

# Interfaces

---

## Orientação a Objetos - DCC025

Prof. Edmar Welington Oliveira  
oliveira.edmar@ufjf.edu.br

Universidade Federal de Juiz de Fora - UFJF  
Departamento de Ciência da Computação - DCC

# Interface

## ■ Definição

- Uma interface descreve um conjunto de métodos que podem ser chamados em um objeto, seja para instruir o mesmo a realizar alguma tarefa ou retornar algumas informações, por exemplo.
- Quando dizemos que uma classe implementa uma interface, obrigatoriamente essa classe tem que implementar todos os métodos declarados na interface.
- Neste sentido, uma interface pode ser vista como sendo um contrato estabelecido entre a classe que implementa a interface (provedora do serviço) e a classe que vai utilizar os métodos definidos na interface (cliente do serviço).

# Interface - Declaração

## ■ Declaração de Interfaces

- Inicia-se com a palavra-chave "interface" e contém somente constantes e métodos do tipo "abstract". Diferentemente das classes, TODOS os membros de interface devem ser **public** e as interfaces não podem especificar nenhum detalhe de implementação como declarações de métodos concretos
- TODOS os métodos declarados em uma interface são implicitamente métodos **public abstract** e todos os campos são **public, static** e **final**.

# Interface - Declaração

- Boa prática de programação

- O estilo adequado para declarar os métodos de uma interface (em Java) é não usar as palavras-chave **public** e **abstract**, porque elas são redundantes nas declarações de método de interface.

- OBS

- Interfaces não contém **atributos**, apenas assinaturas de métodos
- Contudo, **constantes** podem ser declaradas na interface

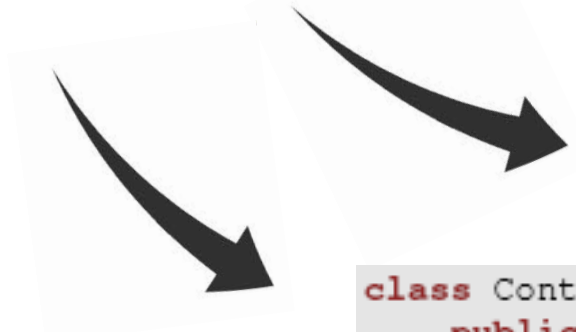
```
1 public interface IUsaCombustivel {  
2  
3     public int valor;  
4  
5 }  
6
```

✖ The blank final field valor may not have been initialized

```
1 public interface IUsaCombustivel {  
2  
3     int valor = 0;  
4  
5 }
```

# Interfaces

```
interface Conta {  
    void deposita(double valor);  
    void saca(double valor);  
}
```




```
class ContaPoupanca implements Conta {  
    public void deposita(double valor) {  
        // implementacao  
    }  
    public void saca(double valor) {  
        // implementacao  
    }  
}
```

```
class ContaCorrente implements Conta {  
    public void deposita(double valor) {  
        // implementacao  
    }  
    public void saca(double valor) {  
        // implementacao  
    }  
}
```

# Interfaces

```
interface Conta {  
    void deposita(double valor);  
    void saca(double valor);  
}
```

Todos os métodos da uma interface são "Abstract" e "Public"



```
// Esta classe não compila porque ela não implementou o método saca()  
class ContaCorrente implements Conta {  
    public void deposita(double valor) {  
        // implementacao  
    }  
}
```

As classes concretas que implementam a interface são obrigados a implementar todos os métodos declarados na interface.

Erro de compilação caso isso não seja feito

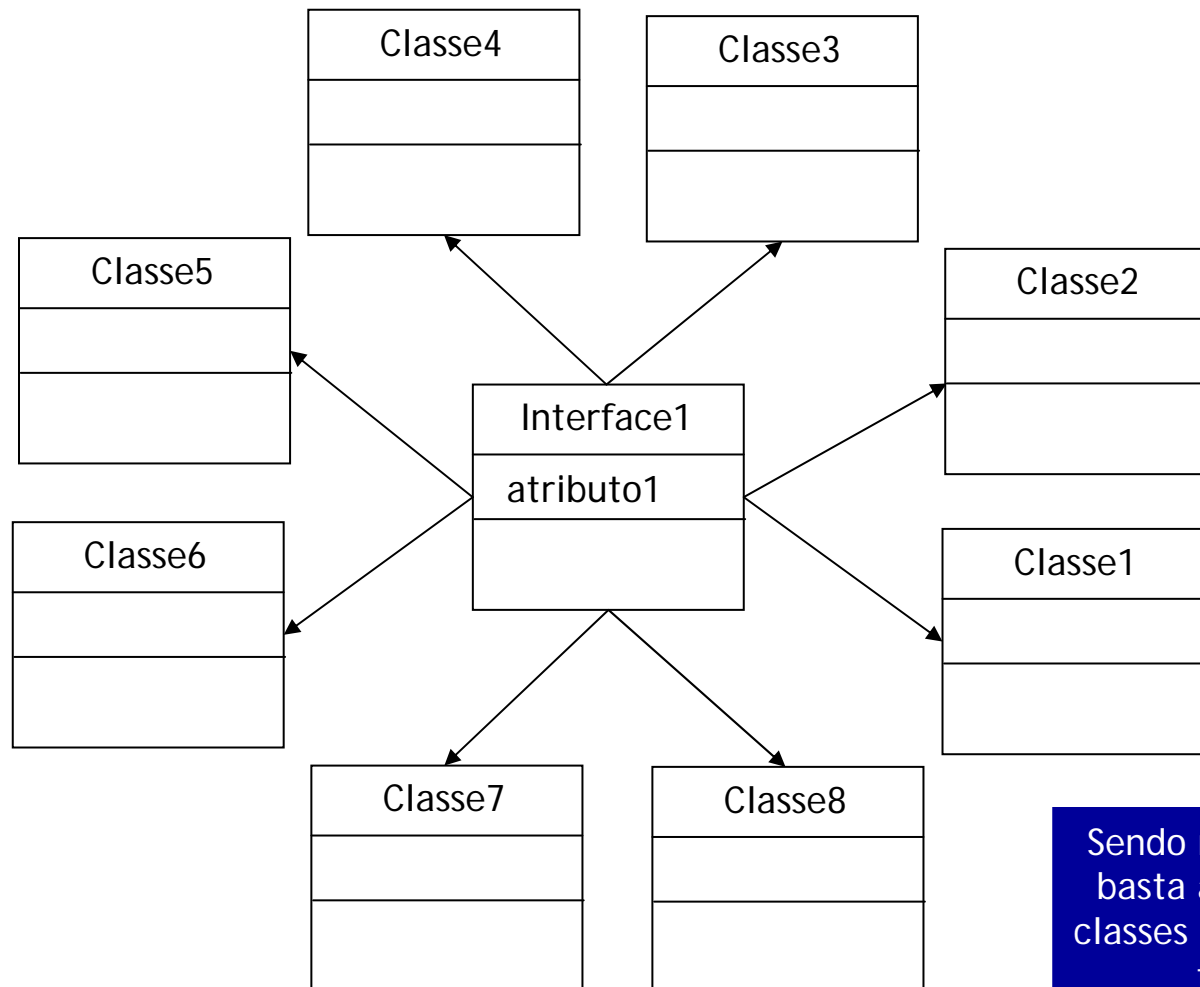
# Interfaces

```
interface Coins {  
    int  
        PENNY = 1,  
        NICKEL = 5,  
        DIME = 10,  
        QUARTER = 25,  
        DOLAR = 100;  
}  
  
class SodaMachine implements Coins {  
    int price = 3*QUARTER;  
    // ...  
}
```

