INTERFACES

As Interfaces em Java são um conceito fundamental da programação orientada a objetos. Elas definem um contrato que as classes devem seguir, especificando um conjunto de métodos que as classes implementadoras devem fornecer. Vou explicar o conceito e discutir quando é interessante aplicá-las:

Conceito de Interface:

1. Uma interface é uma estrutura que contém apenas métodos abstratos e constantes.
2. Ela não pode ser instanciada diretamente.
3. As classes que implementam uma interface devem fornecer implementações para todos os métodos declarados na interface.

Quando aplicar Interfaces:

1. Polimorfismo: Interfaces permitem que objetos de diferentes classes sejam tratados de maneira uniforme, desde que implementem a mesma interface.
2. Abstração: Ajudam a separar a definição do comportamento de sua implementação, permitindo maior flexibilidade no design do sistema.
3. Contratos de API: São úteis para definir contratos claros entre diferentes partes de um sistema ou entre sistemas.
4. Múltipla herança de tipo: Java não suporta múltipla herança de classes, mas uma classe pode implementar várias interfaces.
5. Desacoplamento: Permitem reduzir o acoplamento entre diferentes partes do código, facilitando a manutenção e evolução do sistema.
6. Padrões de projeto: Muitos padrões de projeto em Java fazem uso extensivo de interfaces.
7. Testes unitários: Facilitam a criação de mocks e stubs para testes unitários.
8. Programação orientada a aspectos: Interfaces são frequentemente usadas em conjunto com AOP para definir pontos de corte.

Exemplo prático:

```
// Definição da interface
interface Veiculo {
    void acelerar();
    void frear();
}

// Implementação da interface
public class Carro implements Veiculo {
    @Override
    public void acelerar() {
        System.out.println("Carro acelerando");
    }

    @Override
    public void frear() {
        System.out.println("Carro freando");
    }
}

// Outra implementação da mesma interface
```

```
public class Moto implements Veiculo {
    @Override
    public void acelerar() {
        System.out.println("Moto acelerando");
    }

    @Override
    public void frear() {
        System.out.println("Moto freando");
    }
}
```

Neste exemplo, tanto `Carro` quanto `Moto` implementam a interface `Veiculo`, fornecendo suas próprias implementações para os métodos `acelerar()` e `frear()`.

Declaração de interfaces:

```
public abstract interface Veiculo {}
public interface Veiculo {}
```

Ambas as declarações são equivalentes porque:

- Todas as interfaces são **implicitamente abstratas em Java**.
- A palavra-chave `abstract` para interfaces é redundante e opcional.

O compilador Java automaticamente trata todas as interfaces como abstratas, então você pode incluir ou omitir a palavra-chave `abstract` sem alterar o significado.

Declaração de métodos em interfaces:

```
public abstract int calculoDistanciaVolumeTanque(int tankVolume);
int calculoDistanciaVolumeTanque(int tankVolume);
```

Novamente, ambas as declarações são equivalentes porque:

- Todos os métodos em interfaces **são implicitamente públicos e abstratos**.
- As palavras-chave `public` e `abstract` para métodos de interface são redundantes e opcionais.

O compilador Java automaticamente trata todos os métodos em interfaces como públicos e abstratos, então você pode incluir ou omitir estas palavras-chave.

Declaração de variáveis (constantes) em interfaces:

```
public static final int PESO_MAXIMO = 2000;
int PESO_MAXIMO = 2000;
```

Mais uma vez, ambas as declarações são equivalentes porque:

- Todas as variáveis em interfaces são implicitamente públicas, estáticas e finais.
- As palavras-chave `public`, `static` e `final` para variáveis de interface são redundantes e opcionais.

O compilador Java automaticamente trata todas as variáveis em interfaces como públicas, estáticas e finais (ou seja, constantes), então você pode incluir ou omitir estas palavras-chave.

