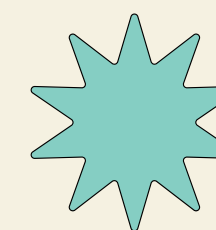
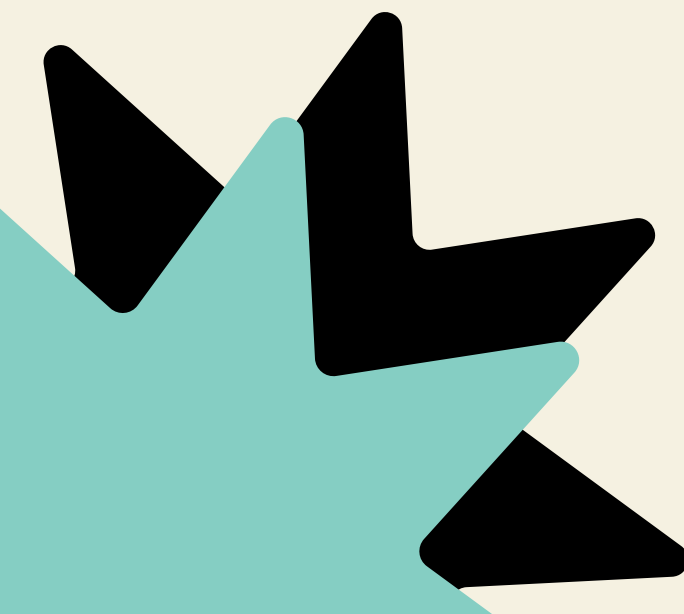
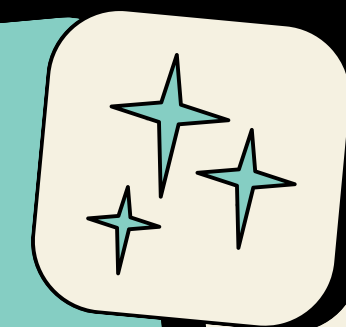
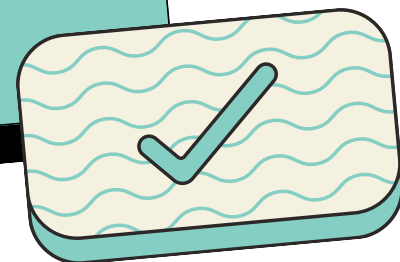
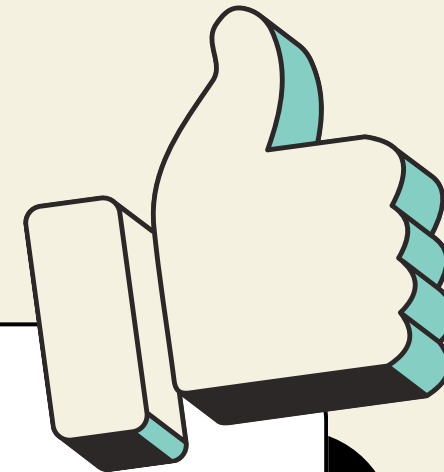


# JAVASCRIPT

## POO



# INDICES



**01**



Pilares

**02**



Associação

**03**

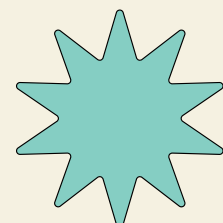
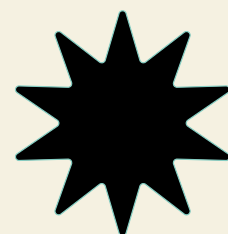