A interface Coins define um conjunto de constantes. As classes que implementam tal interface irão, obrigatoriamente, possuir as mesmas constantes.

Neste caso, não é preciso declarar novamente tais constantes na classe SodaMachine

# Interfaces



Sendo necessário alterar o atributo1, basta alterar na interface - todas as classes que implementam tal interface farão uso do novo valor



# Diferença Classes Abstratas e Interfaces

## ■ Algumas Diferenças

- A interface obrigatoriamente não tem um “corpo” associado. Para que uma classe seja abstrata basta que ela seja assim declarada, mas a classe pode incluir atributos de objetos e definição de métodos, públicos ou não. Na interface, apenas métodos públicos podem ser declarados — mas não definidos. Da mesma forma, não é possível definir atributos — apenas constantes públicas.
- Enquanto uma classe abstrata é “estendida” (palavra chave extends) por classes derivadas, uma interface Java é “implementada” (palavra chave implements) por outras classes.

# Diferença Classes Abstratas e Interfaces

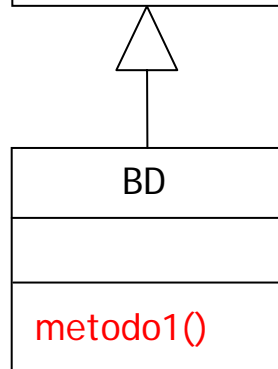
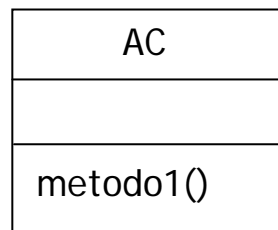
## ■ Diferença

- A diferença essencial entre classes abstratas e interfaces é que **uma classe herdeira somente pode herdar de uma única classe** (abstrata ou não), enquanto que qualquer classe pode implementar várias interfaces simultaneamente.
- Interfaces são um mecanismo simplificado de implementação de herança múltipla em Java, permitindo que mais de uma interface determine os métodos que as que uma classe herdeira deve implementar.
  - OBS: lembre-se que Java não suporta herança múltipla. Logo, uma classe herdeira (uma subclasse) só pode herdar de uma única superclasse.
- Interface é dita ser uma classe abstrata “pura”
- Toda interface deve ser vista como uma classe abstrata (o contrário é falso)

# Quando Usar Interfaces

- Quando projetar com interfaces

- Quando se deseja apoiar polimorfismo sem se comprometer com uma particular hierarquia de classes. Se uma superclasse AC é usada sem uma interface, qualquer nova solução polimórfica precisa ser uma subclasse de AC (se for, será cópia de código)



Implementação de herança

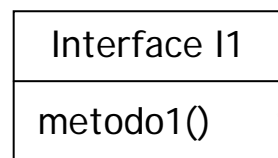


Implementação de outra classe com o mesmo método - cópia de código (erro)

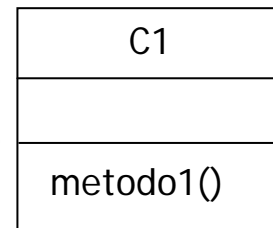
# Quando Usar Interfaces

## ■ Regra Geral

- Se existe uma hierarquia de classes com uma **superclasse abstrata C1**, considere fazer uma interface I1 que corresponde à assinatura do método público de C1, e depois declare C1 para implementar a interface I1.
- Ponto de flexibilização para casos futuros
  - Outras classes podem implementar a interface I1
  - Se necessário, classes podem herdar de C1



C1 implementa I1



C1 "implementa" o metodo1() e toda subclasse de C1 deve implementar o método1()

