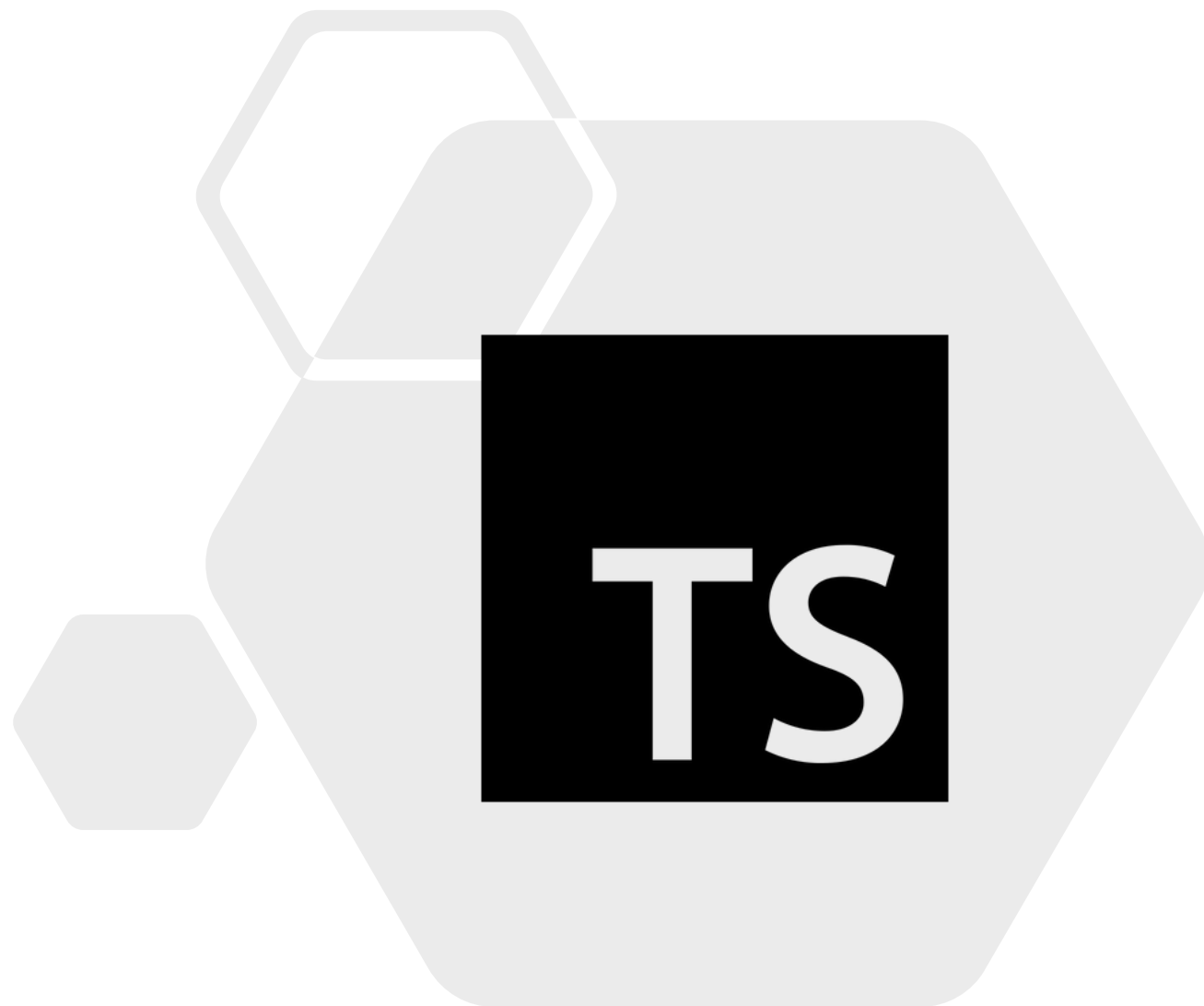
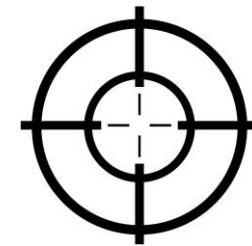


# Herança e polimorfismo



# Separando as coisas...

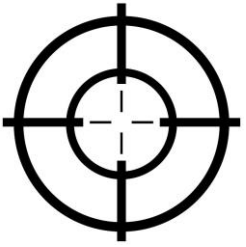


O termo polimorfismo é definido separadamente em quatro disciplinas científicas: biologia, genética, bioquímica e ciência da computação.

