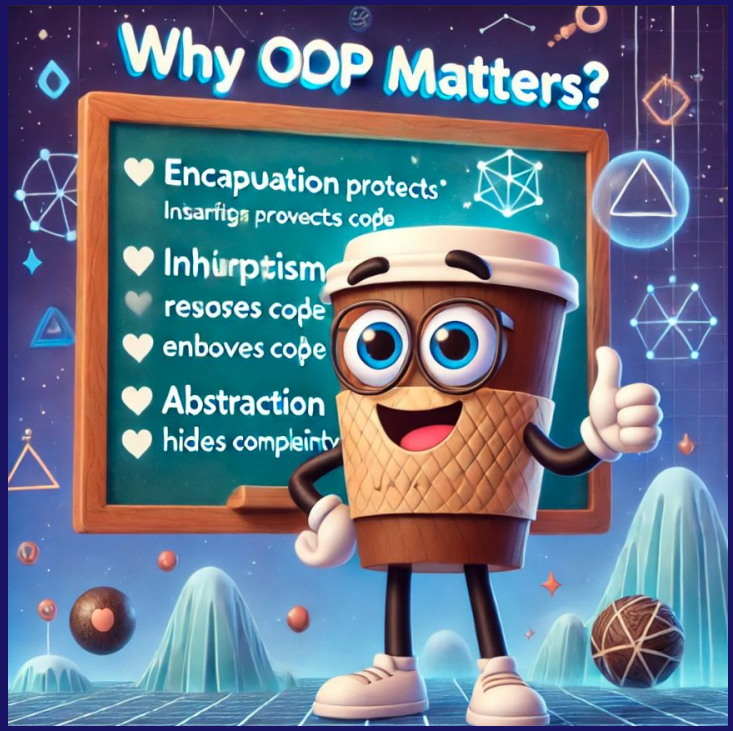
🎯 Conclusão: Por que isso é importante?

✅ **Encapsulamento** protege os dados e controla acessos.

✅ **Herança** reutiliza código e facilita a manutenção.

✅ **Polimorfismo** permite que um mesmo método funcione de formas diferentes.

✅ **Abstração** simplifica o código e esconde detalhes desnecessários.

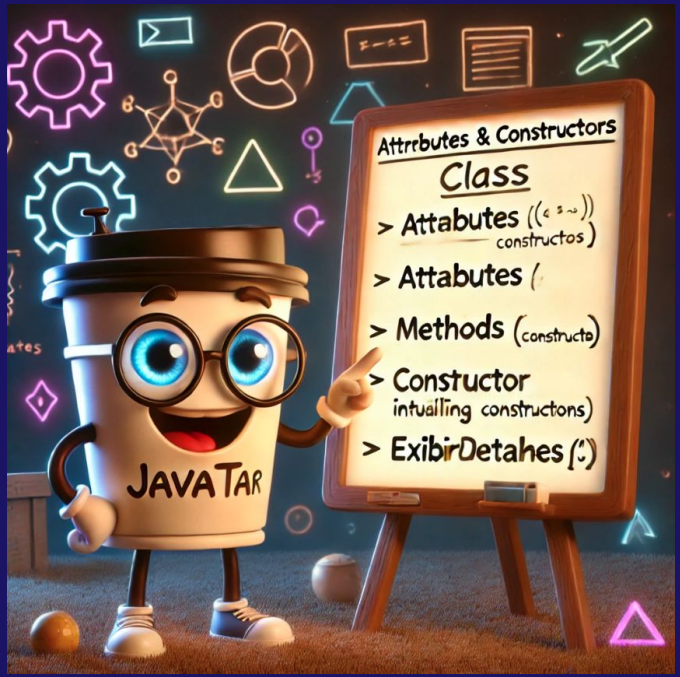


Atributos, Métodos e Construtores – O Coração da POO! 🏗️✨

💡 O que torna uma classe funcional?
💡 Uma classe sem atributos, métodos e construtores é como uma receita vazia! 📜🔧
Para que uma classe realmente tenha vida, ela precisa de:

✅ Atributos – Armazenam informações sobre o objeto. 📦
✅ Construtores – Criam e inicializam os objetos. 🏗️
✅ Métodos – Definem ações e comportamentos do objeto. ✨

📌 Agora, vamos ver como tudo isso funciona! 🚀



Atributos, Métodos e Construtores – O Coração da POO! 🏗️✨

🔹 1. **Atributos** – Definindo as Informações do Objeto

📌 O que são?

- Atributos são as características do objeto, armazenadas dentro da classe.
- Se a classe fosse uma ficha de cadastro, os atributos seriam os campos preenchidos.

```java
public class Produto {
    String nome;        // Nome do produto
    double preco;       // Preço do produto
    String categoria;   // Categoria do produto
}

```

Atributos, Métodos e Construtores – O Coração da POO! 🏗️✨

🔹 2. **Construtores** – Dando Vida ao Objeto! 🏗️✨

📌 O que é?

- Um construtor é um método especial que é chamado automaticamente quando um objeto é criado.
- Ele define valores iniciais para os atributos do objeto.

```java
public class Produto {
    String nome;
    double preco;
    String categoria;

    // Construtor inicializando os atributos
    public Produto(String nome, double preco, String categoria) {
        this.nome = nome;
        this.preco = preco;
        this.categoria = categoria;
    }
}
```

```java
Produto p1 = new Produto("Mouse Gamer", 199.99, "Periféricos");
Produto p2 = new Produto("Teclado Mecânico", 349.99, "Acessórios");
```

```
mathematica                                                    Copiar  Editar
📦 Produto: Monitor Full HD | Preço: R$ 899.99 | Categoria: Periféricos
```

Atributos, Métodos e Construtores – O Coração da POO! 🏗️✨

🔹 3. **Métodos** – Fazendo a Mágica Acontecer! ✨

📌 O que são?

- Métodos são ações que um objeto pode executar.
- Eles fazem o objeto interagir com o mundo! 🌍

```java
public class Produto {
    String nome;
    double preco;
    String categoria;

    public Produto(String nome, double preco, String categoria) {
        this.nome = nome;
        this.preco = preco;
        this.categoria = categoria;
    }

    // Método para exibir detalhes do produto
    public void exibirDetalhes() {
        System.out.println("📦 Produto: " + nome + " | Preço: R$ "
                           + preco + " | Categoria: " + categoria);
    }
}
|
```

```java
Produto p1 = new Produto("Monitor Full HD", 899.99, "Periféricos");
p1.exibirDetalhes();

```

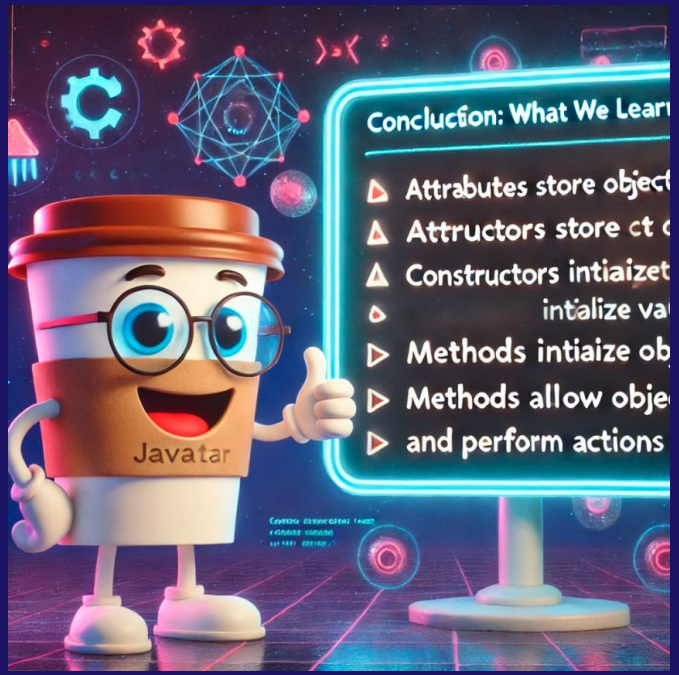
Atributos, Métodos e Construtores - O Coração da POO! 🏗️✨