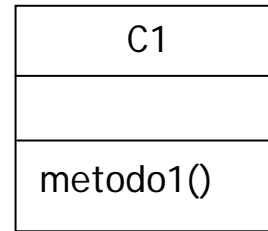
└─ Classe Abstrata

# Quando Usar Interfaces

Interface I1



Posso ter uma classe que não nada a ver com C1 (em termos de herança), mas que precisa implementar os mesmos métodos



Posso ter uma classe que precise herdar de C1

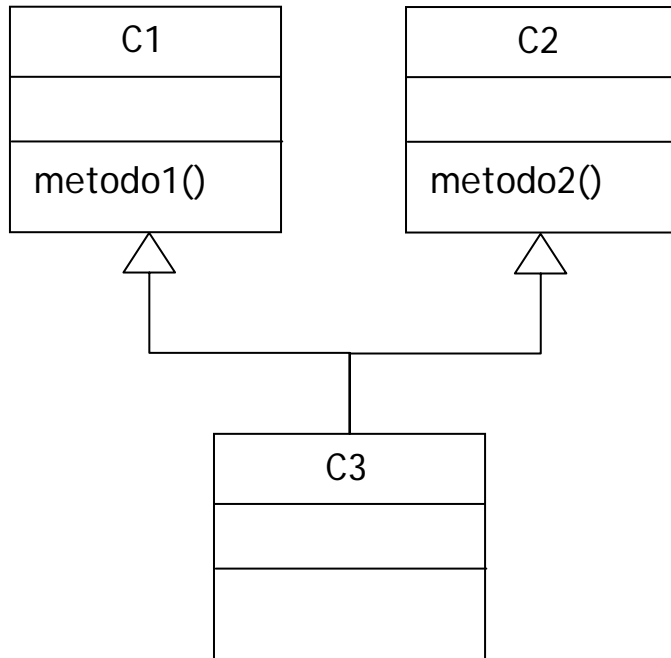
# Interfaces - Vantagens

- Algumas vantagens

- Padronização de assinatura de métodos
- Garantir que certas classes implementem certos métodos
- Possibilidade de implementar diversas interfaces

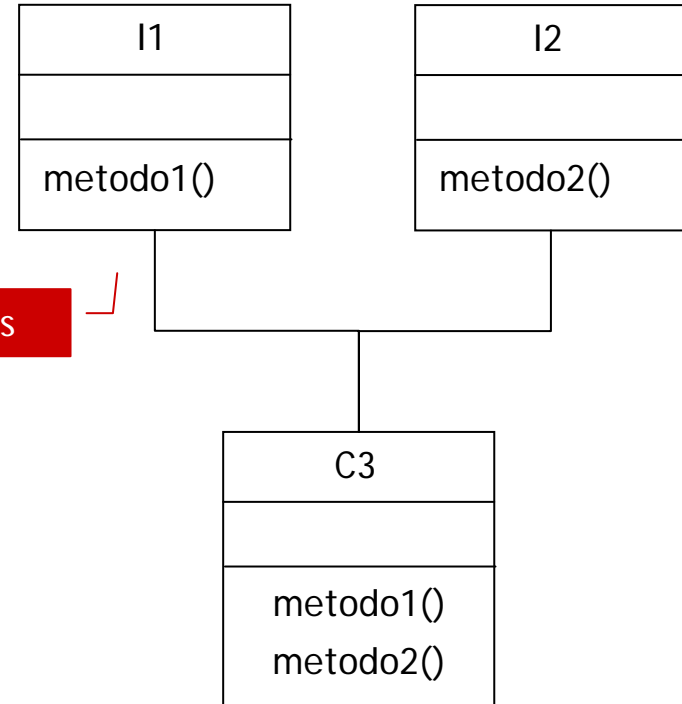
- OBS: Embora OO apresente o conceito de herança múltipla (possibilidade de uma classe herdar de duas ou mais classes), Java não oferece suporte ao mesmo. Contudo, uma forma de fazer uma classe “herdar” coisas de mais de uma classe é através do mecanismo de interfaces.

# Interfaces



Herança múltipla: a classe C3 herda das classes C1 e C2.

Métodos abstratos



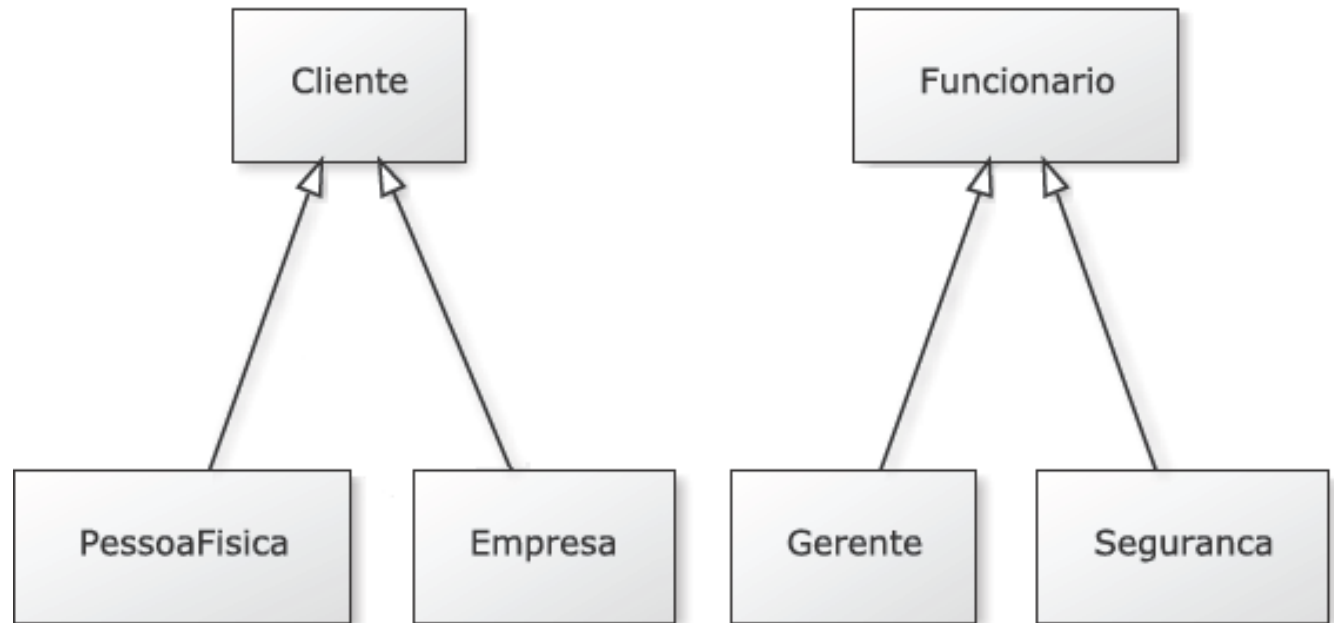
Por ser uma relação de interface, obrigatoriamente, C3 deve implementar os métodos 1 e 2

