Interface em Java

Qual a diferença entre Interface e Abstração

Conceito	O que é	Palavra-chave	Pode ter código implementado?
Abstração (classe abstrata)	Um modelo base que pode ter comportamento **parcialmente implementado**. Representa uma **ideia incompleta**, que precisa ser **especializada**.	abstract class	✅ Sim
Interface	Um **contrato** que define **o que** uma classe deve fazer, mas **não como**.	interface	❌ Não

Interface vs Classe Abstrata

Aspecto	Interface	Classe Abstrata
Implementação	Não possui estado (até Java 8, só métodos abstratos)	Pode ter estado e construtores
Herança múltipla	✅ Permite várias interfaces	❌ Apenas uma classe abstrata
Uso	Contratos	Base de comportamento comum
Métodos concretos	Desde Java 8 com default	Sempre possível

O que significa quando dizemos que a interface não tem estado?

Uma **interface não pode armazenar valores específicos de instância**, porque **ela não tem objetos concretos**.

Definição:

Interface é um **contrato** que define o **que** uma classe deve fazer, mas **não como**.

```
public interface Animal {
    void emitirSom();
}
```

VER EXEMPLO 1 NA IDE

🔷 Serve para **abstrair comportamentos comuns**.
🔷 Garante **padronização** entre classes diferentes.
🔷 Permite **polimorfismo** e **baixo acoplamento**.

Analogias:

- Interface = “lista de promessas” que uma classe deve cumprir.
- Exemplo do mundo real: uma “tomada elétrica” → define o formato, mas não o que está por trás.

Primeira Implementação

```
public class Cachorro implements Animal {
    public void emitirSom() {
        System.out.println("Au au!");
    }
}
```

VER EXEMPLO 2 NA IDE

Conceitos demonstrados:

- Implementação de interface (implements).
- **Polimorfismo**: Animal pode referenciar qualquer classe que implemente a interface.

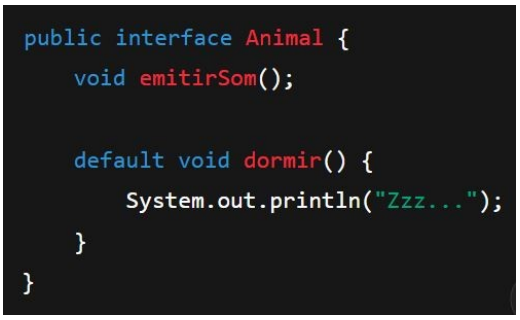
Herança de interface

```
java

interface SerVivo {
    void respirar();
}

interface Animal extends SerVivo {
    void emitirSom();
}
```

VER EXEMPLO 3

Criar interface Servivo e estender pelo Animal

Métodos default



```java
public interface Animal {
    void emitirSom();

    default void dormir() {
        System.out.println("Zzz...");
    }
}
```

Criar o nosso Zoologico usando as interfaces

Ver exemplo 4

Encerramento

- Interfaces garantem **contratos de comportamento**.
- Permitem **polimorfismo** e **baixo acoplamento**.
- São essenciais em **injeção de dependências (Spring)** e **arquiteturas limpas**.

A seguir vai **1 exemplo completo**, com **descrição**, **caso de uso real**, **interfaces**, **herança múltipla de interfaces** e **classe concreta implementando tudo**.

✅ Exemplo: Sistema de Autenticação com Múltiplos Métodos

Imagine que você está criando um sistema onde um usuário pode autenticar de várias maneiras:

- Login e senha
- Autenticação por token
- Autenticação biométrica

Isso é comum em bancos digitais, sistemas corporativos e apps que usam MFA (Multi-Factor Authentication).

Como **Java não possui herança múltipla de classes**, mas **permite herança múltipla de interfaces**, podemos usar interfaces para definir esses comportamentos.

🎯 Objetivo do exemplo

Criar um sistema onde uma classe `UsuarioSistema` é obrigada a implementar:

- `AutenticacaoSenha`
- `AutenticacaoToken`
- `AutenticacaoBiometrica`

Essas interfaces serão herdadas por uma interface maior chamada `AutenticacaoMultiFator`.

Assim teremos **herança múltipla de interfaces** + **uma classe implementando todas** via interface composta.

🧩 Interfaces básicas

```java
public interface AutenticacaoSenha {
    boolean autenticarPorSenha(String usuario, String senha);
}

public interface AutenticacaoToken {
    boolean autenticarPorToken(String token);
}

public interface AutenticacaoBiometrica {
    boolean autenticarPorBiometria(String biometria);
}
```

🧬 Interface com Herança Múltipla

java Copiar código

```java
public interface AutenticacaoMultiFator
        extends AutenticacaoSenha, AutenticacaoToken, AutenticacaoBiometrica {

    void registrarMetodo(String metodo);
}
```

🔷 Aqui temos **herança múltipla de interfaces**.
A interface `AutenticacaoMultiFator` agora exige *todos os métodos* das interfaces herdadas.