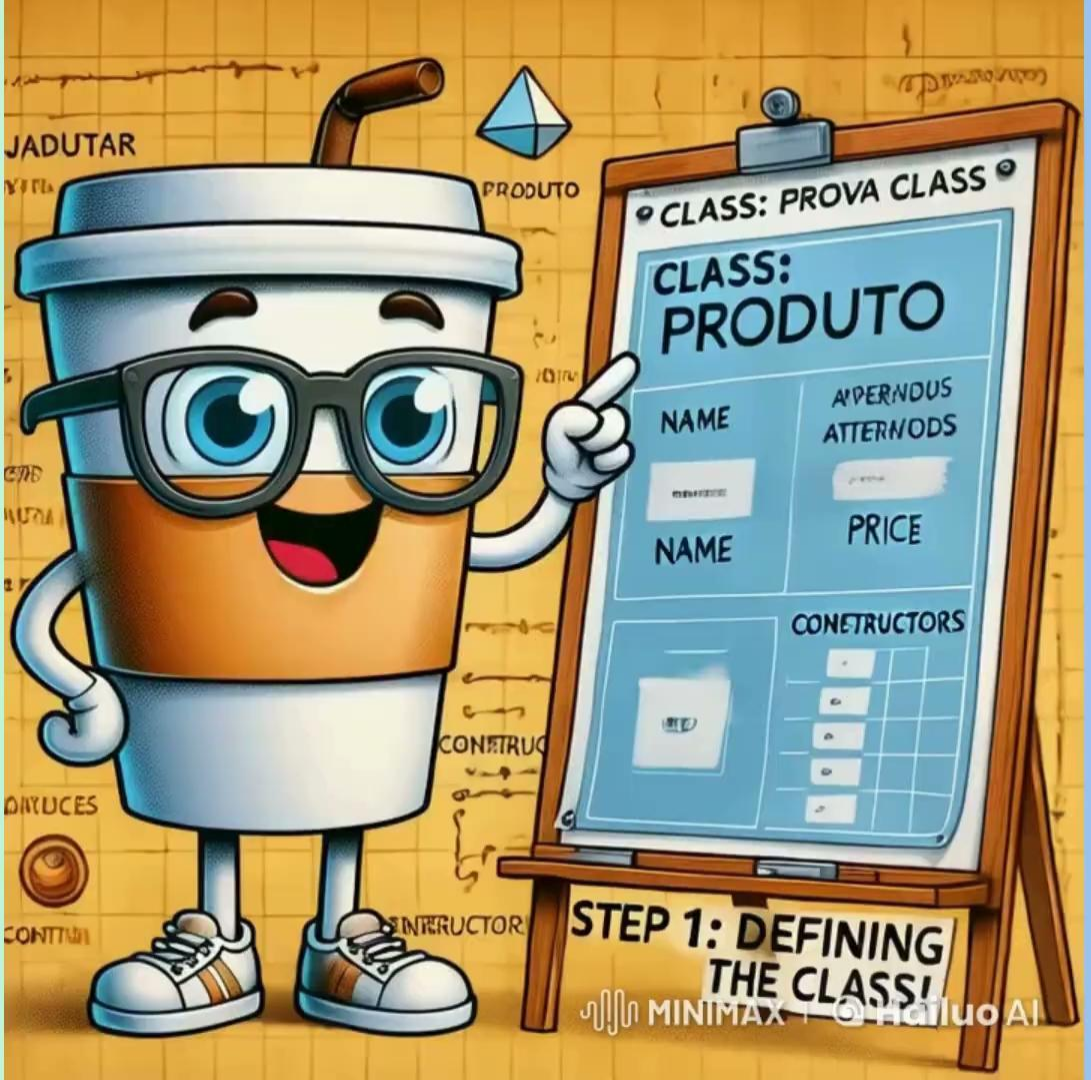
🎯 **Conclusão**: O que aprendemos?

✅ Atributos guardam os dados do objeto.

✅ Construtores inicializam o objeto com valores.

✅ Métodos fazem o objeto interagir e realizar ações.





PROFKELSON.DEV
BORA CODAR?

Mini Projeto 2 (Com o prof)

Você foi contratado para desenvolver um sistema de Cadastro de Produtos para uma loja.

O sistema deve permitir:
✅ Criar produtos com nome, preço e categoria.
✅ Exibir os detalhes do produto.
✅ Aplicar desconto ao preço do produto.

📌 Dica: Pense na classe Produto como um modelo, e os objetos serão os produtos cadastrados!

Mini Projeto 2.2 (Sozinho(a))

VVocê foi contratado para desenvolver um sistema de Cadastro de Funcionários para uma empresa.

O sistema deve permitir:
✅ Criar funcionários com nome, cargo e salário.
✅ Exibir os detalhes do funcionário.
✅ Aplicar um aumento de salário.

📌 Dica: O funcionamento é parecido com o Cadastro de Produtos, mas agora com funcionários!

Os 4 Pilares da Programação Orientada a Objetos ☕🚀

O que são os **Pilares da POO**? 🤔
💡 Os pilares da POO são como os segredos de um bom café! ☕
Eles ajudam a organizar e estruturar o código para que ele **seja mais reutilizável**, seguro e fácil de entender!

📌 Cada pilar tem uma função essencial no código, e o Javatar vai explicar cada um deles! 🚀

Interfaces - O Contrato do Código!

📜✨

💡 O que são Interfaces?

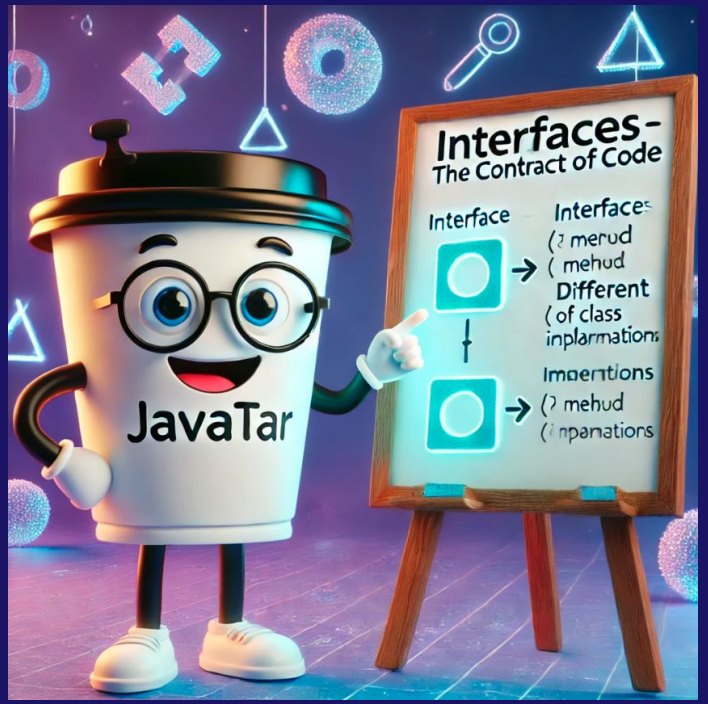
💡 Uma interface é como um contrato! 📄

- Define um conjunto de métodos que uma classe precisa implementar.
- A classe se compromete a seguir esse contrato, mas decide como cada método será implementado.