No contexto biológico, polimorfismo se refere à ideia de que membros de uma mesma população ou colônia podem assumir diferentes características visuais.

Na genética, o polimorfismo reflete a variação no código genético dos indivíduos em uma população.

# Separando as coisas...



No campo da bioquímica, o termo polimorfismo é usado para descrever diferenças estruturais sutis em proteínas que, de outra forma, são idênticas.

Olhar para as raízes gregas do termo polimorfismo pode ajudar a esclarecer ainda mais as semelhanças entre essas definições. A palavra poli significa “muitos” e a palavra metamorfose significa “forma”, portanto, quando fala-se sobre um polimorfismo, está-se falando sobre algo que aparece em muitas formas diferentes.

# Vamos a nossa definição!



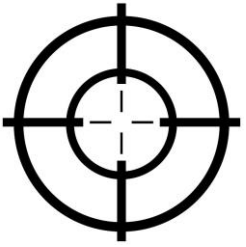
A definição da ciência da computação de polimorfismo, pode ser oferecida em três versões diferentes, para clareza máxima:

O polimorfismo é um recurso das linguagens de programação orientadas à objetos, que permite que uma rotina específica use variáveis de tipos diferentes, com nomes iguais, em momentos diferentes.

Polimorfismo é a capacidade de uma linguagem de programação de apresentar a mesma interface para vários tipos de dados subjacentes diferentes.

Polimorfismo é a capacidade de diferentes objetos responderem de uma maneira única à mesma mensagem.

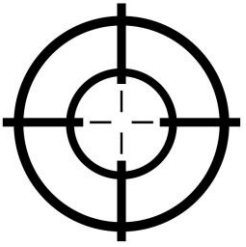
# Por que usar o polimorfismo?



O polimorfismo é uma das principais características de qualquer linguagem de programação, que segue o paradigma de programação orientada à objetos. Linguagens como Ruby, Java, TypeScript, C++ e C# suportam polimorfismo.

O código polimórfico permite que um programa processe objetos de diferentes maneiras, dependendo de seu tipo ou classe, com a capacidade de redefinir métodos para classes derivadas.

# Herança



Geralmente, o polimorfismo é obtido a partir da aplicação de outro recurso, que vem do paradigma de programação orientada à objetos. Este recurso é chamado de herança.

Herança é um mecanismo no qual uma classe adquire a propriedade de outra classe. Com a herança, pode-se reutilizar os campos e métodos de uma classe existente. Consequentemente, a herança facilita a reutilização.

# Aplicando herança...



```
export default abstract class Calculo {  
  public abstract calcular(numero1: number, numero2: number): number;  
}  
  
import Calculo from "./calculo";  
  
export default class Soma extends Calculo {  
  public calcular(numero1: number, numero2: number): number {  
    return numero1 + numero2  
  }  
}
```

Uma classe pode estender outra classe, como uma classe base por exemplo. A classe que estende outra é denominada de classe filha ou subclasse. A classe que é estendida por uma ou mais classes é denominada de classe pai ou superclasse.

# O que o abstract faz de fato?



Transforma uma classe em abstrata. Uma classe restrita que não pode ser usada para criar objetos, para acessá-la deve ser herdada por outra classe.

Transforma um método em abstrato. Métodos abstratos só podem ser declarados dentro de classes abstratas. Este tipo de método é declarado sem corpo, porque é uma abstração! O corpo deve ser fornecido pela subclasse.