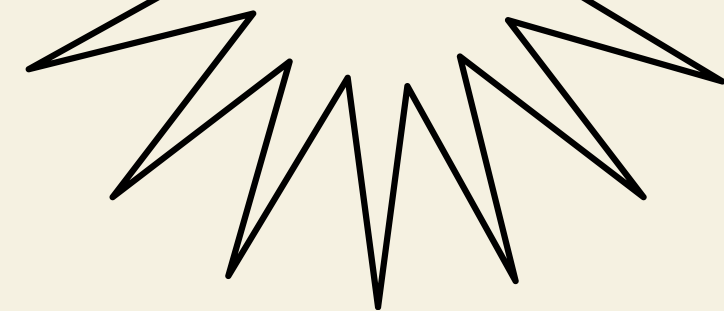
UML

**05**

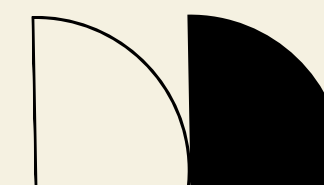


Exercícios





Recapitulação e Introdução ao JavaScript



# ABSTRAÇÃO

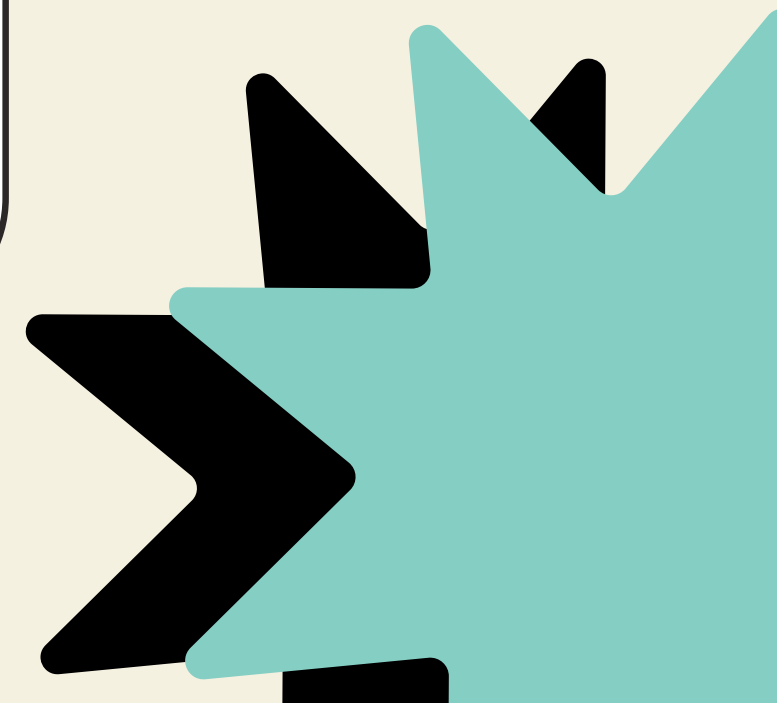
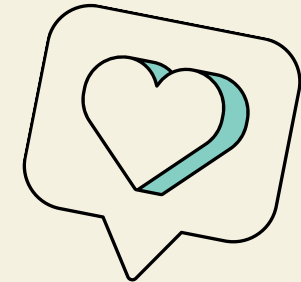


Organiza o código em torno de objetos, que representam entidades do mundo real com atributos e métodos.

- ◆ Foco em modelar comportamentos e responsabilidades
- ◆ Melhora reutilização e manutenção do código



```
class Pessoa {  
  constructor(nome) {  
    this.nome = nome;  
  }  
  
  falar() {  
    console.log(` ${this.nome} está falando.`);  
  }  
}
```



# ENCAPSULAMENTO

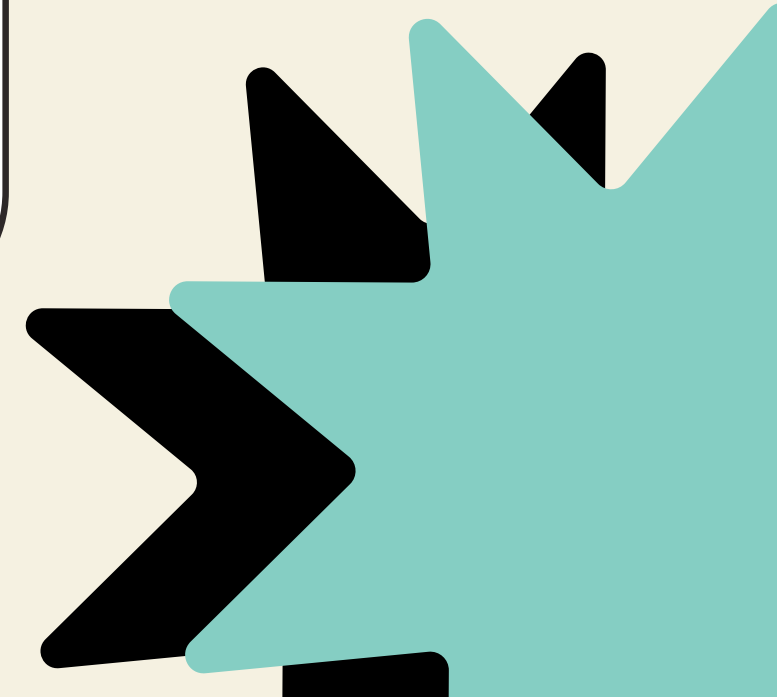
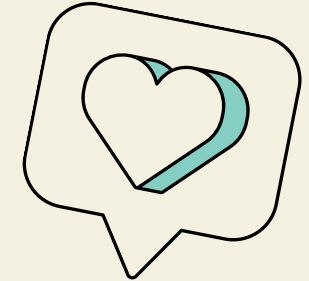


○ Encapsulamento protege os dados internos do objeto e controla seu acesso.