📌 Exemplo do Mundo Real:

Imagine um tomada universal. 🔌⚡

- Qualquer aparelho pode ser plugado, mas cada aparelho funciona de um jeito!



Interfaces - O Contrato do Código!

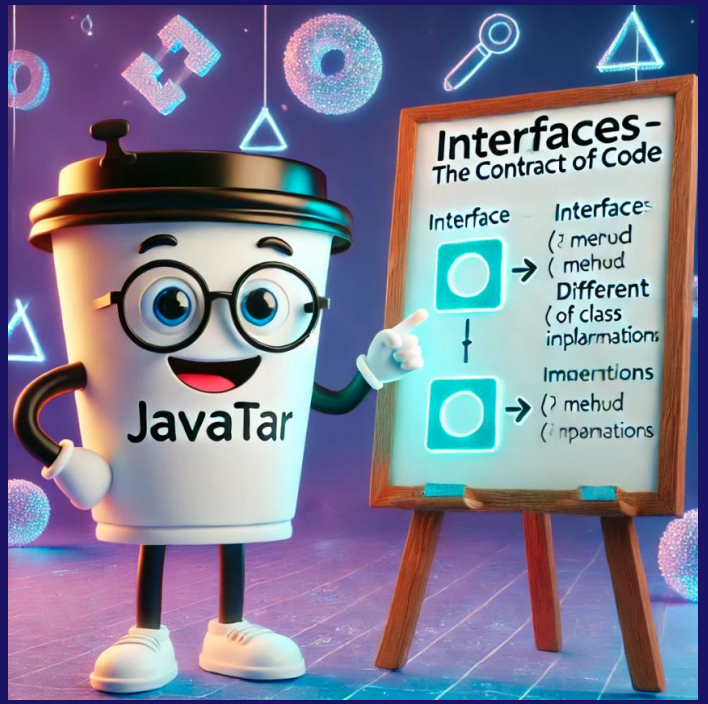
🛠️ Como Funciona?

1️⃣ Criando a Interface (O Contrato)

📌 A interface define os métodos, mas não os implementa.

```java
// Criando a interface Pagamento (Contrato)
public interface Pagamento {
    void processarPagamento(double valor);
}

```

📌 Aqui estamos dizendo que TODA classe que implementar Pagamento **precisa** ter um método processarPagamento().



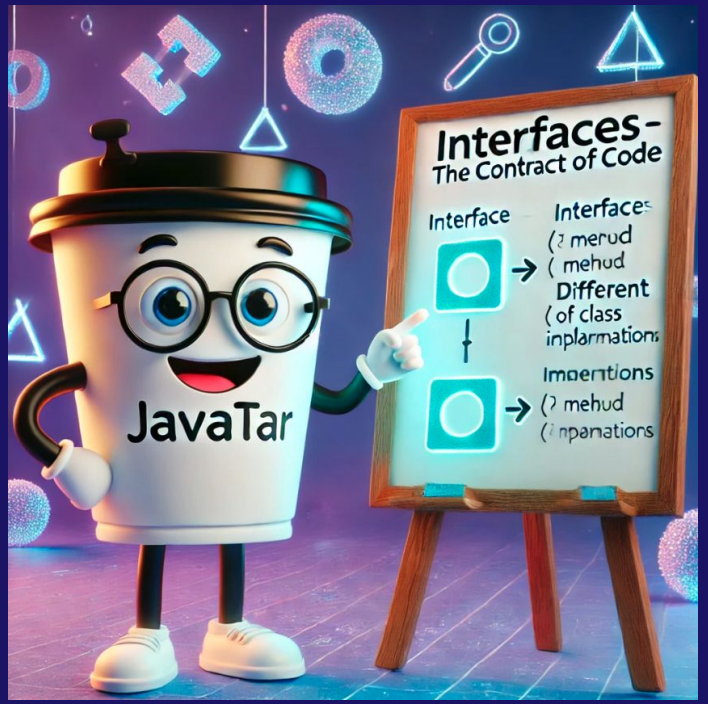
Interfaces - O Contrato do Código!

2️⃣ Implementando a Interface (Definindo a Ação!)

📌 Cada classe implementa os métodos da sua maneira.

```java
// Implementação para pagamento com Cartão
public class PagamentoCartao implements Pagamento {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("💳 Pagamento de R$ " + valor + " processado via Cartão.");
    }
}

// Implementação para pagamento com Pix
public class PagamentoPix implements Pagamento {
    @Override
    public void processarPagamento(double valor) {
        System.out.println("⚡ Pagamento de R$ " + valor + " processado via Pix.");
    }
}
```

📌 Ambas as classes seguem o contrato, mas cada uma tem sua própria implementação!



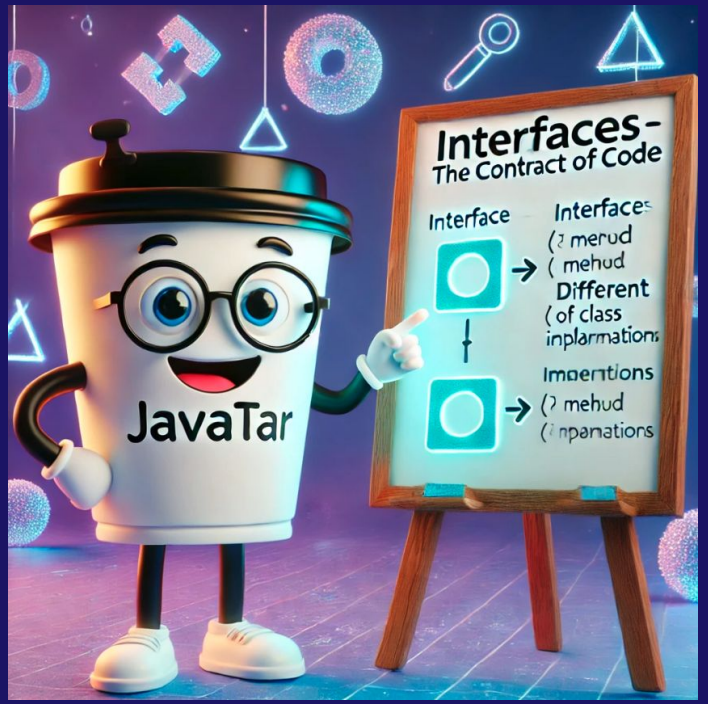
Interfaces - O Contrato do Código!

3️⃣ Usando a Interface (A Magia da Flexibilidade!)

📌 Podemos usar a interface para chamar qualquer tipo de pagamento, sem precisar mudar o código!

