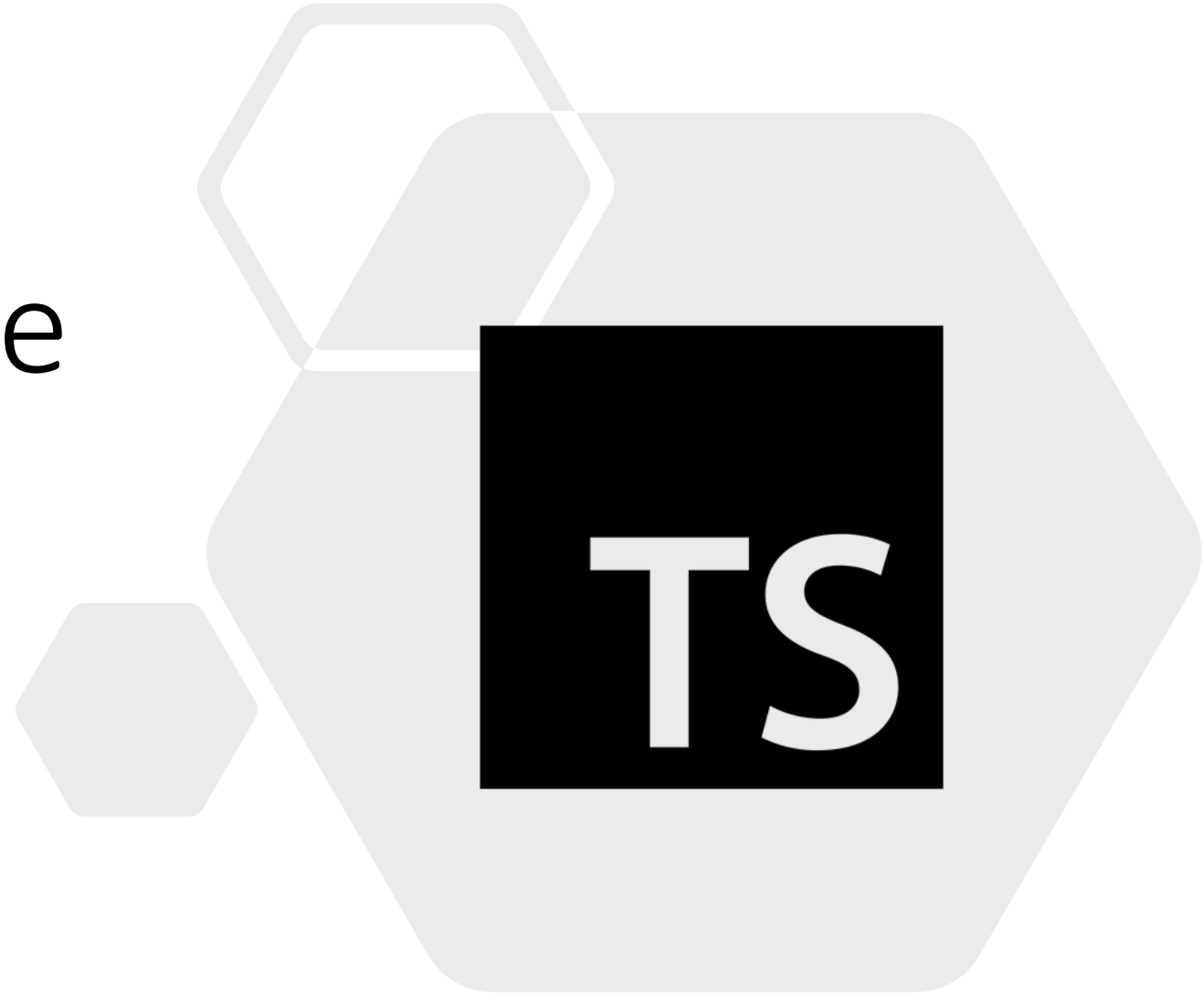
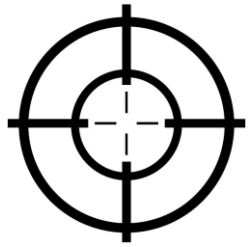
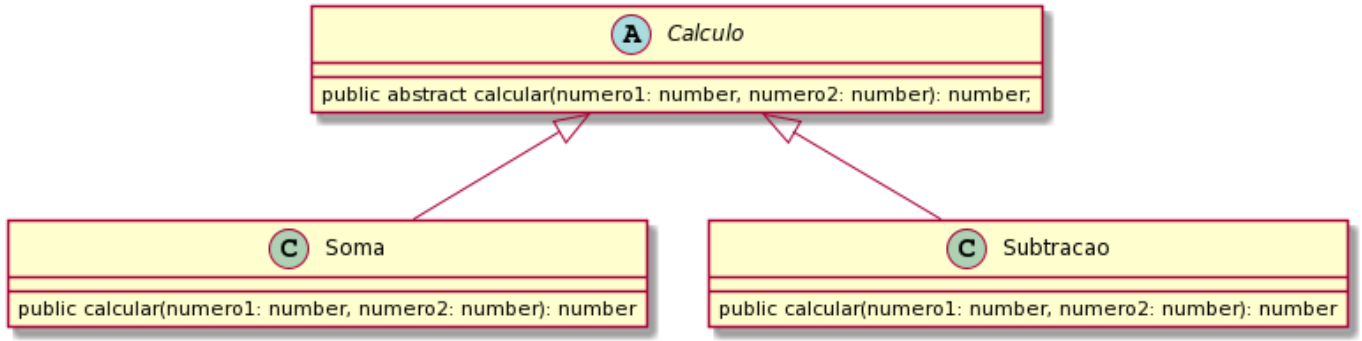
Polimorfismo e interface



Uso comum do polimorfismo...