# Interfaces e Herança

- Tema de discussão
  - Uso de interfaces ou uso de herança
  - A grosso modo, priorizar a utilização de interfaces permite que alterações pontuais em determinados trechos do código fonte sejam feitas mais facilmente pois diminui as ocorrências de efeitos colaterais indesejados no resto da aplicação.
  - Por outro lado, priorizar a utilização de herança pode diminuir a quantidade de código escrito no início do desenvolvimento de um projeto.



# Interfaces e Herança



Vamos supor que gerentes e empresas podem acessar o sistema de um banco através de um usuário e de uma senha.

Proposta inicial: implementar um único método para fazer a autenticação (validação o usuário de senha)

# Interfaces e Herança

```
class AutenticadorDeUsuario {  
    public boolean autentica(??? u) {  
        // implementação  
    }  
}
```

```
class AutenticadorDeUsuario {  
    public boolean autentica(Empresa u)  
}
```

```
class AutenticadorDeUsuario {  
    public boolean autentica(Gerente u)  
}
```

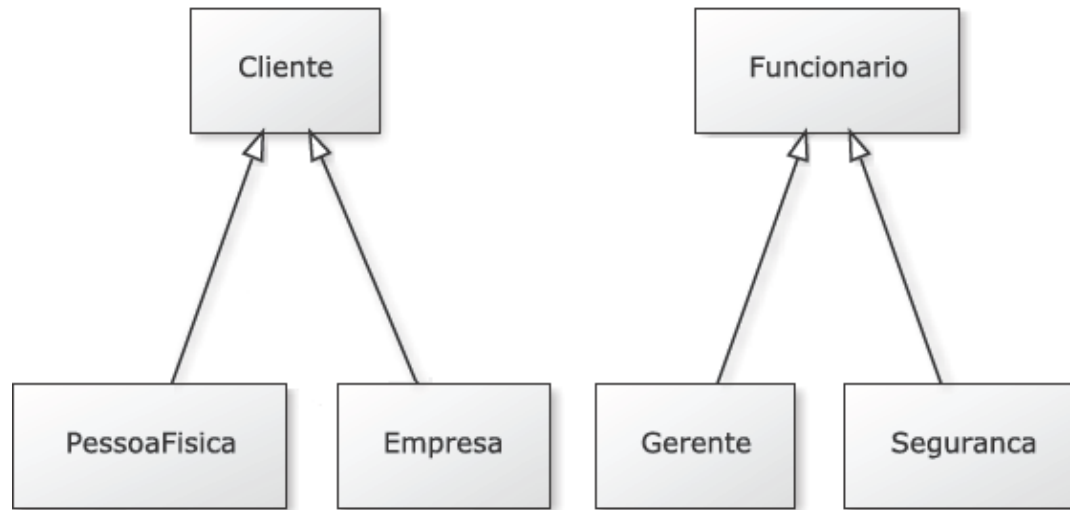
Pergunta: qual seria o tipo de parâmetro de método?

Um objeto do tipo Empresa?  
Um objeto do tipo Gerente?

OBS: Não há como implementar polimorfismo utilizando um método geral.

Quais as consequências disso?