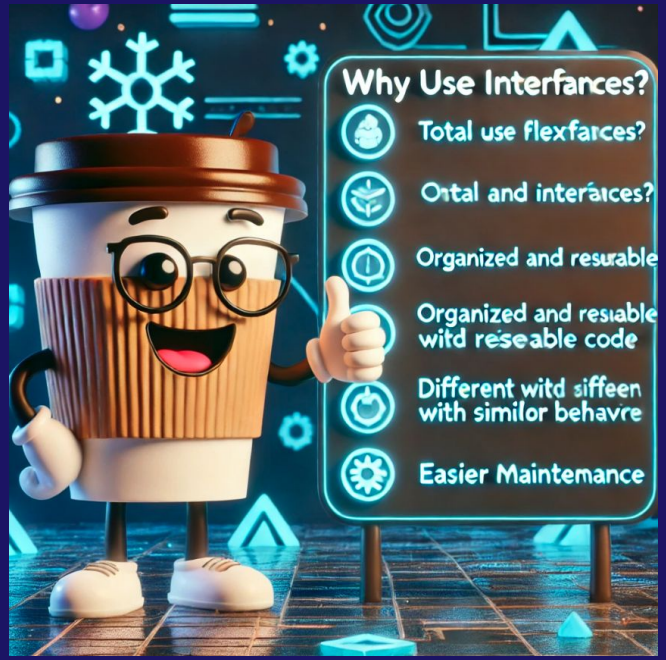
```java
public class Loja {
    public static void main(String[] args) {
        // Criando métodos de pagamento
        Pagamento pagamento1 = new PagamentoCartao();
        Pagamento pagamento2 = new PagamentoPix();

        // Processando pagamentos
        pagamento1.processarPagamento(250.00);
        pagamento2.processarPagamento(150.00);
    }
}
```

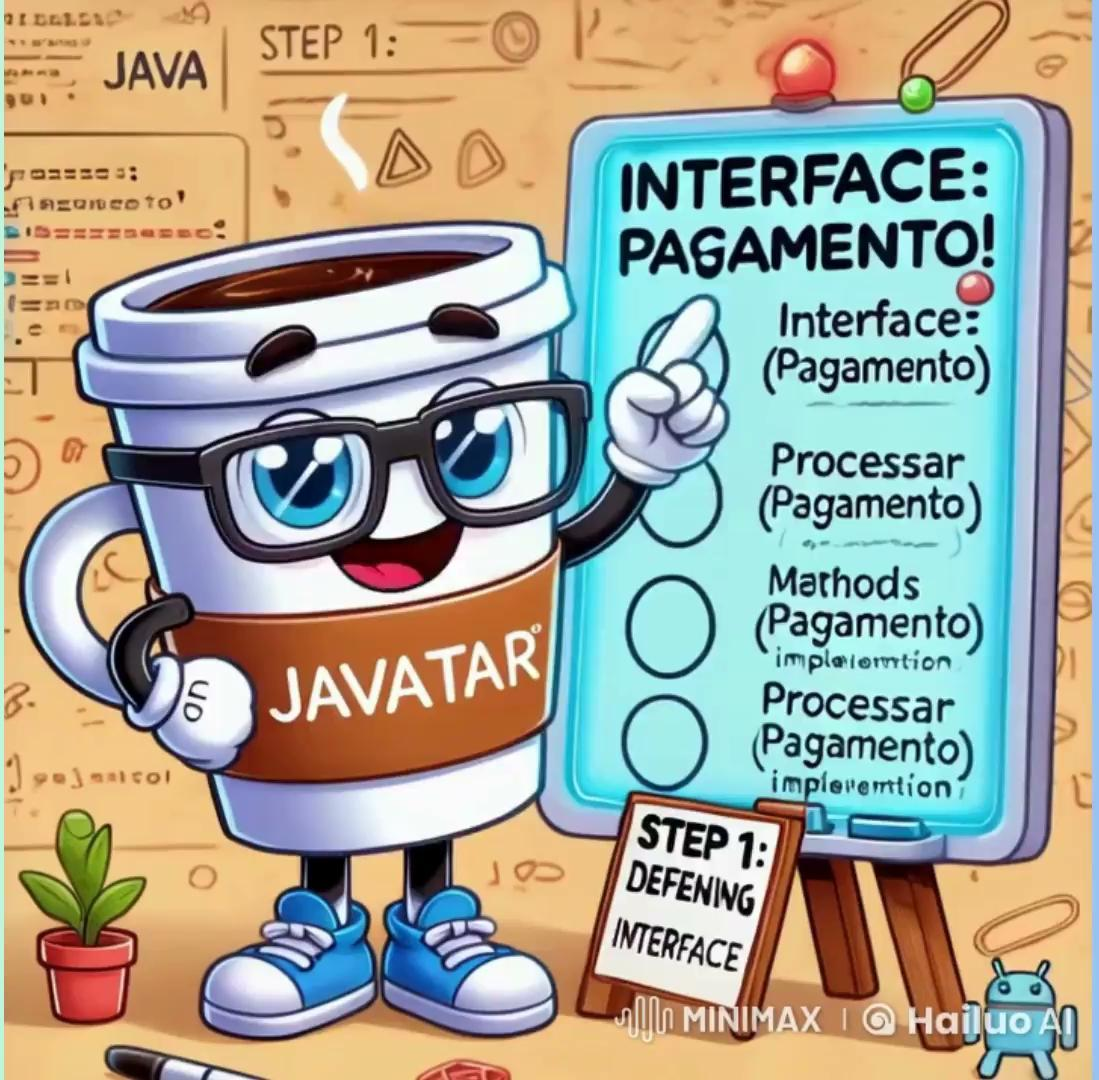
```
💳 Pagamento de R$ 250.0 processado via Cartão.
⚡ Pagamento de R$ 150.0 processado via Pix.
```



Interfaces - O Contrato do Código!

🎯 Por que usar Interfaces?
✅ Flexibilidade total! Podemos trocar implementações sem mudar o sistema.
✅ Código mais organizado e reutilizável!
✅ Facilita a manutenção e evita repetição de código.
✅ Permite criar classes diferentes com comportamentos semelhantes.





PROFKELSON.DEV
BORA CODAR?

Mini Projeto 3 (Com o prof)

Você foi contratado para criar um Sistema de Notificações 📢 que pode enviar mensagens por diferentes canais, como E-mail, SMS e WhatsApp.

O sistema deve permitir:
✅ Criar uma interface Notificacao que define um método enviarMensagem().
✅ Criar diferentes classes que implementam essa interface, representando E-mail, SMS e WhatsApp.
✅ Permitir que o usuário escolha o tipo de notificação e envie a mensagem sem alterar o código principal!

📌 Dica: Pense na interface como um contrato 📜 e nas classes como diferentes formas de enviar notificações!

Mini Projeto 3.1 (Sozinho(a))

Você foi contratado para desenvolver um Sistema de Pagamentos, onde o usuário pode escolher diferentes formas de pagamento:

✅ Criar uma interface Pagamento que define um método realizarPagamento().
✅ Criar diferentes classes que implementam essa interface, representando Cartão de Crédito, Boleto Bancário e Pix.
✅ Permitir que o usuário escolha o tipo de pagamento e efetue o pagamento sem alterar o código principal!

📌 Dica: O funcionamento é parecido com o Sistema de Notificações, mas agora com pagamentos!

Loops e Coleções - Repetindo Tarefas e Organizando Dados! 🔄📦

💡 O que são Loops?
💡 Loops são como baristas automáticos preparando café! ☕🔄
Eles permitem repetir tarefas automaticamente, sem precisar escrever o mesmo código várias vezes.

📌 Tipos de loops mais usados em Java:
✅ for – Ideal quando sabemos quantas vezes repetir.
✅ while – Repete enquanto uma condição for verdadeira.
✅ do-while – Sempre executa pelo menos uma vez.

Loops e Coleções - Repetindo Tarefas e Organizando Dados! 🔄📦

🛠️ Como Funcionam os Loops?