Regras para Implementação de Interfaces em Java

A. A palavra-chave `public` é obrigatória

Ao implementar métodos de uma interface, a palavra-chave `public` deve ser usada. Isso ocorre porque todos os métodos em uma interface são implicitamente públicos, e a visibilidade do método na classe implementadora não pode ser mais restrita do que a visibilidade na interface

Exemplo:

```
interface Animal {
    void fazerSom();
}

class Cachorro implements Animal {
    // Correto: uso da palavra-chave public
    public void fazerSom() {
        System.out.println("O cachorro late");
    }

    // Incorreto: ausência de public
    // void fazerSom() {
    //     System.out.println("O cachorro late");
    // }
}
```

Explicação:

O método `fazerSom()` na interface `Animal` é implicitamente `public`. Portanto, na classe `Cachorro`, o método `fazerSom()` também deve ser declarado como `public`. Caso contrário, o compilador Java gerará um erro.

B. O tipo de retorno deve ser covariante com a interface

O tipo de retorno do método implementado pode ser o mesmo ou um subtipo (covariante) do tipo de retorno especificado na interface. Isso permite mais flexibilidade ao retornar objetos que sejam mais específicos que o tipo declarado na interface.

Exemplo:

```
interface Animal {
    Animal criarFilhote();
}

class Gato implements Animal {
    // Correto: o tipo de retorno é covariante
    public Gato criarFilhote() {
        return new Gato();
    }

    // Também seria correto
    // public Animal criarFilhote() {
    //     return new Gato();
    // }
}
```

Explicação:

Na interface `Animal`, o método `criarFilhote()` retorna um `Animal`. Na classe `Gato`, é permitido retornar um `Gato`, que é um subtipo de `Animal`. Isso é um exemplo de tipo de retorno covariante.

C. **A assinatura (nome e parâmetros) deve coincidir com a da interface**

A assinatura do método implementado, ou seja, seu nome e parâmetros, deve corresponder exatamente à assinatura definida na interface. Isso garante que o método implementado seja reconhecido como uma implementação válida do método da interface.

Exemplo:

```
interface Animal {
    void mover(int distancia);
}

class Passaro implements Animal {
    // Correto: assinatura do método coincide com a da interface
    public void mover(int distancia) {
        System.out.println("O pássaro voa " + distancia + "
metros.");
    }

    // Incorreto: diferença nos parâmetros
    // public void mover(double distancia) {
    //     System.out.println("O pássaro voa " + distancia + "
metros.");
    // }
}
```

Explicação:
O método `mover()` na interface `Animal` aceita um parâmetro `int`. Na classe `Passaro`, o

método deve ter a mesma assinatura, ou seja, deve também aceitar um `int`. Se você tentar mudar o tipo de parâmetro para `double`, isso resultará em um erro de compilação, pois a assinatura não coincide.

D. **Todos os métodos herdados devem ser implementados**

Todos os métodos definidos na interface ou herdados de interfaces pai devem ser implementados na classe que está implementando a interface. Se uma classe não implementar todos os métodos da interface, ela deve ser declarada como uma classe abstrata. Deve se usar a notação `@Override` ao implementar métodos de uma interface ou ao sobrescrever métodos de uma classe pai.

Exemplo:

```
interface Animal {
    void comer();
    void dormir();
}

class Cavalo implements Animal {
    // Correto: ambos os métodos da interface são implementados

    @Override
    public void comer() {
        System.out.println("O cavalo está comendo.");
    }

    @Override
    public void dormir() {
        System.out.println("O cavalo está dormindo.");
    }

    // Incorreto: método dormir() não implementado
    // public void comer() {
    //     System.out.println("O cavalo está comendo.");
    // }
}
```

Explicação:
A interface `Animal` define dois métodos: `comer()` e `dormir()`. A classe `Cavalo` deve implementar ambos. Se um desses métodos não for implementado, a classe `Cavalo` precisa ser marcada como `abstract` ou o código não compilará.

Herança de Interfaces

Quando uma interface estende outra interface, ela herda todos os métodos da interface pai. Isso é semelhante à herança de classes, mas em interfaces.
Uma classe que implementa a interface filha deve fornecer implementações para **todos** os métodos declarados na interface filha e também para todos os métodos herdados das interfaces pais. Se a classe não implementar todos os métodos exigidos, ela deve ser declarada como `abstract`, ou ocorrerá um erro de compilação.