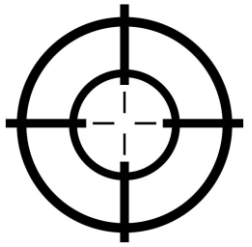


Geralmente, aplica-se o polimorfismo utilizando uma herança, ou seja, uma classe que deriva de outra classe base ou superior. Este tipo de polimorfismo é denominado de polimorfismo de subtipo.



Este tipo de polimorfismo também é conhecido como polimorfismo de tempo de execução.

Outras formas de polimorfismo...



Existem outras formas de aplicar o polimorfismo. Contudo, a possibilidade de aplicação delas depende se a linguagem de programação utilizada permite.

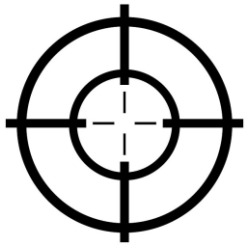
Para a linguagem TypeScript destacam se os polimorfismos de subtipo, sobrecarga e paramétrico.

Sobrecarga

Este tipo de polimorfismo é conhecido como sobrecarga de método. Com a sobrecarga de método, o programador pode implementar vários métodos com o mesmo nome, mas com parâmetros diferentes em quantidade ou tipo.

Este tipo de polimorfismo também é chamado de polimorfismo de ligação estática ou tempo de compilação.

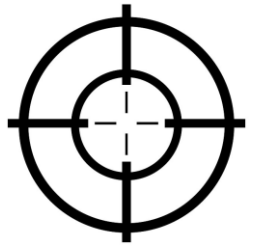
Aplicando a sobrecarga...



Todos os métodos são construtores da mesma classe, possuem o mesmo nome, mas não a mesma assinatura.

```
export default class Empresa {
    private razaoSocial!: string
    private cpnj!: string
    private nomeFantasia!: string

    constructor(razaoSocial: string, cpnj: string, nomeFantasia: string)
    constructor(razaoSocial: string, cpnj: string)
    constructor()
    constructor(razaoSocial?: string, cnpj?: string, nomeFantasia?: string) {
        if (razaoSocial !== undefined && cnpj !== undefined && nomeFantasia !== undefined) {
            this.razaoSocial = razaoSocial
            this.cpnj = cnpj
            this.nomeFantasia = nomeFantasia
        }
        if (razaoSocial !== undefined && cnpj !== undefined && nomeFantasia === undefined) {
            this.razaoSocial = razaoSocial
            this.cpnj = cnpj
        }
    }
}
```

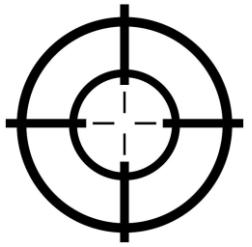
Implementação única...



A linguagem TypeScript permite declarar sobrecargas nas assinaturas dos métodos, mas, a implementação dos métodos deve ser apenas uma. Além disso, essa implementação deve ter uma assinatura que seja compatível com todas as sobrecargas.

```
constructor(razaoSocial: string, cpnj: string, nomeFantasia: string)
constructor(razaoSocial: string, cpnj: string)
constructor()
constructor(razaoSocial?: string, cnpj?: string, nomeFantasia?: string) {
    if (razaoSocial !== undefined && cnpj !== undefined && nomeFantasia !== undefined) {
        this.razaoSocial = razaoSocial
        this.cpnj = cnpj
        this.nomeFantasia = nomeFantasia
    }
    if (razaoSocial !== undefined && cnpj !== undefined && nomeFantasia === undefined) {
        this.razaoSocial = razaoSocial
        this.cpnj = cnpj
    }
}
```

Implementação única...



```
constructor(razaoSocial: string, cpnj: string, nomeFantasia: string)
constructor(razaoSocial: string, cpnj: string)
constructor()
constructor(razaoSocial?: string, cnpj?: string, nomeFantasia?: string) {
    if (razaoSocial !== undefined && cnpj !== undefined && nomeFantasia !== undefined) {
        this.razaoSocial = razaoSocial
        this.cpnj = cnpj
        this.nomeFantasia = nomeFantasia
    }
    if (razaoSocial !== undefined && cnpj !== undefined && nomeFantasia === undefined) {
        this.razaoSocial = razaoSocial
        this.cpnj = cnpj
    }
}
```

A implementação única deve “cobrir” todas as possibilidades de sobrecarga.

O símbolo “?” indica que o argumento pode ser passado ou não para o método.