1️⃣ O Loop for – Repetições Controladas 🔢

📌 Quando usar? Quando sabemos quantas vezes queremos repetir algo.

```java
// Imprime os números de 1 a 5 usando um loop for
for (int i = 1; i <= 5; i++) {
    System.out.println("Café número: " + i);
}

```

```
Café número: 1
Café número: 2
Café número: 3
Café número: 4
Café número: 5
```

Loops e Coleções - Repetindo Tarefas e Organizando Dados! 🔄📦

🛠️ Como Funcionam os Loops?

2️⃣ O Loop while – Repetições Condicionais 🔁

📌 Quando usar? Quando não sabemos exatamente quantas vezes repetir, mas temos uma condição.

```java
int estoqueCafe = 3;
while (estoqueCafe > 0) {
    System.out.println("Servindo um café... Restam " + estoqueCafe);
    estoqueCafe--;
}

```

```
Servindo um café... Restam 3
Servindo um café... Restam 2
Servindo um café... Restam 1
```

Loops e Coleções - Repetindo Tarefas e Organizando Dados! 🔄📦

🛠️ Como Funcionam os Loops?

3️⃣ O Loop do-while – Faz pelo Menos Uma Vez! 🔄☕

📌 Quando usar? Quando queremos garantir que o código execute pelo menos uma vez, mesmo se a condição já for falsa.

```java
int cafe = 0;
do {
    System.out.println("Preparando um café...");
} while (cafe > 0);
```

```
Preparando um café...
```

Loops e Coleções - Repetindo Tarefas e Organizando Dados! 🔄📦

🛠️ Trabalhando com Coleções – Lists, Sets e Maps 📦

4️⃣ List – Armazenando Muitos Itens em Ordem 📜

📌 Quando usar? Quando precisamos armazenar múltiplos elementos na ordem em que foram adicionados.

```java
import java.util.ArrayList;
import java.util.List;

public class ExemploLista {
    public static void main(String[] args) {
        List<String> cardapio = new ArrayList<>();

        // Adicionando itens à lista
        cardapio.add("Café Expresso");
        cardapio.add("Cappuccino");
        cardapio.add("Latte");

        // Percorrendo a lista com foreach
        for (String item : cardapio) {
            System.out.println("☕ Opção no cardápio: " + item);
        }
    }
}
```

```
☕ Opção no cardápio: Café Expresso
☕ Opção no cardápio: Cappuccino
☕ Opção no cardápio: Latte
```

Loops e Coleções - Repetindo Tarefas e Organizando Dados! 🔄📦

🛠️ Trabalhando com Coleções – Lists, Sets e Maps 📦

5️⃣ Set – Armazenando Itens Únicos 🌍
📌 Quando usar? Quando precisamos armazenar elementos sem repetição.

O Set não permite elementos duplicados! 🚫

```java
import java.util.HashSet;
import java.util.Set;

public class ExemploSet {
    public static void main(String[] args) {
        Set<String> pedidosUnicos = new HashSet<>();

        // Adicionando itens ao Set
        pedidosUnicos.add("Café Expresso");
        pedidosUnicos.add("Cappuccino");
        pedidosUnicos.add("Cappuccino"); // Repetido, não será adicionado

        // Percorrendo o Set
        for (String pedido : pedidosUnicos) {
            System.out.println("📦 Pedido único: " + pedido);
        }
    }
}
```

```
📦 Pedido único: Café Expresso
📦 Pedido único: Cappuccino
```

Loops e Coleções - Repetindo Tarefas e Organizando Dados! 🔄📦

🛠️ Trabalhando com Coleções – Lists, Sets e Maps 📦

6️⃣ Map – Criando Relações Chave-Valor 🔑

📌 Quando usar? Quando queremos armazenar pares de chave-valor, como preços de produtos.

📌 Com Map, associamos chaves (nome do café) a valores (preço).

```java
import java.util.HashMap;
import java.util.Map;

public class ExemploMap {
    public static void main(String[] args) {
        Map<String, Double> precosCafe = new HashMap<>();

        // Adicionando preços ao mapa
        precosCafe.put("Café Expresso", 5.99);
        precosCafe.put("Cappuccino", 7.99);
        precosCafe.put("Latte", 6.99);

        // Percorrendo o Map
        for (String cafe : precosCafe.keySet()) {
            System.out.println("☕ " + cafe + " custa R$ " + precosCafe.get(cafe));
        }
    }
}
```

```
☕ Café Expresso custa R$ 5.99
☕ Cappuccino custa R$ 7.99
☕ Latte custa R$ 6.99
```

Loops e Coleções - Repetindo Tarefas e Organizando Dados! 🔄📦

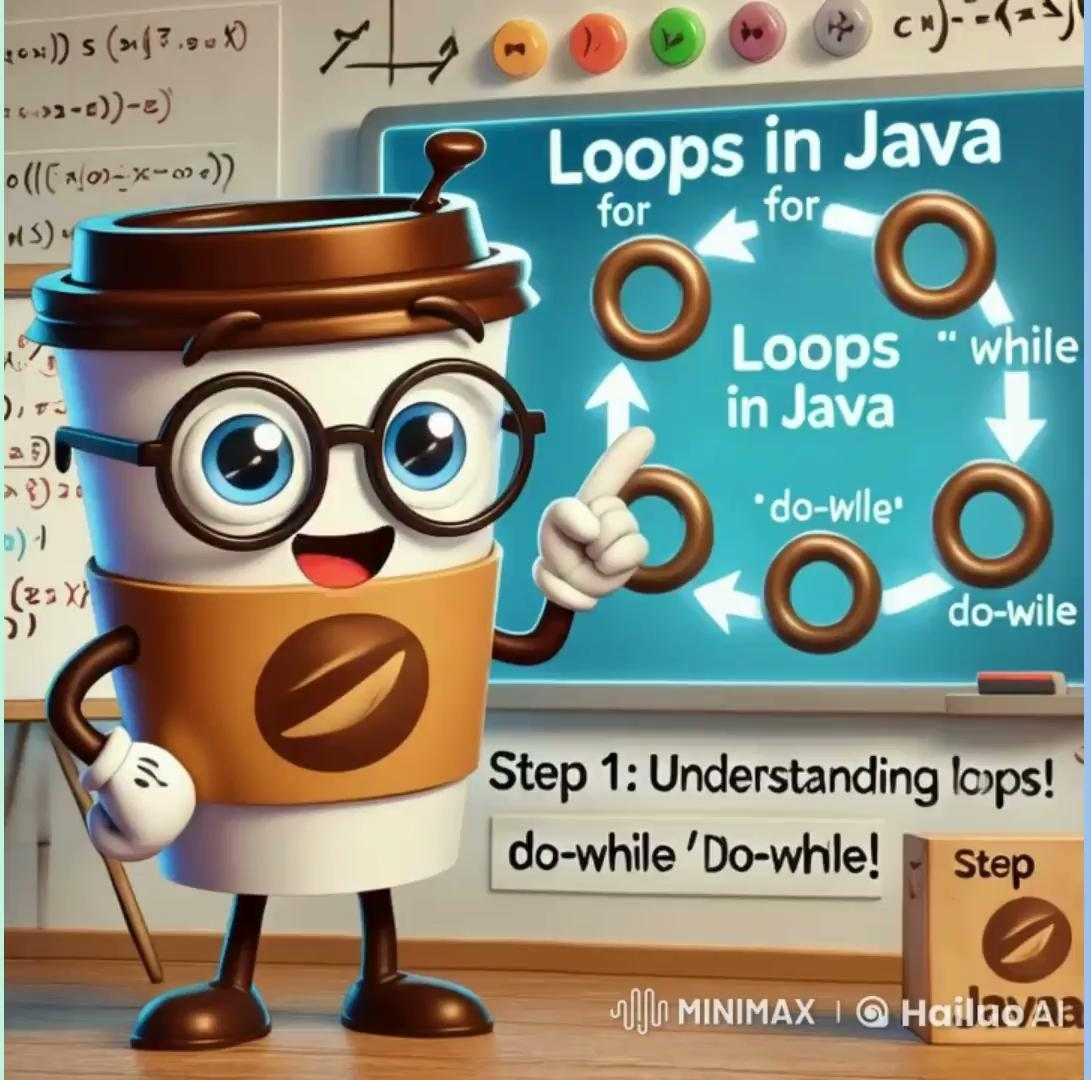
🎯 Conclusão: O que aprendemos?