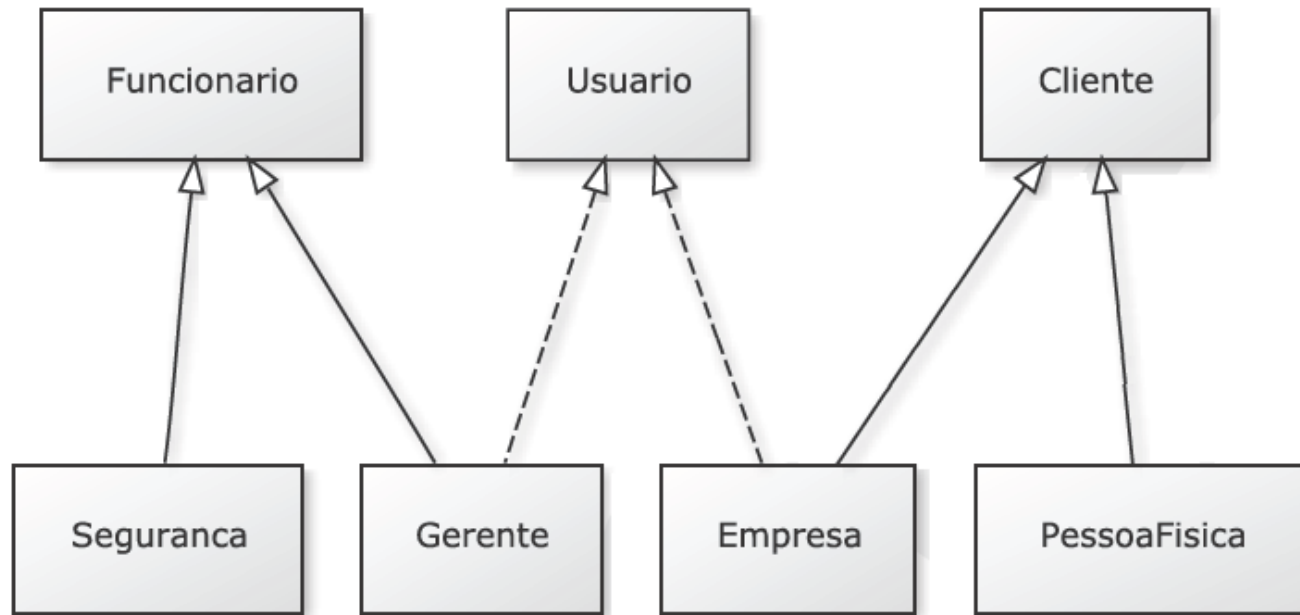
# Interfaces e Herança



OBS: Não há polimorfismo entre objetos da classe Empresa e objetos da classe Gerente

Para obter polimorfismo entre os objetos dessas duas classes somente com herança, deveríamos colocá-las na mesma árvore de herança. Mas, isso não faz sentido pois uma empresa não é um funcionário e o gerente não é cliente.

# Interfaces e Herança

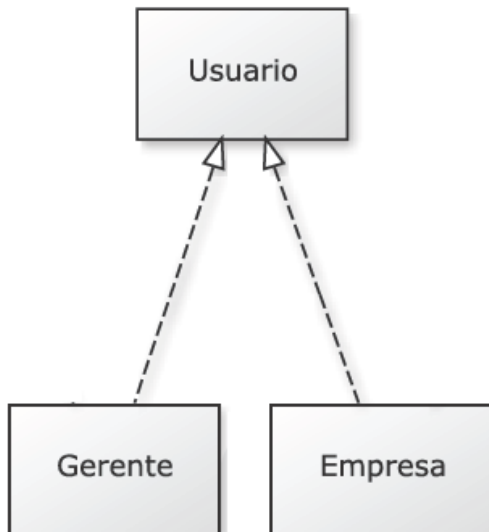


A solução é usar interfaces para obter polimorfismo entre objetos da classe Gerente e da classe Empresa

# Interfaces e Herança

```
class AutenticadorDeUsuario {  
    public boolean autentica(Usuario u) {  
        // implementação  
    }  
}
```

consequimos definir o que o método AUTENTICA() deve receber um parâmetro para trabalhar tanto com gerentes quanto com empresas. Ele deve receber um parâmetro do tipo USUARIO.



Pergunta: a Interface Usuário poderia ser uma classe?

# Interfaces na Prática

```
2 public class SodaMachine implements Coins{  
3  
4 }
```

```
2 public interface Coins {  
3  
4 int  
5 PENNY = 1,  
6 NICKEL = 5,  
7 DIME = 10,  
8 QUARTER = 25,  
9 DOLAR = 100;  
10 }
```

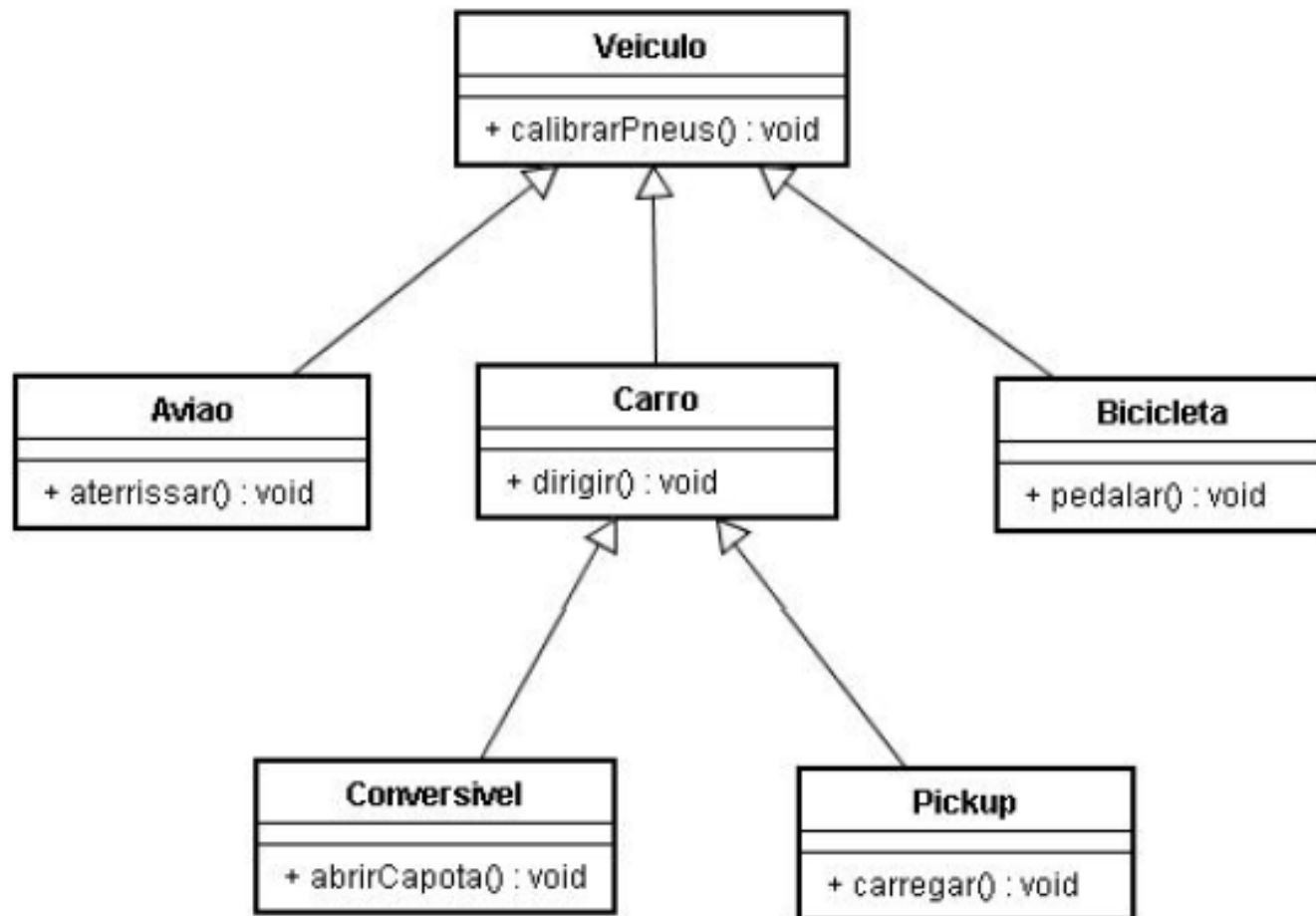
```
1 public class Teste {  
2  
3 public static void main(String args[]){  
4  
5     Teste teste = new Teste();  
6     SodaMachine sm = new SodaMachine();  
7     teste.autentica(sm);  
8 }  
9  
10 public void autentica(Coins c){  
1     System.out.println(c.DOLAR);  
2 }  
3 }
```