Utilizando a sobrecarga...

```
import Empresa from "./empresa";

                                  3/3  Empresa(): Empresa

let empresa = new Empresa()
```

Três possíveis formas de criar o objeto.

```
import Empresa from "./empresa";

                                  2/3  Empresa(razaoSocial: string, cpnj: string): Empresa

let empresa = new Empresa()
```

```
import Empresa from "./empresa";

                                  1/3  Empresa(razaoSocial: string, cpnj: string,
                                       nomeFantasia: string): Empresa

let empresa = new Empresa()
```

```
let empresa = new Empresa()
let empresa2 = new Empresa('ABC LTDA','999.999.999-99')
let empresa3 = new Empresa('ABC LTDA','999.999.999-99','Mercado Online')
```

Vantagens da sobrecarga de métodos...

Qualquer método pode ser sobrecarregado, não apenas construtores. A sobrecarga de métodos reduz a complexidade do código e impede o desenvolvimento de métodos diferentes para a mesma funcionalidade com uma assinatura diferente.

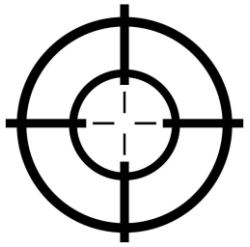
A sobrecarga de método aumenta a legibilidade do programa. Isso fornece flexibilidade aos programadores para que eles possam chamar o mesmo método para diferentes tipos ou quantidades de dados.

Dificuldades com a sobrecarga...

A linguagem TypeScript permite apenas uma implementação para várias assinaturas. Isto obriga ao desenvolvedor o tratar a complexidade da variação de parâmetros.

A sobrecarga não remove a necessidade da asserção de atribuição definitiva.

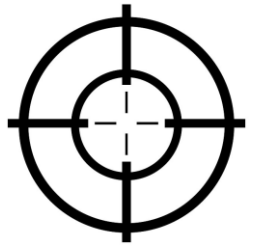
Polimorfismo paramétrico



Este tipo de polimorfismo é baseado na programação genérica.

Programação genérica é um estilo de programação no qual algoritmos são escritos em termos de tipos a serem especificados posteriormente, que são instanciados quando necessário para tipos específicos fornecidos como parâmetros.

Aplicando o polimorfismo paramétrico...



```
export default class Lista<T>{
    private dados: T[]
    constructor(dados: T[]) {
        this.dados = dados
    }

    public quantidade(): number{
        return this.dados.length
    }

    public pegarDado(posicao:number): T{
        return this.dados[posicao]
    }
}
```

O símbolo “T” representa um tipo que será colocado (substituído) no momento que o objeto for criado.

```
let textos: string[] = []
let lista  = new Lista(textos)
console.log(`O tamanho da lista é: ${lista.quantidade()}`)
```

Por que tipos genéricos são importantes?

Eles foram projetados para estender o sistema de tipos da linguagem TypeScript e permitir que um objeto ou método possa operar em outros elementos de vários tipos enquanto fornece segurança de tipo em tempo de execução.

Lembre-se, este recurso não é obrigatório no desenvolvimento com TypeScript. TypeScript é uma linguagem dinamicamente tipada. O transcompilador pode inferir o tipo de uma variável dinamicamente dependendo do seu valor.

Tipo genérico versus tipo “any”

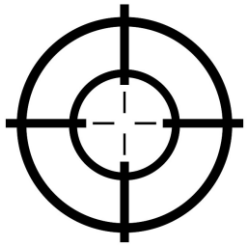
O tipo “any” possibilita a flexibilização na tipagem de variáveis, por inferência. Contudo, o tipo “any” não permite a segurança de dados.

```
export default class Lista{
    private dados: any[]
    constructor(dados: any[]) {
        this.dados = dados
    }

    get obterDados(): any[]{
        return this.dados
    }
}
```

```
let coisas = ['1',2,'a']
let lista  = new Lista(coisas)

let somatorio = 0
for (let coisa of lista.obterDados) {
    somatorio = somatorio + coisa
}
console.log(`Resultado do somatorio: ${somatorio}`)
```

Faz sentido um somatório de caracteres e números?

Por que aprender a usar genéricos?



A programação genérica permite a verificação de tipo mais abrangente, a eliminação de conversões e a capacidade de desenvolver algoritmos genéricos.

Uma das principais vantagens dos tipos genéricos é a segurança de tipo (TypeSafe). Segurança de tipo significa que as variáveis são verificadas estaticamente para atribuição apropriada em tempo de execução.

Sobre tipos genéricos...

Classes genéricas podem ter mais de um tipo genérico nos parâmetros, separados por vírgulas, na declaração da classe.

```
export default class Lista<T,X>{
    private dados: T[]
    private informacao: X
    constructor(dados: T[], informacao: X) {
        this.dados = dados
        this.informacao = informacao
    }
}
```

Desenvolver utilizando tipos genéricos pode ser interessante. Contudo, sua aplicação exige atenção ao desenvolvedor. Além disso, utilizar tipos genéricos por motivos inadequados pode aumentar a complexidade do código, dificultando sua manutenção ou atualização.

O que é herança múltipla?

A herança múltipla é um recurso de algumas linguagens de programação orientadas à objetos em que uma classe filha pode herdar características e recursos de mais de uma classe pai.

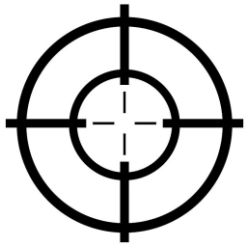
Herança múltipla significa que uma subclasse pode herdar de duas ou mais superclasses. Algumas linguagens, como C++, permitem a herança múltipla, mas TypeScript permite apenas herança única, ou seja, uma subclasse pode herdar de apenas uma superclasse.