✅ Loops nos permitem repetir tarefas automaticamente.

✅ List armazena itens ordenados.

✅ Set armazena itens únicos.

✅ Map armazena pares de chave-valor.



PROFKELSON.DEV
BORA CODAR?

Mini Projeto 4 (Com o prof)

Você foi contratado para criar um Gerenciador de Tarefas, onde o usuário pode:

✅ Adicionar tarefas a uma lista.
✅ Exibir todas as tarefas utilizando foreach.
✅ Filtrar tarefas concluídas utilizando Streams.

📌 Dica: O funcionamento será parecido com uma lista de afazeres (To-Do List)!

Mini Projeto 4.1 (Sozinho(a))

Você foi contratado para desenvolver um sistema de Lista de Compras, onde o usuário pode:

✅ Adicionar itens à lista de compras.
✅ Exibir todos os itens da lista utilizando foreach.
✅ Filtrar os itens acima de um determinado preço utilizando Streams.

📌 Dica: O funcionamento será parecido com uma lista de mercado, onde cada item tem um nome e um preço!

Tratamento de Exceções - Lidando com Erros no Código! ⚠️🚀

💡 O que são Exceções?
💡 Exceções são como derramar café no teclado! ☕💥
Elas ocorrem quando algo inesperado acontece no código, como:
✅ Tentar dividir por zero ➗❌
✅ Acessar um índice inexistente em uma lista 📦❌
✅ Conectar a um banco de dados fora do ar 🌐🚫

📌 Java permite tratar essas situações para evitar que o programa quebre!

Tratamento de Exceções - Lidando com Erros no Código! ⚠️🚀

⚡ Tipos de Exceções em Java

1️⃣ Checked Exceptions (Exceções Checadas) ✅

📌 O que são?

Erros previstos que o compilador obriga você a tratar.
Exemplo: Falha ao abrir um arquivo.

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ExemploChecked {
    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("arquivo.txt"));
            System.out.println("Arquivo lido com sucesso!");
        } catch (FileNotFoundException e) {
            System.out.println("❌ Erro: Arquivo não encontrado!");
        }
    }
}

```

```
❌ Erro: Arquivo não encontrado!
```

Tratamento de Exceções - Lidando com Erros no Código! ⚠️🚀

⚡ Tipos de Exceções em Java

2️⃣ Unchecked Exceptions (Exceções Não Checadas) ❌

📌 O que são?

Erros inesperados que podem acontecer durante a execução do programa.
Exemplo: Divisão por zero ou acesso a índice inexistente.

```java
public class ExemploUnchecked {
    public static void main(String[] args) {
        try {
            int resultado = 10 / 0; // Erro! Divisão por zero
            System.out.println("Resultado: " + resultado);
        } catch (ArithmeticException e) {
            System.out.println("❌ Erro: Não é possível dividir por zero!");
        }
    }
}
```

```
❌ Erro: Não é possível dividir por zero!
```

Tratamento de Exceções - Lidando com Erros no Código! ⚠️🚀

🛠️ Criando Exceções Personalizadas 🎭

📌 Por que criar nossas próprias exceções?

Para tornar o código mais descritivo e fácil de entender.
Quando queremos que um erro tenha um significado especial.
📌 Exemplo: Criando uma exceção para saldo insuficiente no banco.

```java
// Criando uma exceção personalizada
class SaldoInsuficienteException extends Exception {
    public SaldoInsuficienteException(String mensagem) {
        super(mensagem);
    }
}

// Classe Conta Bancária que usa essa exceção
public class ContaBancaria {
    private double saldo = 100.00;

    public void sacar(double valor) throws SaldoInsuficienteException {
        if (valor > saldo) {
            throw new SaldoInsuficienteException("Saldo insuficiente para saque de R$ " + valor);
        }
        saldo -= valor;
        System.out.println("✅ Saque realizado! Novo saldo: R$ " + saldo);
    }

    public static void main(String[] args) {
        ContaBancaria conta = new ContaBancaria();
        try {
            conta.sacar(200.00); // Tentando sacar mais do que o saldo
        } catch (SaldoInsuficienteException e) {
            System.out.println("❌ Erro: " + e.getMessage());
        }
    }
}

```

```
❌ Erro: Saldo insuficiente para saque de R$ 200.0
```

Tratamento de Exceções - Lidando com Erros no Código! ⚠️🚀

📡🔄 Propagação de Exceções (Throws)

📌 O que é?

Permite que uma exceção suba na hierarquia do código.
Usamos **throws** para indicar que um método pode lançar uma exceção.

📌 Isso é útil quando queremos que outro código decida como tratar o erro!

```java
public class ExemploThrows {
    public static void metodoQueLancaExcecao() throws Exception {
        throw new Exception("Erro gerado dentro do método!");
    }

    public static void main(String[] args) {
        try {
            metodoQueLancaExcecao();
        } catch (Exception e) {
            System.out.println("❌ Erro capturado: " + e.getMessage());
        }
    }
}
```