Problems Javadoc Declaration Console

<terminated> Teste [Java Application] C:\Program Files\Java\jdk  
100



# Aplicação de Interfaces

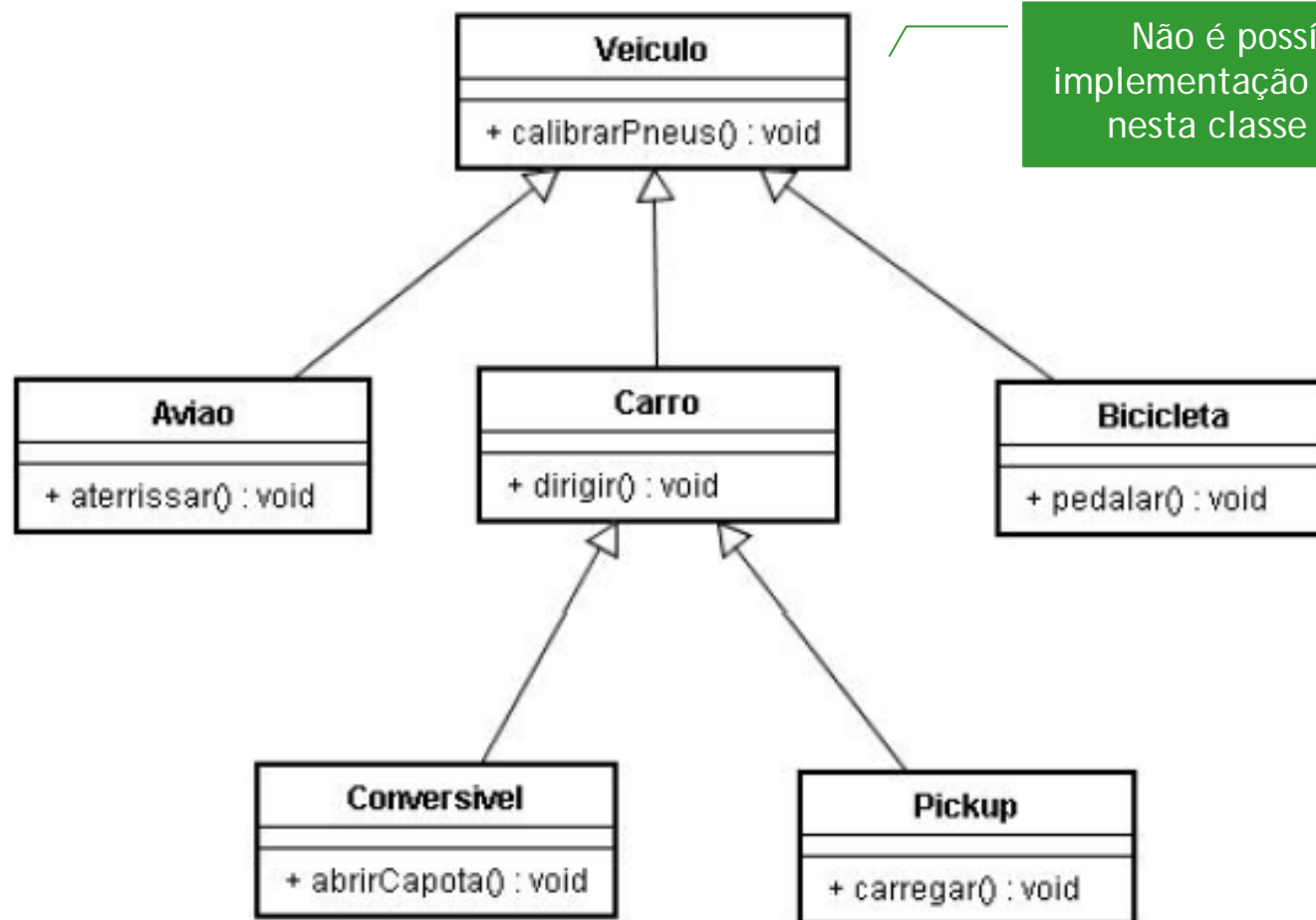


# Aplicação de Interfaces

- Considerações (diagrama anterior)
  - Todos os meios de transporte listados na hierarquia possuem pneus. Por isso, o método `calibrarPneus()` foi declarado no topo da hierarquia, sendo, dessa forma, herdado pelas demais subclasses;
  - Apenas os aviões aterrissam. Logo, o método `aterrissar()` foi declarado na subclasse `Aviao`.