Exemplo:

```
// Interface pai
interface Veiculo {
    void iniciar();
    void parar();
}

// Interface filha que estende Veiculo
interface Carro extends Veiculo {
    void abrirPorta();
}

// Classe que implementa a interface Carro
class Sedan implements Carro {
    @Override
    public void iniciar() {
        System.out.println("O carro está ligado.");
    }

    @Override
    public void parar() {
        System.out.println("O carro está desligado.");
    }

    @Override
    public void abrirPorta() {
        System.out.println("A porta do carro foi aberta.");
    }
}
```

Explicação:

- A interface `Veiculo` define os métodos `iniciar()` e `parar()`.
- A interface `Carro` estende `Veiculo`, herdando os métodos `iniciar()` e `parar()`, e também define um novo método `abrirPorta()`.
- A classe `Sedan` implementa a interface `Carro`, então ela deve implementar todos os três métodos: `iniciar()`, `parar()` (herdados de `Veiculo`) e `abrirPorta()` (definido em `Carro`).

Se a classe `Sedan` não implementar todos esses métodos, o código não compilará a menos que `Sedan` seja declarada como uma classe abstrata.

Interfaces com métodos duplicados e iguais.

Se duas interfaces possuem métodos com a mesma assinatura (mesmo nome e mesmo tipo de parâmetros) e o mesmo tipo de retorno, a classe que implementa essas interfaces precisa implementar o método apenas uma vez. Essa única implementação será válida para ambas as interfaces.

Como os métodos são idênticos em ambas as interfaces (mesma assinatura e tipo de retorno), o compilador não considera isso um conflito. A implementação na classe atende simultaneamente às duas interfaces.

Exemplo:

```
interface InterfaceA {
    void realizarAcao();
}

interface InterfaceB {
    void realizarAcao();
}

class MinhaClasse implements InterfaceA, InterfaceB {
    @Override
    public void realizarAcao() {
        System.out.println("Ação realizada.");
    }
}
```

Explicação:

- A `InterfaceA` e a `InterfaceB` ambas definem um método `realizarAcao()` com a mesma assinatura e tipo de retorno.
- A classe `MinhaClasse` implementa as duas interfaces e, portanto, fornece uma implementação do método `realizarAcao()`.
- Essa única implementação satisfaz os requisitos de ambas as interfaces.

O Que Acontece?

- **Uma Única Implementação:**
  A implementação de `realizarAcao()` em `MinhaClasse` serve tanto para `InterfaceA` quanto para `InterfaceB`.
- **Chamadas ao Método:**
  Se você tiver uma instância de `MinhaClasse` e a referenciar como um objeto de `InterfaceA` ou `InterfaceB`, chamar `realizarAcao()` resultará na execução da mesma implementação.

```
InterfaceA objA = new MinhaClasse();
objA.realizarAcao(); // "Ação realizada."

InterfaceB objB = new MinhaClasse();
objB.realizarAcao(); // "Ação realizada."
```

Entretanto, se as duas interfaces possuem métodos com a mesma assinatura (mesmo nome e mesmos parâmetros) mas **com tipos de retorno diferentes**, a situação causa um conflito. Isso porque o compilador não sabe qual dos métodos implementar, uma vez que eles não podem coexistir na mesma classe devido à ambiguidade.

Exemplo:

```
// Interface Pai
public interface Ave {
    public void voa();
}

// Interface Filha
public interface Peixe {
    public boolean voa();
}

// Classe que tenta implementar ambas as interfaces
public class Pinguim implements Ave, Peixe {
    @Override
    public void voa() {
        System.out.println("O pinguim nao voa.");
    }

    @Override
    public boolean voa() {
        return false;
    }
}
```

Explicacao:

- A interface `Ave` define um método `voa()` que não retorna nada (`void`).
- A interface `Peixe` define um método `voa()` que retorna um valor booleano (`boolean`).
- A classe `Pinguim` tenta implementar ambas as interfaces.

Este código **nao compila** porque há dois métodos `voa()` com a mesma assinatura (mesmo nome e sem parâmetros), mas com tipos de retorno diferentes (`void` e `boolean`). O compilador não permite a implementação de ambos na mesma classe, pois não consegue distinguir entre os métodos apenas pelo tipo de retorno.

Métodos de interface `default`

Métodos default em interfaces do Java foram introduzidos no Java 8 como uma forma de permitir que interfaces tenham métodos com uma implementação padrão. Antes do Java 8, todas as interfaces podiam apenas declarar métodos, sem fornecer implementações. Essa limitação significava que, se uma **interface fosse alterad**a para incluir novos métodos, **todas as classes que implementavam essa interface precisariam ser atualizadas** para implementar os novos métodos.