É possível fazer herança múltipla em TypeScript?

Algumas linguagens de programação que seguem o paradigma de programação orientada à objetos não permitem herança múltipla. Isto acontece com o TypeScript. Contudo, existem situações em que aplicar este recurso pode ser vantajoso.

Como uma alternativa à herança múltipla, a linguagem TypeScript oferece o conceito de Interface. As interfaces também servem para refinar a aplicação do polimorfismo, ou seja, auxiliam a aplicação do polimorfismo em contextos variados.

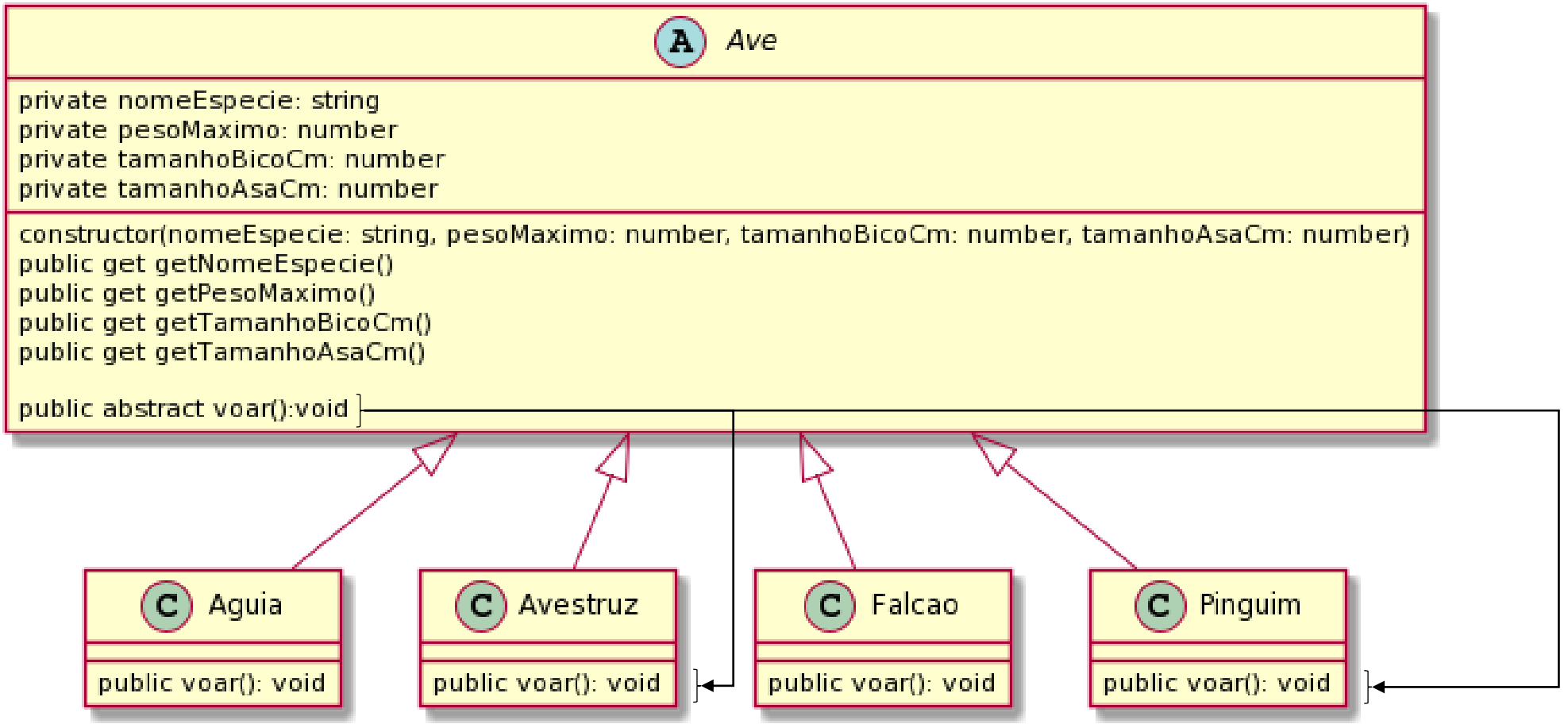
O que é uma interface?



Uma interface é uma forma de definir um protocolo de comportamento que pode ser implementado por uma classe. Uma interface é um contrato, que defini um comportamento obrigatório para a classe.

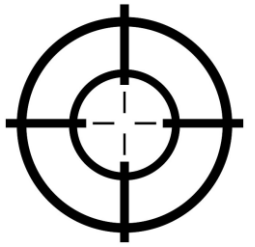
As interfaces são úteis para declarar métodos que uma ou mais classes devem implementar simulando a herança múltipla. Podem ser utilizadas para atribuir comportamentos comuns a um grupo de classes sem alterar sua hierarquia de herança ou obrigar todas as demais classes da hierarquia a ter o mesmo comportamento.

Herança sem sentido...



Faz algum sentido a inclusão de um método voar para uma classe que representa uma ave que não é voadora?

Herança sem sentido...