```
❌ Erro capturado: Erro gerado dentro do método!
```

Tratamento de Exceções - Lidando com Erros no Código! ⚠️🚀

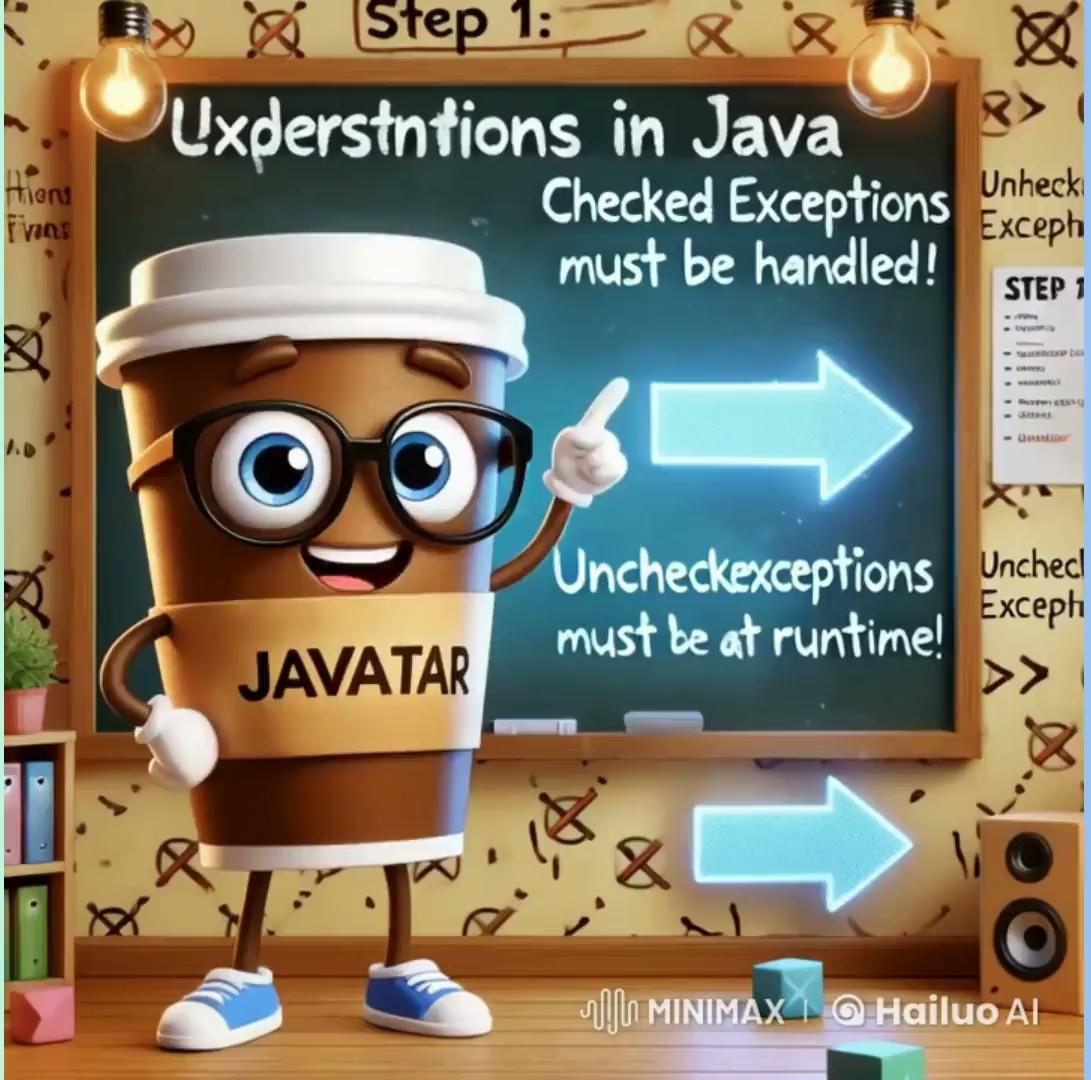
🎯 Conclusão: O que aprendemos?

✅ Checked Exceptions – O compilador exige tratamento!

✅ Unchecked Exceptions – Erros inesperados que podem ser evitados com validações.

✅ Exceções Personalizadas – Criamos para tornar o código mais claro.

✅ Propagação de Exceções (throws) – Permite que um erro suba na hierarquia do código.



PROFKELSON.DEV
BORA CODAR?

Mini Projeto 5 (Com o prof)

Você foi contratado para criar um Sistema de Reservas de Hotel, onde os clientes podem reservar quartos.

O sistema deve:
✅ Criar uma classe Reserva que recebe o nome do cliente e a quantidade de diárias.
✅ Lançar uma exceção personalizada caso o número de diárias seja menor que 1.
✅ Lançar uma exceção padrão caso o nome do cliente esteja vazio.
✅ Propagar a exceção (throws) para que o erro seja tratado na classe principal.

📌 Dica: O funcionamento será parecido com um sistema de reservas real, onde algumas regras precisam ser verificadas antes de confirmar uma reserva!

Mini Projeto 5.1 (Com o prof)

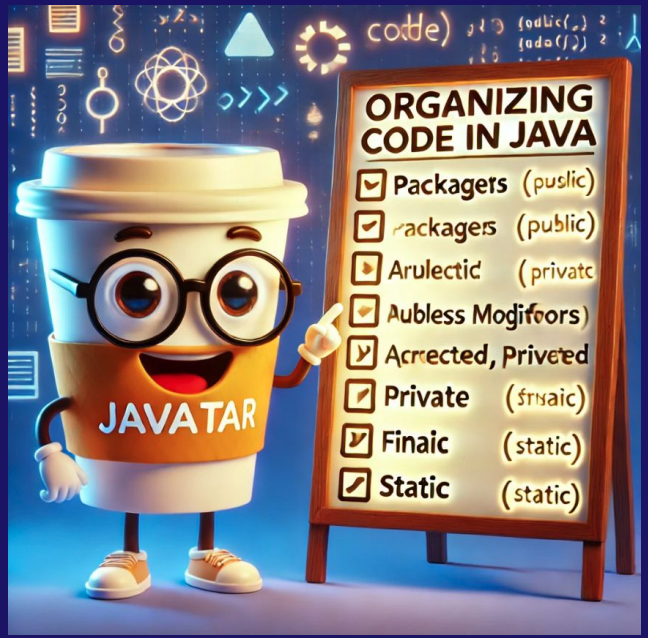
Você foi contratado para desenvolver um Sistema de Saques Bancários. O sistema deve:

✅ Permitir saques de uma conta bancária.
✅ Lançar exceções personalizadas se o usuário tentar sacar mais do que o saldo disponível.
✅ Lidar com exceções comuns, como entrada de valores inválidos.
✅ Propagar exceções quando necessário para métodos externos.

Organização do Código - Pacotes, Modificadores, Final e Static 📦🔧

💡 Por que organizar o código?
💡 Organizar o código é como organizar sua cafeteria favorita! ☕📂
Cada parte tem seu lugar e função, o que deixa tudo mais fácil de encontrar, entender e manter.

📌 Em Java, usamos:
✅ Pacotes para agrupar classes relacionadas.
✅ Modificadores de acesso para controlar a visibilidade do código.
✅ final para evitar mudanças.
✅ static para usar métodos sem precisar criar



Organização do Código - Pacotes, Modificadores, Final e Static 📦🔧

📦1️⃣ Pacotes – Separando o Código por Função

📌 O que são Pacotes?

Pacotes ajudam a organizar o código em diferentes partes, como modelos, serviços e controladores. Cada pacote contém classes relacionadas àquela funcionalidade.

📌 Estrutura de Pacotes Exemplo:

```
/meuapp
  /modelo       -> Contém as classes que representam os dados (Produto, Cliente).
  /servico      -> Contém as classes que executam regras de negócio.
  /controlador  -> Contém as classes que gerenciam as interações do sistema.
```

```java
package modelo;

public class Produto {
    private String nome;
    private double preco;

    public Produto(String nome, double preco) {
        this.nome = nome;
        this.preco = preco;
    }
}
```

Organização do Código - Pacotes, Modificadores, Final e Static 📦🔧

🔐2️⃣ Modificadores de Acesso – Quem Pode Ver o Quê?
📌 O que são?

Modificadores de acesso controlam quem pode acessar uma classe, atributo ou método.
📌 Tipos de Modificadores:
Modificador Visibilidade
public -> Visível para todos. Qualquer classe pode acessar.
protected -> Visível apenas para classes no mesmo pacote e subclasses.
private -> Visível apenas dentro da própria classe. Ninguém mais pode acessar.

📌 Exemplo de Código:

Organização do Código - Pacotes, Modificadores, Final e Static 📦🔧

🔐2️⃣ Modificadores de Acesso — Quem Pode Ver o Quê?

📌 O que são?