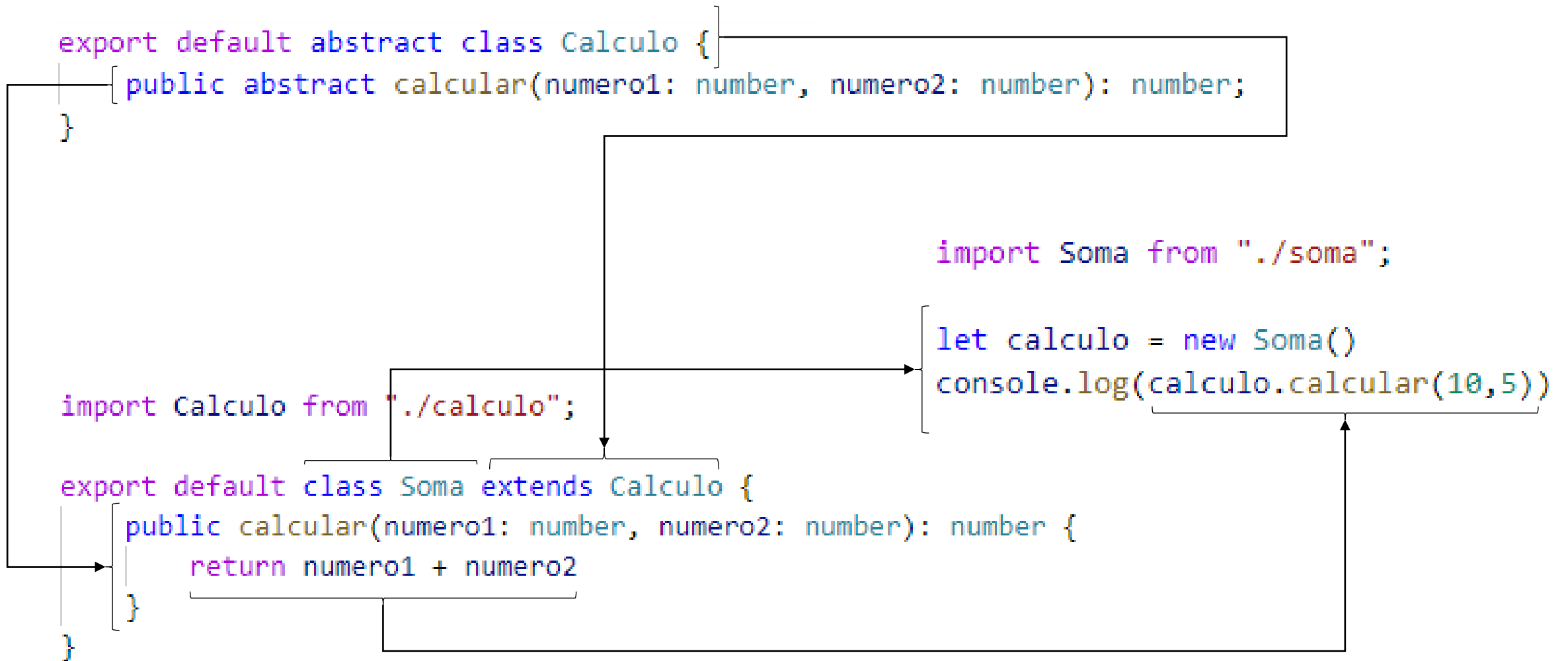
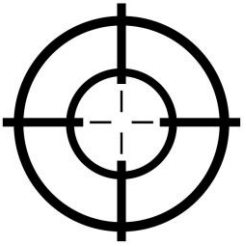


# Extends...

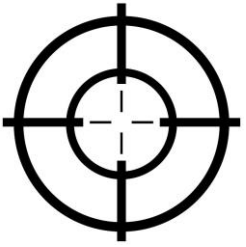
Em TypeScript, a palavra-chave `extends` é usada para indicar que uma classe, que está sendo definida, é derivada de uma classe base, usando herança. Então, basicamente, a palavra-chave `extends` é usada para estender a funcionalidade de uma superclasse para uma subclasse.

Uma classe filha tem todas as propriedades e métodos de sua classe pai e também pode definir membros adicionais.

# Executando o polimorfismo...



# Executando o polimorfismo...



```
export default class Soma extends Calculo {  
  public calcular(numero1: number, numero2: number): number {  
    return numero1 + numero2  
  }  
}
```

```
import Multiplicacao from "../multiplicacao";  
import Soma from "../soma";  
import Subtracao from "../subtracao";
```

```
let calculo = new Soma()  
console.log(calculo.calcular(10,5))
```

```
export default class Subtracao extends Calculo{  
  public calcular(numero1: number, numero2: number): number {  
    return numero1 - numero2  
  }  
}
```