Os métodos default permitem que as interfaces sejam evoluídas ao longo do tempo **sem quebrar as implementações existentes**. Isso é particularmente útil em APIs públicas, onde adicionar um novo método a uma interface pode potencialmente quebrar milhares de implementações em clientes. Com métodos default, **você pode adicionar novos métodos a uma interface sem obrigar todas as classes que a implementam a fornecer uma implementação**.

As principais características de métodos *default* (essa palavra chave só pode ser usada em interfaces, se usar em uma classe vai dar erro de compilação) são:

1. **Múltiplas interfaces com métodos de mesmo nome**: Antes dos métodos default, se uma classe implementasse duas interfaces com métodos de mesmo nome, haveria um conflito que o programador teria que resolver explicitamente. Com métodos default, a classe pode herdar implementações de múltiplas interfaces, mesmo se os nomes dos métodos forem iguais.

2. **Múltipla herança de comportamento:** Java não permite herança múltipla de classes, mas com métodos default, as interfaces podem fornecer implementações concretas. Isso permite que uma classe herde comportamentos de várias interfaces, simulando um tipo limitado de herança múltipla.

3. **Sem herança de estado:** É importante notar que, embora permita herdar comportamentos, **as interfaces ainda não podem conter estado** (variáveis de instância). A herança múltipla é **limitada apenas a comportamentos**.

4. **Flexibilidade no design:** Métodos default permitem adicionar novos métodos a interfaces existentes sem quebrar as implementações existentes. Isso oferece mais flexibilidade na evolução de APIs.

5. **Composição modular:** Os desenvolvedores podem criar interfaces menores e mais específicas com implementações default, e então compor essas interfaces para criar comportamentos mais complexos nas classes que as implementam.

Regras para Usar Métodos Default

1. A palavra-chave *default* com um método **só pode ser usada** na interface
2. Deve ter um **corpo** (implementação padrão);
3. É implicitamente **público;**
4. Não pode ser abstrato, final ou estático;
5. **Pode ou não ser sobrescrito** por uma classe que implementa a interface;
6. Se a classe herdar dois ou mais métodos default com a mesma assinatura de método, **então ela deve sobrescrever o método.**

Explicando melhor o item 4:

Métodos finais são projetados para serem imutáveis em classes filhas. Isso garante segurança, permite otimizações do compilador e indica que o comportamento é essencial para a classe. → métodos defaults permitem *overriding*, métodos do tipo `final` não, logo um método com as *keywords* `final default` é contrasenso.

Já os métodos **estáticos pertencem à classe**, não a instâncias específicas. Eles **são vinculados em tempo de compilação** e não participam do polimorfismo, tornando a sobrescrita sem sentido. → métodos defaults permitem *overriding*, métodos estáticos não, logo um método com as *keywords* `static default` é contrasenso.

Agora, **pense nos métodos default**. Tentar marcá-los como abstratos seria um contrassenso total. **Métodos abstratos são apenas declarações sem corpo**, enquanto **métodos default precisam ter uma implementação concreta**. É como tentar miscar água e óleo - simplesmente não funciona!

Explicando melhor o item 6:

Quando uma classe implementa múltiplas interfaces, e duas ou mais dessas interfaces contêm métodos default com a mesma assinatura (mesmo nome, mesmos parâmetros e tipo de retorno), ocorre um conflito. Java não pode decidir automaticamente qual implementação usar. Para resolver esse conflito, a classe que implementa essas interfaces é obrigada a sobrescrever o método em questão.

Exemplo:

```java
interface Cumprimento {
    default void saudar() {
        System.out.println("Olá!");
    }
}

interface Despedida {
    default void saudar() {
        System.out.println("Tchau!");
    }
}

class Conversa implements Cumprimento, Despedida {
    // Esta classe DEVE sobrescrever o método saudar()
    @Override
    public void saudar() {
        // Podemos escolher uma implementação, combinar ambas, ou criar uma nova
        Cumprimento.super.saudar();
        System.out.println("Como vai?");
    }
}

public class Exemplo {
    public static void main(String[] args) {
        Conversa conversa = new Conversa();
        conversa.saudar();
    }
}

```

Neste exemplo:

1. Temos duas interfaces, `Cumprimento` e `Despedida`, ambas com um método default `saudar()`.
2. A classe `Conversa` implementa ambas as interfaces.
3. Como há um conflito entre os métodos default `saudar()`, a classe `Conversa` é obrigada a sobrescrever este método.
4. Na sobrescrita, podemos escolher usar a implementação de uma das interfaces (usando `Cumprimento.super.saudar()` ou `Despedida.super.saudar()`), combinar ambas, ou fornecer uma implementação completamente nova.
   Neste caso, escolhemos usar a implementação de `Cumprimento` e adicionar uma nova linha.