# Aplicação de Interfaces

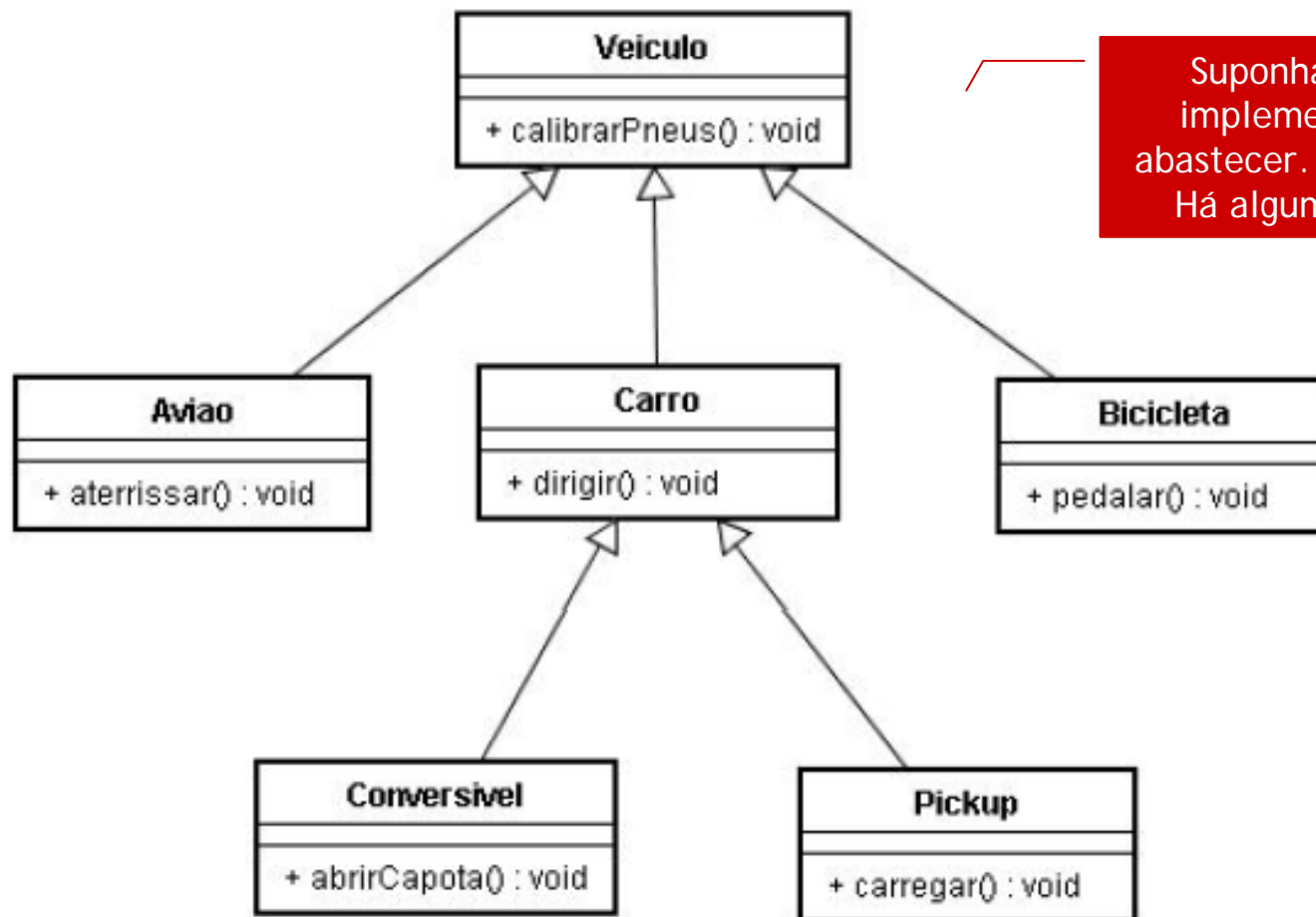


Não é possível definir uma implementação para abastecimento nesta classe Veículo. Porque?