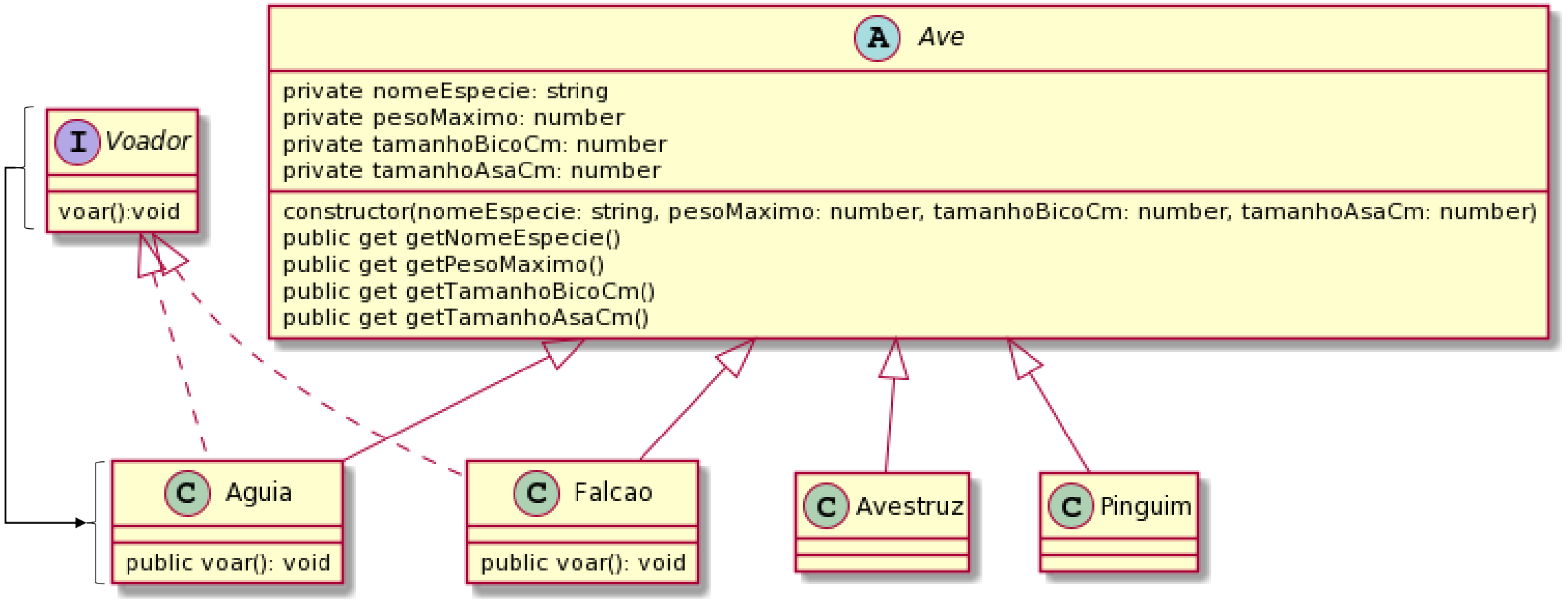
```ts
export default abstract class Ave {
    private nomeEspecie: string
    private pesoMaximo: number
    private tamanhoBicoCm: number
    private tamanhoAsaCm: number
    constructor(nomeEspecie: string, pesoMaximo: number, tamanhoBicoCm: number, tamanhoAsaCm: number) {
        this.nomeEspecie = nomeEspecie
        this.pesoMaximo = pesoMaximo
        this.tamanhoBicoCm = tamanhoBicoCm
        this.tamanhoAsaCm = tamanhoAsaCm
    }
    public get getNomeEspecie(){
        return this.nomeEspecie
    }
    public get getPesoMaximo(){
        return this.pesoMaximo
    }
    public get getTamanhoBicoCm(){
        return this.tamanhoBicoCm
    }
    public get getTamanhoAsaCm(){
        return this.tamanhoAsaCm
    }

    public abstract voar():void
}
```

```ts
export default class Aguia extends Ave{
    public voar(): void {
        console.log(`Voar até 3.000m`)
    }
}
```

```ts
export default class Pinguim extends Ave{
    public voar(): void {
    }
}
```

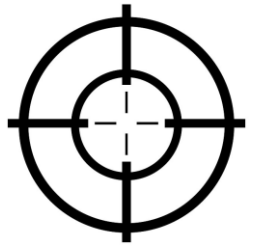
Não é interessante deixar um método em branco ou permitir um comportamento errado.

Entendendo interface...



Com a interface o comportamento é aplicado, obrigatoriamente, somente nas classes que precisam tê-lo.

Implementação de uma interface



```
export default interface Voador{
    voar():void
}
```