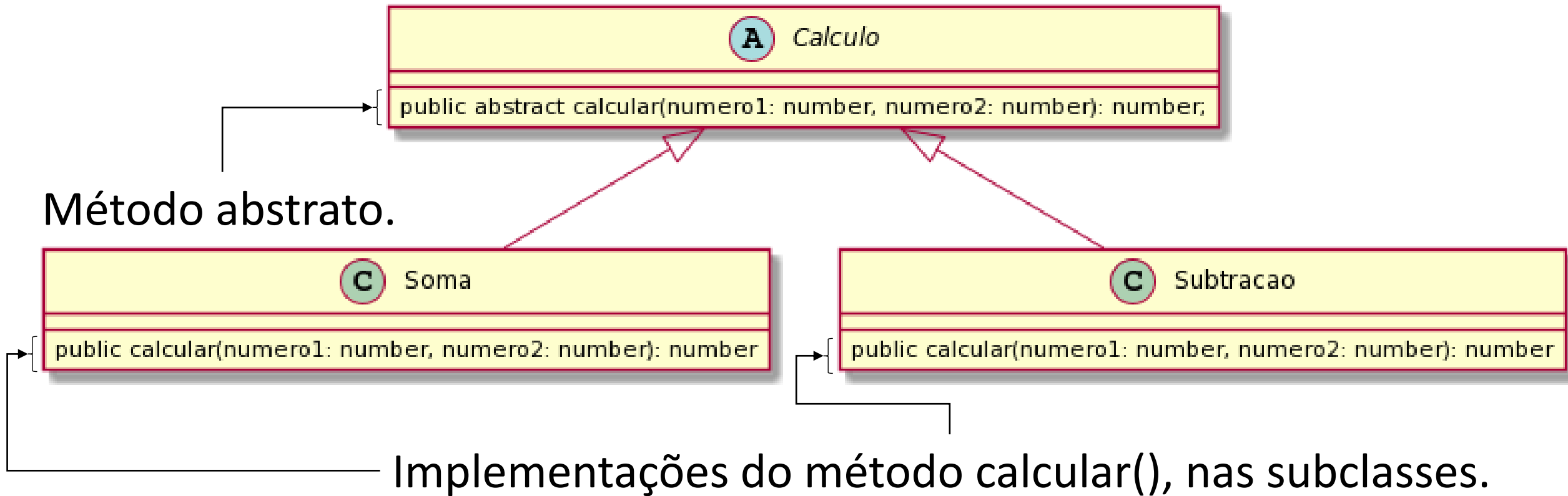
```
calculo = new Multiplicacao()  
console.log(calculo.calcular(2,4))
```

```
calculo = new Subtracao()  
console.log(calculo.calcular(21,4))
```

```
export default class Multiplicacao extends Calculo {  
  public calcular(numero1: number, numero2: number): number {  
    return numero1 * numero2  
  }  
}
```

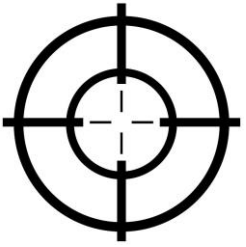
O método calcular() se comporta de modo polimórfico.

# Representação em UML...



Neste diagrama, as classes Soma e Subtração herdam de Calculo e, portanto, são obrigadas a definir como o seu método calcular() deve funcionar.

# Sobrescrever métodos...

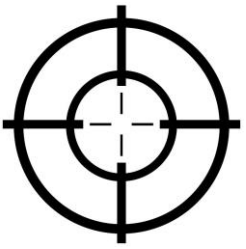


Sobrescrever métodos significa alterar seu comportamento, durante a herança.

Qualquer método pode ser sobrescrito, desde que não seja bloqueado, explicitamente, contra sobrescrição.

Na sobrescrição, a assinatura do método deve ser, sempre, respeitada! Do contrário, não se estará sobrescrevendo, mas sim, criando um método novo.

# O que é assinatura de um método?

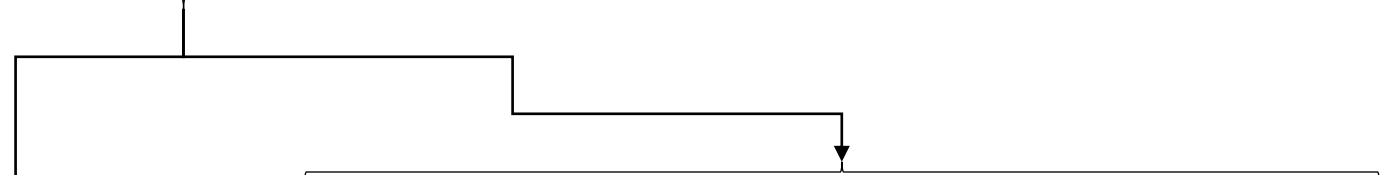


Uma assinatura de método faz parte da declaração do método. É a combinação do nome do método e da lista de parâmetros.


`public abstract calcular(numero1: number, numero2: number): number;`

A bracket is drawn under the signature part of the abstract method declaration: `calcular(numero1: number, numero2: number): number;`. An arrow points from the text above to this bracket.

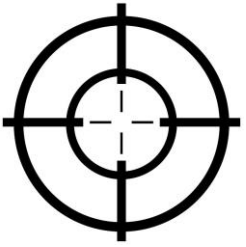
`public calcular(numero1: number, numero2: number): number {  
 return numero1 * numero2  
}`

A bracket is drawn under the signature part of the concrete method declaration: `public calcular(numero1: number, numero2: number): number {`. An arrow points from the abstract signature bracket to this bracket.