- ◆ Garante segurança e integridade dos dados
- ◆ Usa métodos getters e setters para manipulação controlada



```
class Conta {  
    #saldo = 0; // atributo privado  
  
    depositar(valor) {  
        this.#saldo += valor;  
    }  
  
    getSaldo() {  
        return this.#saldo;  
    }  
}
```



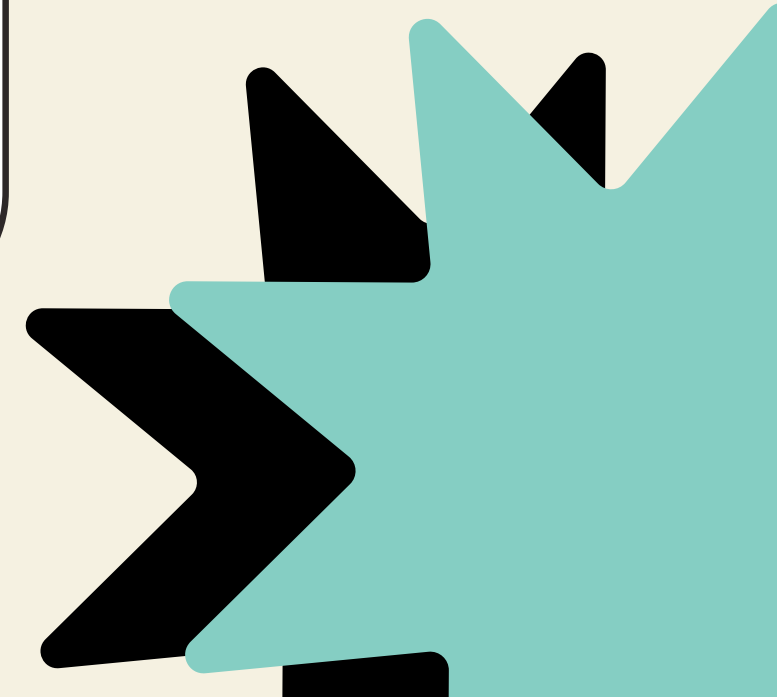
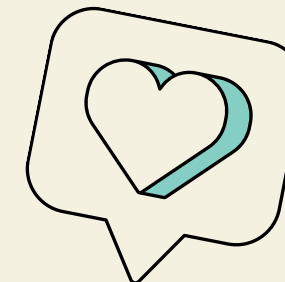
# HERANÇA



A Herança permite que uma classe herde atributos e métodos de outra.

- ◆ Promove reuso de código
- ◆ Define uma relação “é um” entre classes

```
class Animal {  
  falar() {  
    console.log("Som genérico...");  
  }  
}  
  
class Cachorro extends Animal {  
  falar() {  
    console.log("Au au!");  
  }  
}
```

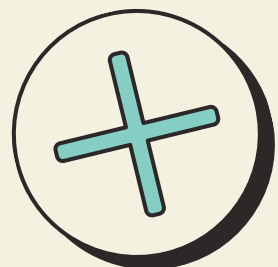


# POLIMORFISMO

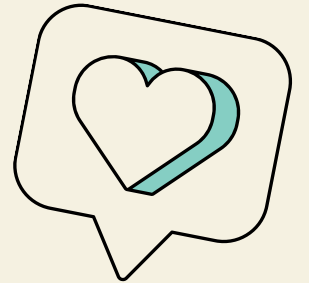


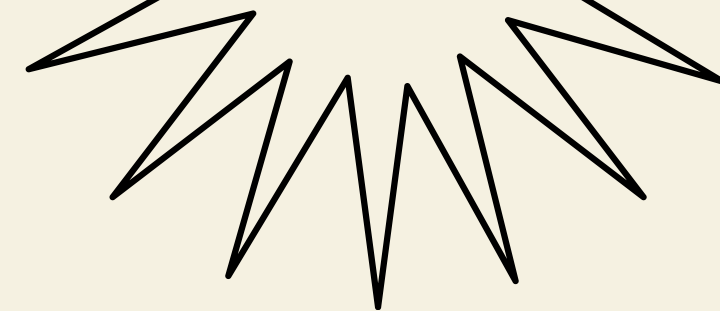
○ Polimorfismo permite que diferentes classes respondam de maneiras distintas ao mesmo método.