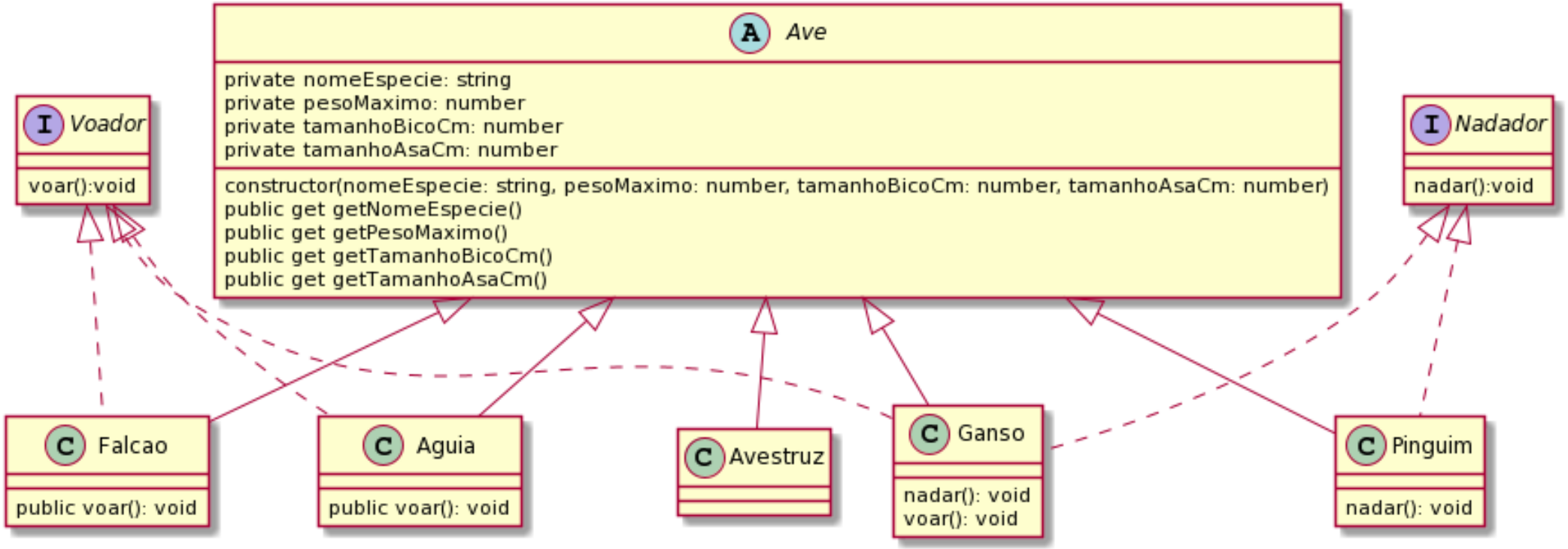
```
export default class Aguia extends Ave implements Voador{
    public voar(): void {
        console.log(`Voar até 3.000m`)
    }
}
```

```
export default class Falcao extends Ave implements Voador{
    public voar(): void {
        console.log(`Voar até 3.200m`)
    }
}
```

```
export default class Pinguim extends Ave{
}
```

A implementação da interface não impede a herança. Sua utilização permite separar o comportamento desejado apenas para as classes que a implementam.

Abstração de comportamentos



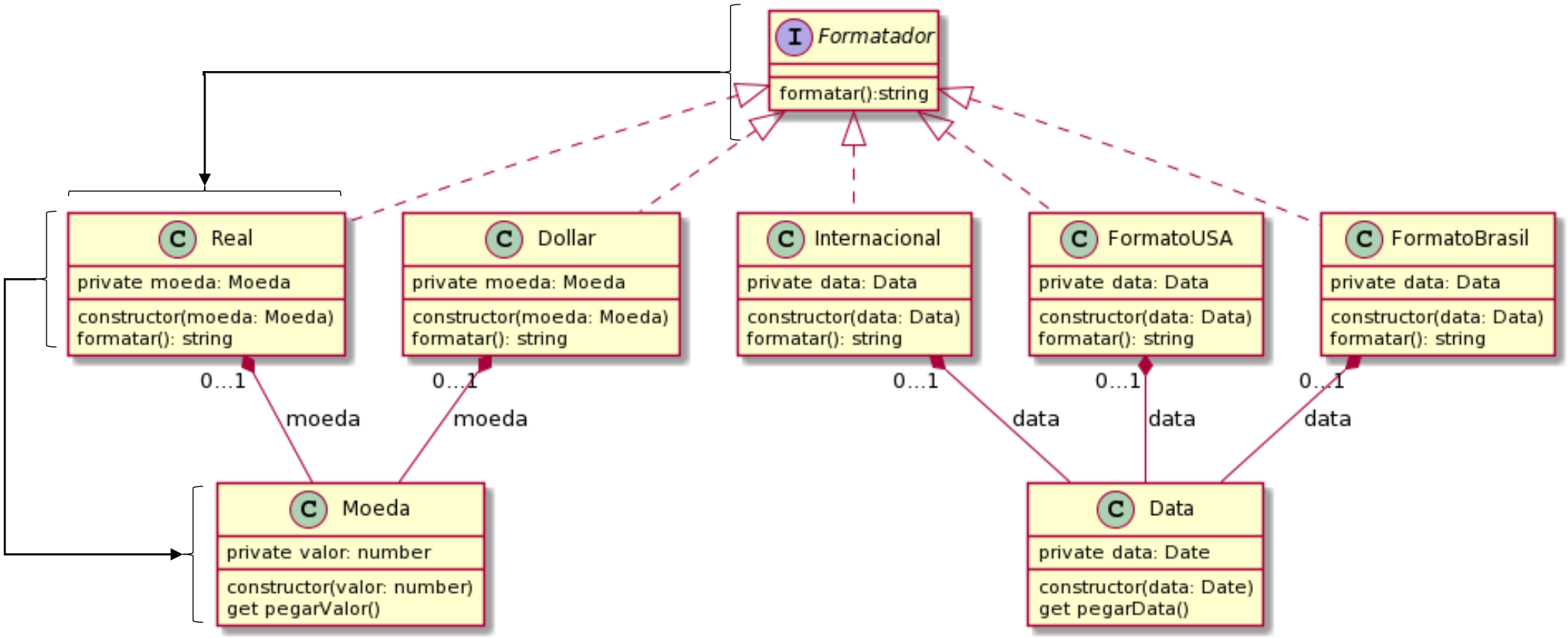
Uma classe pode implementar mais de uma interface. A implementação de várias interfaces não afeta ou impede a herança.