👨‍💻 Classe concreta implementando tudo

java Copiar código

```java
public class UsuarioSistema implements AutenticacaoMultiFator {

    @Override
    public boolean autenticarPorSenha(String usuario, String senha) {
        System.out.println("Autenticando por senha...");
        return "admin".equals(usuario) && "123".equals(senha);
    }
```

```java
    @Override
    public boolean autenticarPorToken(String token) {
        System.out.println("Autenticando por token...");
        return token != null && token.startsWith("TK-");
    }

    @Override
    public boolean autenticarPorBiometria(String biometria) {
        System.out.println("Autenticando por biometria...");
        return "BIO-OK".equals(biometria);
    }

    @Override
    public void registrarMetodo(String metodo) {
        System.out.println("Registrando método: " + metodo);
    }
}
```

▶️ Classe Main para testar

java

Copiar código

```java
public class Main {
    public static void main(String[] args) {

        UsuarioSistema usuario = new UsuarioSistema();

        usuario.registrarMetodo("Senha");
        System.out.println("Senha OK? " + usuario.autenticarPorSenha("admin", "

        usuario.registrarMetodo("Token");
        System.out.println("Token OK? " + usuario.autenticarPorToken("TK-999"))

        usuario.registrarMetodo("Biometria");
        System.out.println("Biometria OK? " + usuario.autenticarPorBiometria("B
    }
}
```

📌 Resumo do Caso de Uso

Recurso	Descrição
Contexto real	Autenticação multifatorial em sistemas corporativos
Problema	Permitir vários tipos de autenticação sem herança múltipla de classes
Solução	Usar herança múltipla de interfaces
Benefício	Código limpo, modular, fácil de expandir (ex.: adicionar reconhecimento facial)

✅ Vantagens desse padrão

- Classes ficam **obrigadas** a implementar todos os métodos de segurança.
- Código fica **organizado** e **modular**.
- Novos fatores de autenticação podem ser adicionados facilmente (ex.: SmartCard, OTP via SMS).
- Implementa bem o princípio **Interface Segregation** (SOLID).

TAREFA DE CASA

✅ Tarefa de casa – Versão com INTERFACE

Sistema de Pagamentos Bancários (Interfaces e Polimorfismo)

🎯 Objetivo da atividade

O objetivo desta atividade é aplicar os conceitos de **interfaces, implementação e polimorfismo por contrato** em Java, simulando um sistema real de pagamentos bancários.

Você deverá criar um conjunto de **interfaces** que representem comportamentos comuns de pagamentos, permitindo que o sistema processe qualquer tipo de pagamento sem saber qual é o tipo real do objeto.

🧩 Descrição do problema

Você foi contratado por um banco digital para desenvolver o módulo de pagamentos. Esse banco permite pagamentos de três tipos:

- 💠 **PIX**
- 📄 **Boleto**
- 💳 **Cartão de Crédito**

O sistema deve ser capaz de processar qualquer pagamento de forma genérica usando **interfaces**, garantindo que todas as formas de pagamento sigam um mesmo contrato de operação.

Ou seja, o sistema deve receber um objeto que implemente a interface de pagamento, e o Java deve executar o comportamento adequado de acordo com o tipo fornecido.

🧱 Requisitos técnicos

1️⃣ Crie uma interface Pagamento

A interface deve conter:

- Um método `double getValor();`
  (o valor pode ser armazenado por cada classe concreta)
- Um método `void efetuarPagamento();`
  (cada forma de pagamento implementa à sua maneira)
- Um método padrão `default void confirmarPagamento()`
  que imprime uma mensagem genérica de confirmação.

Exemplo da assinatura:

java Copiar código

```java
public interface Pagamento {
    double getValor();
    void efetuarPagamento();

    default void confirmarPagamento() {
```

```
        System.out.println("Pagamento confirmado com sucesso!");
    }
}
```

---

2 Crie três classes que implementam Pagamento

- **PagamentoPIX**
- **PagamentoBoleto**
- **PagamentoCartao**

Cada classe deve:

- Implementar a interface `Pagamento`
- Ter um atributo `valor`
- Implementar o método `efetuarPagamento()` com uma mensagem personalizada

Exemplo esperado:

java Copiar código

```java
public class PagamentoPIX implements Pagamento {
    private double valor;

    public PagamentoPIX(double valor) {
        this.valor = valor;
    }

    @Override
    public double getValor() {
        return valor;
    }

    @Override
    public void efetuarPagamento() {
        System.out.println("Efetuando pagamento via PIX no valor de R$ " + valo
    }
}
```

As outras duas classes devem seguir o mesmo padrão.

3️⃣ Crie a classe SistemaBancario

Deve conter o método:

java Copiar código

```java
public void processarPagamento(Pagamento pagamento)
```

O corpo deve:

1. Exibir mensagem inicial
2. Chamar `pagamento.efetuarPagamento()`
3. Chamar `pagamento.confirmarPagamento()`

4️⃣ No método main()

Faça:

1. Criar um objeto `SistemaBancario`
2. Criar os três pagamentos (PIX, Boleto e Cartão)
3. Chamar `processarPagamento()` para cada um

✅ Resultado esperado

O aluno deve observar que:

- Interfaces permitem polimorfismo sem herança
- Classes diferentes implementam o mesmo contrato
- O sistema pode processar qualquer pagamento sem saber seu tipo real
  → **Polimorfismo baseado em interface**.

FIM