# Aplicação de Interfaces

- Mais considerações
  - Quase todos os veículos da hierarquia anterior podem ser abastecidos com combustível. Entretanto, se fosse definida uma implementação para tal na classe **Veiculo**, ela seria herdada por **Bicicleta**, que não pode ser abastecida;

# Aplicação de Interfaces



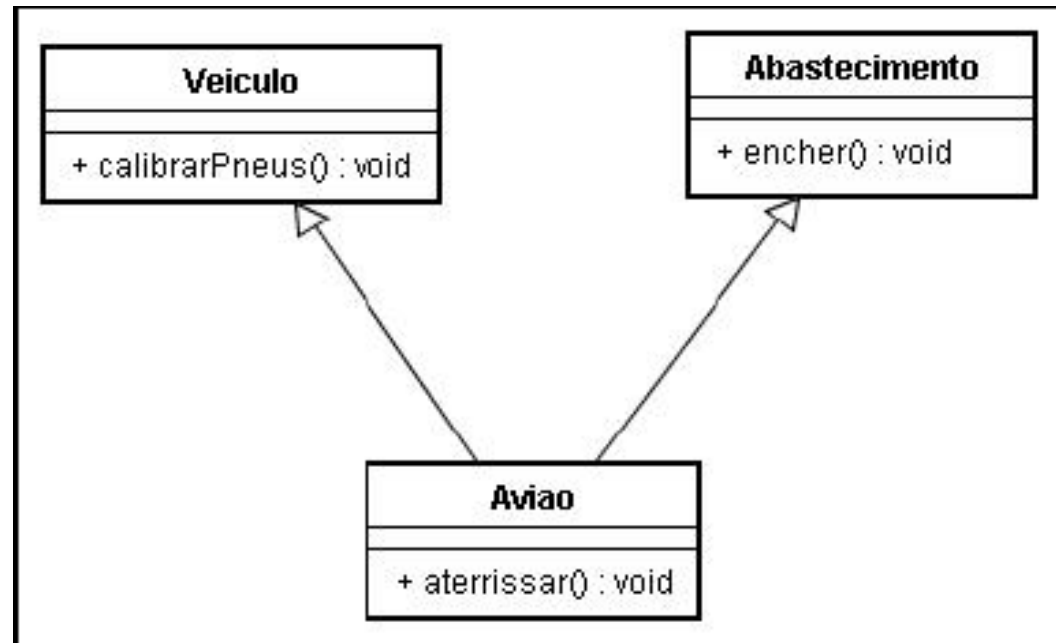
Suponha que seja necessário implementar um método para abastecer. O que poderia ser feito? Há algum problema envolvido?

# Aplicação de Interfaces

## ■ Considerações

- Definir métodos distintos para abastecimento nas classes Aviao e Carro introduziria uma redundância indesejável.
- Logo, como resolver este dilema
  - Não podemos declarar um método para abastecimento em Veículo, pois nem todas as subclasses podem ser abastecidas. Também não podemos declarar um método para cada classe que necessite de abastecimento, pois isso gera redundância de código.

# Aplicação de Interfaces



Uma solução seria criar uma classe **Abastecimento**, com um método para abastecer. Assim, toda classe que necessite de abastecimento, iria herdar da classe **Abastecimento**

Porque isso é problemático em Java?

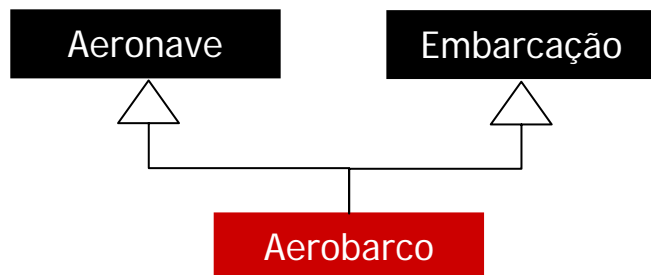
# Aplicação de Interfaces

```
public class Aviao extends Veiculo implements IUsaCombustivel
{
    private double capcTanque=5000.00;
    private double totComb=2000.00;
    public void aterrisar(){};
    public double encher(TipoComb tipo,double qtd)
    {
        if(tipo!=TipoComb.Querosene)
            return qtd;
        double falta=capcTanque-totComb;
        if(qtd>falta)
        {
            totComb=capcTanque;
            return qtd-falta;
        }
        else
        {
            totComb+=qtd;
            return 0.0;
        }
    }
}
```

# Aplicação de Interfaces

```
public class Carro extends Veiculo implements IUsaCombustivel
{
    private double capcTanque=50.00;
    private double totComb=20.00;
    public void dirigir(){};
    public double encher(TipoComb tipo,double qtd)
    {
        if(tipo!=TipoComb.Alcool &&
            tipo!=TipoComb.Gasolina)
            return qtd;
        double falta=capcTanque-totComb;
        if(qtd>falta)
        {
            totComb=capcTanque;
            return qtd-falta;
        }
        else
        {
            totComb+=qtd;
            return 0.0;
        }
    }
}
```

# Herança Múltipla



```
interface Aeronave
{ int navegar(Ponto origem, Ponto destino);
  void decolar();
  void aterrisar();
  void abastecer(double combustivel);
}

interface Embarcacao
{ int navegar(Ponto origem, Ponto destino);
  void ancorar();
  void desancorar();
}
```



```
class Aerobarco implements Aeronave, Embarcacao
{ int navegar(Ponto origem, Ponto destino) { ... };
  void decolar() { ... };
  void aterrisar() { ... };
  void abastecer(double combustivel) { ... };
  void ancorar() { ... };
  void desancorar() { ... };
}
```



# Herança Múltipla

```
interface Aeronave
{ int navegar(Ponto origem, Ponto destino);
  void decolar();
  void aterrisar();
  void abastecer(double combustivel);
}

interface Embarcacao
{ int navegar(Ponto origem, Ponto destino);
  void ancorar();
  void desancorar();
}
```



```
class Helicoptero implements Aeronave
{ int navegar(Ponto origem, Ponto destino) { ... };
  void decolar() { ... };
  void aterrisar() { ... };
  void abastecer(double combustivel) { ... };
  void pairar() { ... };
}
```

# Interfaces

---

## Orientação a Objetos - DCC025

Prof. Edmar Welington Oliveira  
oliveira.edmar@ufjf.edu.br

Universidade Federal de Juiz de Fora - UFJF  
Departamento de Ciência da Computação - DCC