Implementação de várias interfaces

```
export default interface Voador{
    voar():void
}

export default interface Nadador{
    nadar():void
}

export default class Ganso extends Ave implements Voador, Nadador{
    nadar(): void {
        console.log(`Voar até 1m`)
    }
    voar(): void {
        console.log(`Voar até 9.000m`)
    }
}
```

A classe deve implementar todos os métodos definidos por cada uma das interfaces que ela implementa.

Refinamento do polimorfismo...



Pode-se criar inúmeras estruturas polimórficas com interfaces, ainda que as classes não pertençam a mesma hierarquia de herança.

Flexibilização de código

```
let formatador: Formatador

let moeda = new Moeda(50.3)
formatador = new Real(moeda)
console.log(formatador.formatar())

formatador = new Dollar(moeda)
console.log(formatador.formatar())

let data = new Data(new Date())
formatador = new FormatoBrasil(data)
console.log(formatador.formatar())

formatador = new FormatoUSA(data)
console.log(formatador.formatar())
```

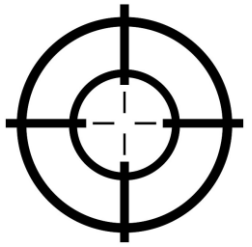
```
export default interface Formatador{
    formatar():string
}
```

```
export default class Dollar implements Formatador {
    private moeda: Moeda
    constructor(moeda: Moeda) {
        this.moeda = moeda
    }
    formatar(): string {
        return `US$ ${this.moeda.pegarValor}`
    }
}
```

```
export default class Real implements Formatador {
    private moeda: Moeda
    constructor(moeda: Moeda) {
        this.moeda = moeda
    }
    formatar(): string {
        let literal = new String(this.moeda.pegarValor)
        let comVirgula = literal.replace('.', ',')
        return `R$ ${comVirgula}`
    }
}
```

A interface aumenta o poder do polimorfismo.

Sobre interface



Uma interface pode definir uma série de métodos, mas nunca conter suas implementações. Ela só expõe o que o objeto deve fazer, e não como ele o faz, nem o que ele tem. Como ele o faz será definido em uma implementação dessa interface, na classe do objeto.

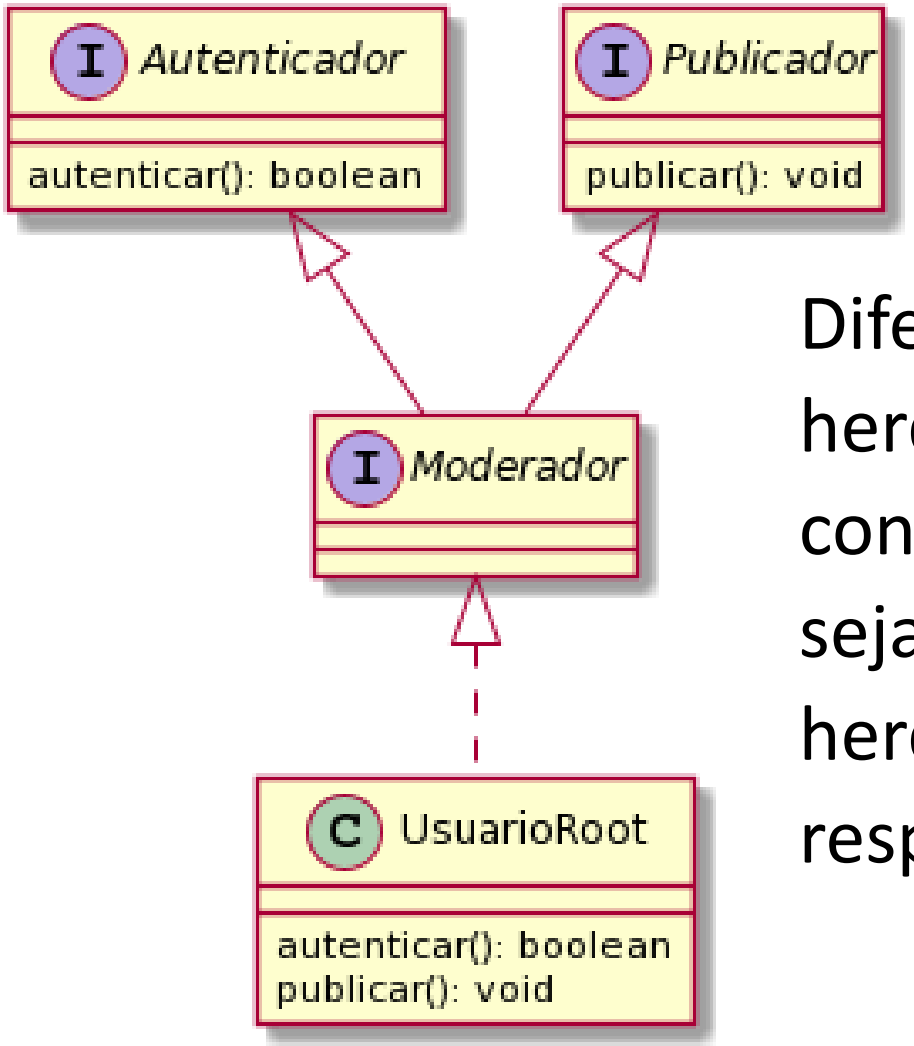
Uma interface é um contrato que quem assina se responsabiliza por implementar seus métodos e, com isso, cumprir o contrato. O cumprimento do contrato é o que garante a aplicação do polimorfismo.