Quando executado, este código imprimirá:

```
Olá!
Como vai?
```

Métodos de interface `static`

Métodos estáticos em interfaces **não podem ser sobrescritos** porque eles são **associados diretamente à interface** em que foram definidos, e não a qualquer classe que implemente a interface. Alguns detalhes importantes:

Métodos estáticos pertencem à interface: Em Java, um método estático é associado à interface ou classe onde foi declarado. Isso significa que você sempre invoca o método estático através do nome da interface (ou classe) e não de uma instância ou subclasse. Como métodos estáticos não são herdados por classes que implementam a interface, eles não podem ser sobrescritos. Apenas métodos de instância (não estáticos) são herdados e, portanto, sujeitos a sobrescrita em subclasses.

Corpo obrigatório: Sim, métodos estáticos em interfaces devem ter um corpo. Ao contrário dos métodos abstratos, que não possuem corpo e são implementados pelas classes que implementam a interface, um método estático deve ser completamente definido na interface.

Design intencional: A decisão de não permitir a sobrescrita de métodos estáticos é um design intencional da linguagem Java, para garantir que a lógica estática em uma interface seja consistente e não possa ser alterada por classes individuais. Isso ajuda a manter uma separação clara entre métodos que são específicos de uma classe/interface e aqueles que são associados a instâncias.

Exemplo de código:

```java
public interface Car {
    // Método estático com corpo
    static int getMaxSpeed() {
        return 100;
    }
}

public class Ford implements Car {
    // Método não estático com a mesma assinatura, mas não estático
    public int getMaxSpeed() {
        return 120;
    }
}

public class Main {
    public static void main(String[] args) {
        // Chamada ao método estático da interface
        System.out.println(Car.getMaxSpeed());  // Saída: 100

        // Chamada ao método não estático da instância Ford
        Ford ford = new Ford();
        System.out.println(ford.getMaxSpeed());  // Saída: 120
    }
}
```

`myFord.getMaxSpeed()`: Chama o método não estático da classe `Ford`, que retorna `120`.
`Car.getMaxSpeed()`: Chama o método estático da interface `Car`, que retorna `100`.

Métodos de interface `private`

Desde o Java 9, é possível declarar métodos private em interfaces. Esses métodos são usados para organizar o código dentro da interface, permitindo que métodos default e static reutilizem a lógica comum sem expor essa lógica para as classes que implementam a interface. Devem obrigatoriamente ter um corpo uma vez que não podem ser sobreescritos.

Quando usar métodos private em interfaces:

1. **Reutilização de código**: Se você tiver um código comum que é utilizado por vários métodos default ou static na interface, **você pode encapsular essa lógica em um método private**.
2. **Organização**: Isso ajuda a manter o código mais limpo e modular, evitando duplicação de código e mantendo a implementação detalhada oculta.

Regras para métodos private em interfaces:

1. **Visibilidade**: Métodos private em interfaces só podem ser acessados por outros métodos da mesma interface (default, static ou private).

2. **Não podem ser sobrescritos**: Como são private, esses métodos não são herdados pelas classes que implementam a interface e, portanto, **não podem ser sobrescritos**.