```java
public class Produto {
    public String nome;       // Acessível de qualquer lugar
    protected double preco;   // Acessível no mesmo pacote ou subclasses
    private String codigo;    // Acessível apenas dentro da classe Produto
}

```

Organização do Código - Pacotes, Modificadores, Final e Static 📦🔧

🚫3️⃣ final – Evitando Mudanças no Código
📌 O que é final?

final impede que atributos, métodos ou classes sejam modificados.
Uma vez definido, não pode ser alterado.
📌 Como Usar final:
✅ Atributos constantes: Não podem ser modificados.
✅ Métodos final: Não podem ser sobrescritos.
✅ Classes final: Não podem ser herdadas.

📌 Exemplo de Código:

```java
public class Configuracoes {
    public static final String APLICACAO_NOME = "MinhaApp";  // Constante
    public static final int VERSAO = 1;                      // Constante
}

```

📌 Tentativa de alterar APLICACAO_NOME resultará em erro! 🚫

Organização do Código - Pacotes, Modificadores, Final e Static 📦🔧

⚙️4️⃣ static – Acessando Métodos Sem Criar Objetos

📌 O que é static?

static permite acessar métodos e atributos sem precisar criar um objeto da classe.
Pertence à classe, não ao objeto.

📌 Quando usar static:

✅ Métodos utilitários, como cálculos ou formatações.

✅ Atributos que não mudam de uma instância para outra.

📌 Exemplo de Código:

```java
public class Calculadora {
    public static int somar(int a, int b) {
        return a + b;
    }
}

// Usando o método static sem criar um objeto
int resultado = Calculadora.somar(5, 3);
System.out.println("Resultado: " + resultado);
```

```
Resultado: 8
```

Organização do Código - Pacotes, Modificadores, Final e Static 📦🔧

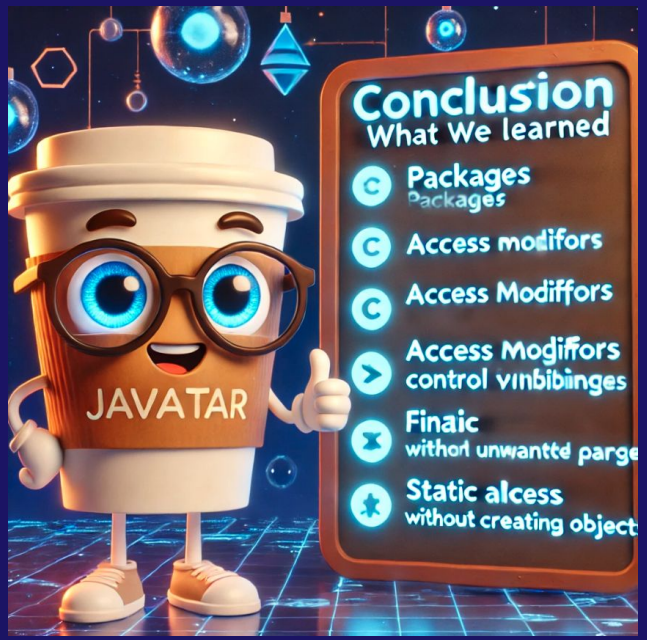
🎯 Conclusão: O que aprendemos?

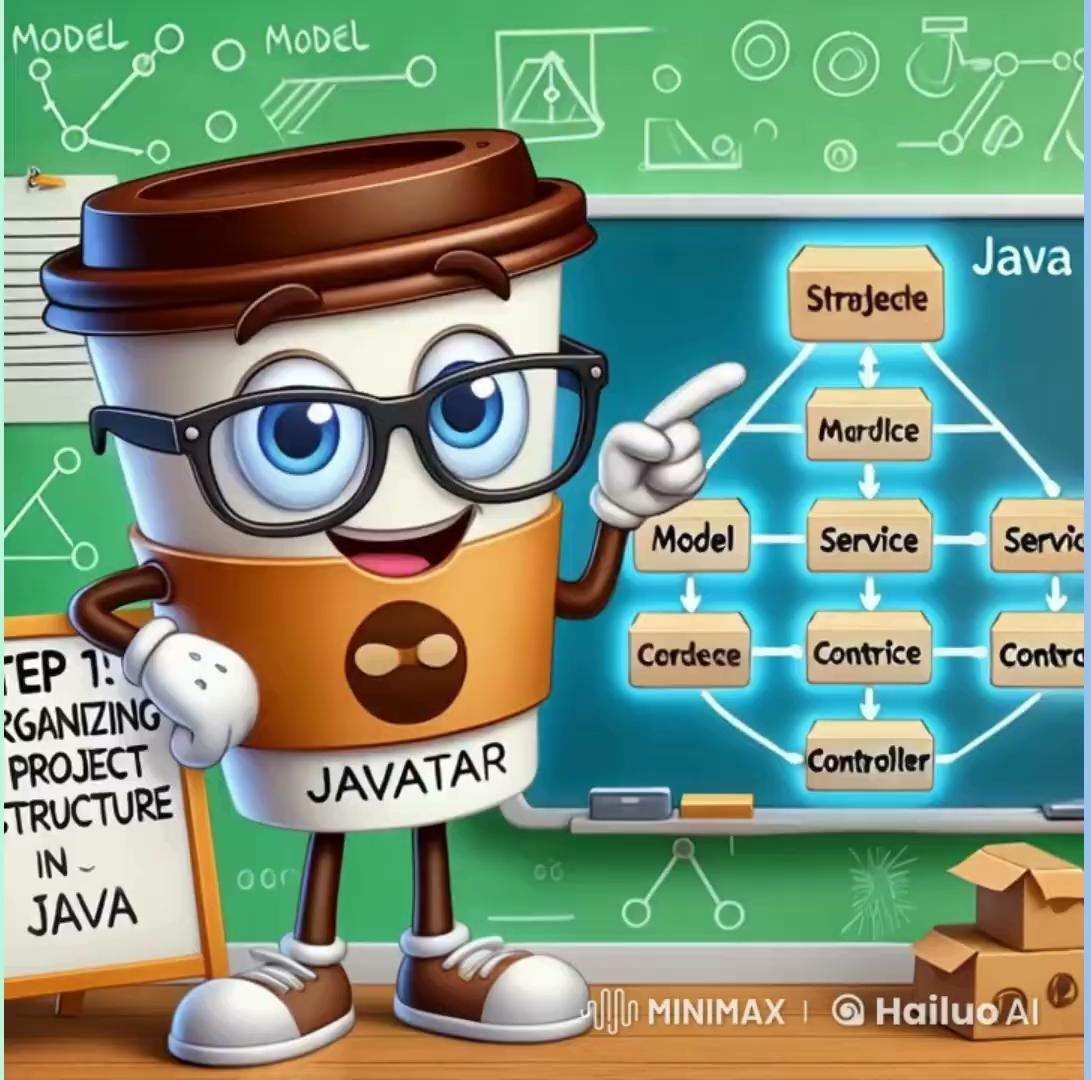
✅ Pacotes organizam o código em partes funcionais.

✅ Modificadores de acesso controlam a visibilidade de atributos e métodos.

✅ final evita mudanças indesejadas no código.

✅ static permite acessar métodos e atributos sem criar objetos.





PROFKELSON.DEV
BORA CODAR?

Mini Projeto 6 (Com o prof)

Você foi contratado para desenvolver um Sistema de Cadastro de Produtos para uma loja. O sistema deve:

✅ Usar pacotes para separar as classes por função (modelo, serviço, utilitário).
✅ Proteger atributos sensíveis usando modificadores de acesso (public, protected, private).
✅ Utilizar o modificador final para definir constantes.
✅ Criar métodos utilitários com static para calcular descontos.

```
/meuapp
  /modelo       -> Contém a classe Produto.java
  /servico      -> Contém a classe ProdutoService.java
  /util         -> Contém a classe CalculadoraDesconto.java
  Main.java     -> Classe principal para rodar o sistema
```