Regra de ouro 

A maneira pela qual os objetos se comunicam em um sistema orientado à objetos é muito mais importante do que como eles executam. O que um objeto faz é mais importante do que como ele o faz.

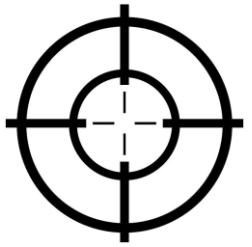
Aqueles que seguem essa regra terão sistemas mais fáceis de manter e modificar. Conforme já se percebeu, essa é uma das ideias principais do paradigma de programação orientada à objetos, provavelmente, a mais importante.

Herança entre interfaces



Diferentemente das classes, uma interface pode herdar de mais de uma interface. É como um contrato o qual depende que outros contratos sejam fechados antes daquele valer. Não se herda métodos e atributos, mas, sim, responsabilidades.

Herança entre interfaces



```
export default interface Autenticador {
    autenticar(): boolean
}
```

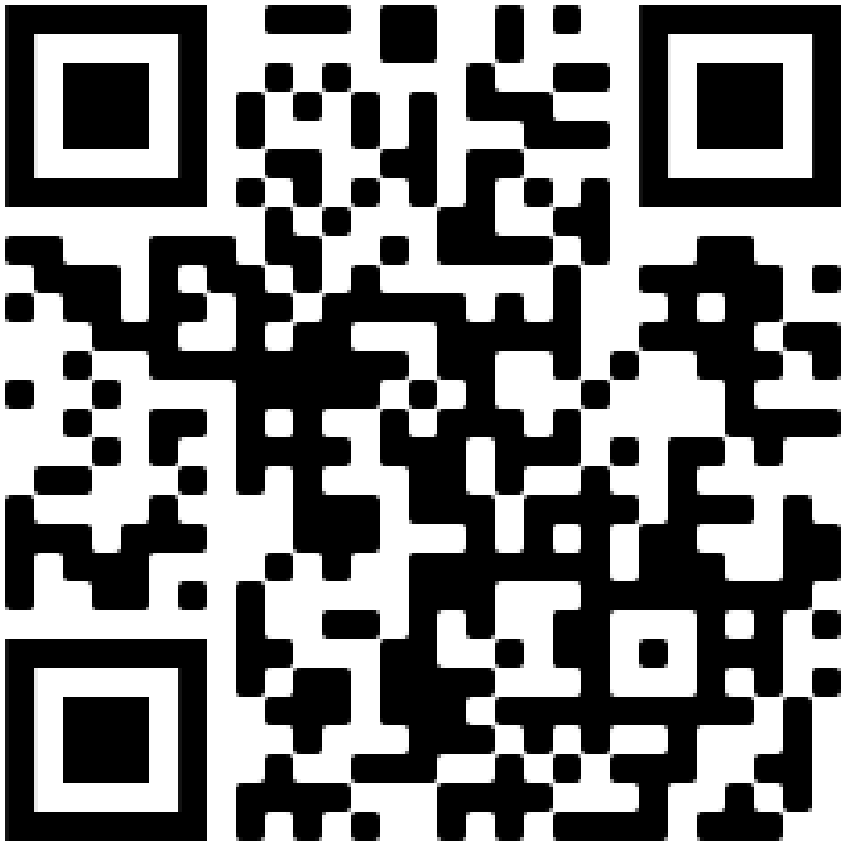
```
export default interface Publicador {
    publicar(): void
}
```

```
export default interface Moderador extends Autenticador, Publicador{
}
```

```
export default class UsuarioRoot implements Moderador{
    autenticar(): boolean {
        return true
    }
    publicar(): void {
        console.log(`Efetuar publicação`)
    }
}
```

A classe deve implementar todos os métodos definidos por todas as interfaces na herança.

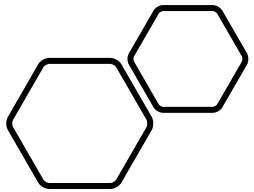
https://github.com/gerson-pn/interface-typescript



Importante!

Interfaces representam uma barreira no aprendizado de linguagens de programação que seguem o paradigma de programação orientada à objetos, como Java, C# e TypeScript.

Alguns desenvolvedores percebem interfaces como códigos que não servem para nada, uma vez que é preciso escrever o corpo dos métodos nas classes implementadoras das interfaces. Essa é uma maneira errada de se pensar. O objetivo do uso de uma interface é deixar o código mais flexível e possibilitar a mudança de implementação sem maiores traumas. Não é apenas um código de prototipação ou um cabeçalho!



TypeScript