Exemplo de código:

```
1
2  public interface Car {
3      // Método default que pode ser chamado pelas classes que implementam a interface
4      default double calculateFuelEfficiency(int speed, int rpm) {
5          // Usando o método private para calcular a eficiência
6          return efficiencyAlgorithm(speed, rpm);
7      }
8
9      // Método private para encapsular a lógica de cálculo da eficiência
10     private double efficiencyAlgorithm(int speed, int rpm) {
11         // Lógica fictícia para calcular a eficiência do combustível
12         double efficiency = (double) speed / rpm;
13         return efficiency * 100;
14     }
15 }
16
17 public class Ford implements Car {
18     // A classe Ford pode usar o método default, MAS NÃO PODE USAR O MÉTODO PRIVATE
19 }
20
21 public class Main {
22     public static void main(String[] args) {
23         Ford myFord = new Ford();
24
25         // Chamando o método default da interface Car
26         double efficiency = myFord.calculateFuelEfficiency(80, 3000);
27
28         System.out.println("Fuel Efficiency: " + efficiency);  // Exemplo de saída: Fuel Efficiency: 2.6666666666666665
29     }
30 }
31
```

Explicação:

Método default (`calculateFuelEfficiency`) na interface `Car`:

- Esse método é acessível a todas as classes que implementam a interface `Car`.
- Ele usa o método private (`efficiencyAlgorithm`) para realizar o cálculo.

Método private (efficiencyAlgorithm) na interface `Car`:

- Esse método encapsula a lógica de cálculo da eficiência do combustível.
- Ele não pode ser acessado diretamente pelas classes que implementam a interface `Car`.

IMPORTANTE!!

Métodos privados podem **ser estáticos ou não estáticos**, e o uso de cada um deles tem **suas próprias regras e aplicações**.

Métodos `private static`

Esses **métodos podem ser chamados por qualquer método da interface**, **seja ele estático ou não estático**. Isso significa que, independentemente de você estar dentro de um método default, static, ou private, você pode chamar um método private static.

Métodos `private` não estáticos

Métodos **private não estáticos** pertencem à instância da interface, ou seja, eles **só podem ser chamados por métodos não estáticos** (default ou outros métodos private não estáticos) dentro da mesma interface.

Exemplo de código:

```java
public interface Vehicle {

    // Método default que usa um método private não estático
    default void start() {
        System.out.println("Starting vehicle...");
        logAction("Vehicle started");
        System.out.println("Max speed: " + calculateMaxSpeed(2.5));
    }

    // Método default que usa um método private não estático
    default void stop() {
        System.out.println("Stopping vehicle...");
        logAction("Vehicle stopped");
    }

    // Método private não estático para registrar uma ação
    private void logAction(String action) {
        System.out.println("Log: " + action);
    }

    // Método static que usa um método private static
    static double calculateMaxSpeed(double engineFactor) {
        return getBaseSpeed() * engineFactor;
    }

    // Método private static para calcular a velocidade base
    private static double getBaseSpeed() {
        return 100.0; // Velocidade base fictícia
    }
}

public class Car implements Vehicle {
    // A classe Car não precisa implementar nada adicional,
    // pois os métodos default da interface Vehicle já fornecem o comportamento.
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();

        // Chamando métodos default que utilizam métodos private
        myCar.start();  // Saída: Starting vehicle... Log: Vehicle started
                        //         Max speed: 250.0
        myCar.stop();   // Saída: Stopping vehicle... Log: Vehicle stopped

        // Chamando método static diretamente na interface
        double maxSpeed = Vehicle.calculateMaxSpeed(3.0);
        System.out.println("Calculated max speed: " + maxSpeed);  // Saída: Calculated max speed: 300.0
    }
}
```

Explicação:

Métodos default (`start` e `stop`):

- Esses métodos podem ser chamados por qualquer instância de uma classe que implemente a interface `Vehicle`.
- Eles utilizam o método private não estático `logAction` para registrar uma ação.

Método private não estático (`logAction`):

- Este método é usado para encapsular a lógica de log dentro da interface.
- Ele só pode ser chamado por outros métodos não estáticos (default ou private) dentro da mesma interface.

Método static (`calculateMaxSpeed`):

- Este método pode ser chamado diretamente na interface (`Vehicle.calculateMaxSpeed(3.0)`).
- Ele utiliza o método private static `getBaseSpeed` para obter a velocidade base e, em seguida, calcular a velocidade máxima.

Método private static (`getBaseSpeed`):

- Encapsula a lógica de calcular a velocidade base.
- Pode ser chamado por qualquer método `static` ou `default` dentro da interface.

Se você tentar usar um método private não estático em um método estático o compilador vai responder com uma mensagem de erro

“Cannot make a static reference to the non-static method *** from the type ***” [Eclipse IDE]