`public calcular(numero1: number, numero2: number): number {  
 return numero1 + numero2  
}`

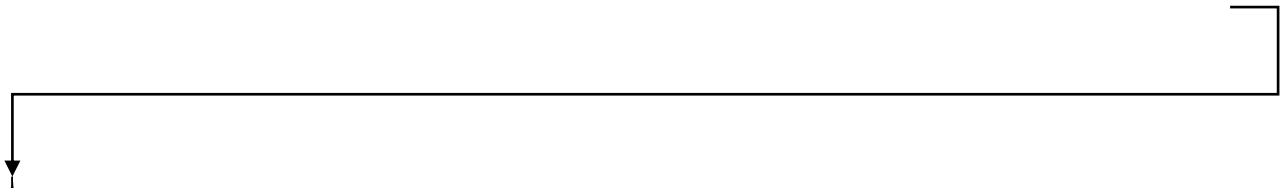
A bracket is drawn under the signature part of the concrete method declaration: `public calcular(numero1: number, numero2: number): number {`. An arrow points from the abstract signature bracket to this bracket.

# Parâmetro ou argumento?



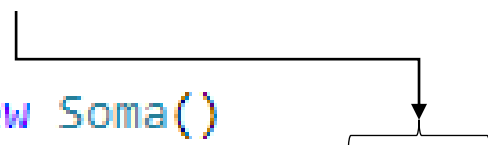
Parâmetros são as variáveis, locais, que são usadas para passar valor para dentro dos métodos.

```
public calcular(numero1: number, numero2: number): number {  
    return numero1 + numero2  
}
```

A diagram consisting of a horizontal line with a downward-pointing arrow at its center. This line originates from the word 'Parâmetros' in the text above and points directly to the parameter list '(numero1: number, numero2: number)' in the code snippet below.

Argumentos são os valores que são passados para dentro dos métodos, passados através das variáveis.

```
let calculo = new Soma()  
console.log(calculo.calcular(10,5))
```

A diagram consisting of a horizontal line with a downward-pointing arrow at its center. This line originates from the word 'Argumentos' in the text above and points directly to the values '10' and '5' in the function call 'calculo.calcular(10,5)' in the code snippet below.

# Sobre classes abstratas...



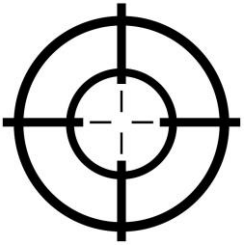
Lembre-se, não é possível criar um objeto diretamente de uma classe abstrata.

Uma classe abstrata pode ter métodos não abstratos, eles também serão herdados, caso ocorra aplicação da herança para uma subclasse.

Na classe abstrata também pode-se aplicar os modificadores de acesso, como o `private` e o `public`.



# Sobre herança...

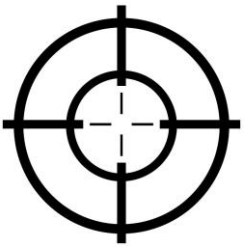


Herança é um dos aspectos mais importantes do paradigma de programação orientada à objetos.

A chave para entender a herança é que ela fornece a reutilização do código. Em vez de escrever o mesmo código, repetidamente, pode-se simplesmente herdar as propriedades de uma classe para a outra.

A herança pode acontecer mesmo que a superclasse não seja abstrata!

# Aproveitando a herança...



```
export default abstract class Veiculo{
  readonly nome:string
  public peso:number
  readonly fabricante:string
```

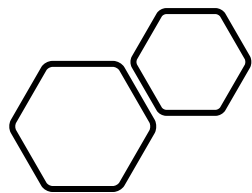
```
  constructor(nome: string, peso: number,fabricante: string){
    this.nome = nome
    this.peso = peso
    this.fabricante = fabricante
  }
```

```
  getDescricao(): string{
    return `Nome: ${this.nome}, peso: ${this.peso}kg, fabricante: ${this.fabricante}`
  }
}
```

```
import Veiculo from "../veiculo";
```

```
export default class Automovel extends Veiculo{
  readonly quantidadeMaximaPassageiros:number
  constructor(nome: string, peso: number,fabricante: string, quantidadeMaximaPassageiros: number){
    super(nome,peso,fabricante)
    this.quantidadeMaximaPassageiros = quantidadeMaximaPassageiros
  }
}
```

Todos os atributos da superclasse são herdados, inclusive métodos, se houverem.



TypeScript