- ◆ “Muitas formas” de um mesmo comportamento
- ◆ Facilita extensibilidade do sistema

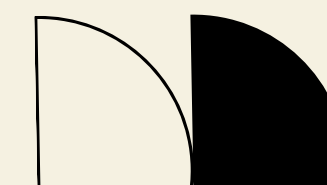


```
class Ave {  
  voar() {  
    console.log("A ave está voando.");  
  }  
}  
  
class Pinguim extends Ave {  
  voar() {  
    console.log("O pinguim não pode voar.");  
  }  
}  
  
const aves = [new Ave(), new Pinguim()];  
aves.forEach(a => a.voar());
```



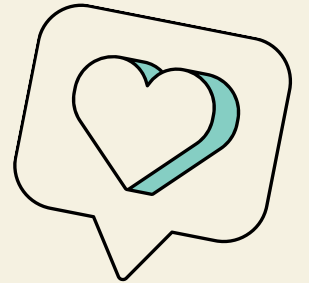
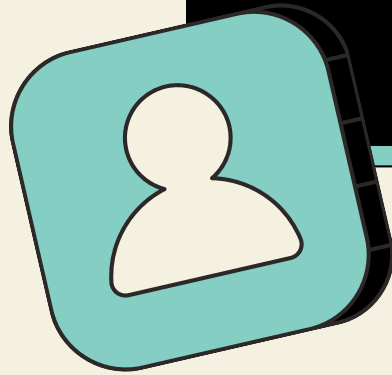


Composição: uma classe contém outra  
Agregação: uma classe usa outras que  
podem existir separadas.

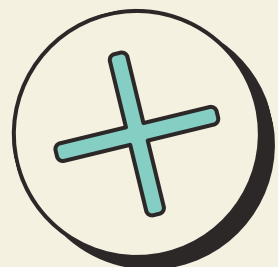




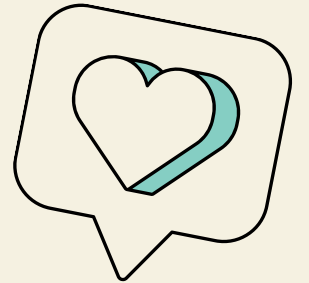
# ASSOCIAÇÃO



```
class Motor {  
  ligar() {  
    console.log("Motor ligado.");  
  }  
}  
  
class Rodas {  
  constructor(qtd) {  
    this.qtd = qtd;  
  }  
}  
  
class Carro {  
  constructor(rodas) {  
    this.motor = new Motor(); // composição  
    this.rodas = rodas;      // agregação  
  }  
}
```



# COLLECTIONS



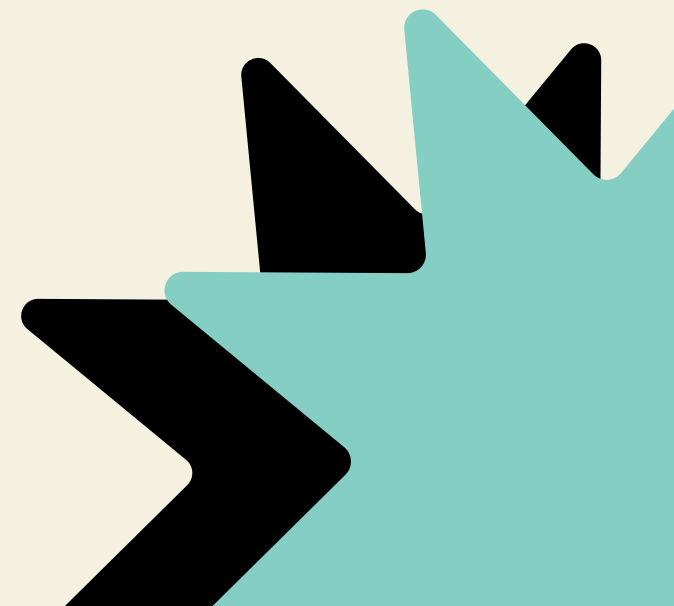
JavaScript possui diversas estruturas para armazenar objetos:

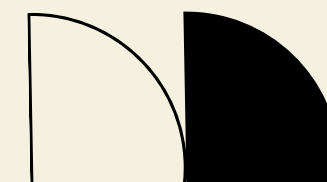
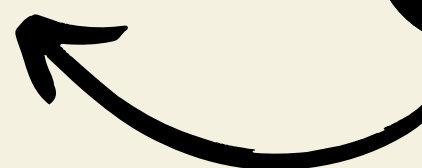
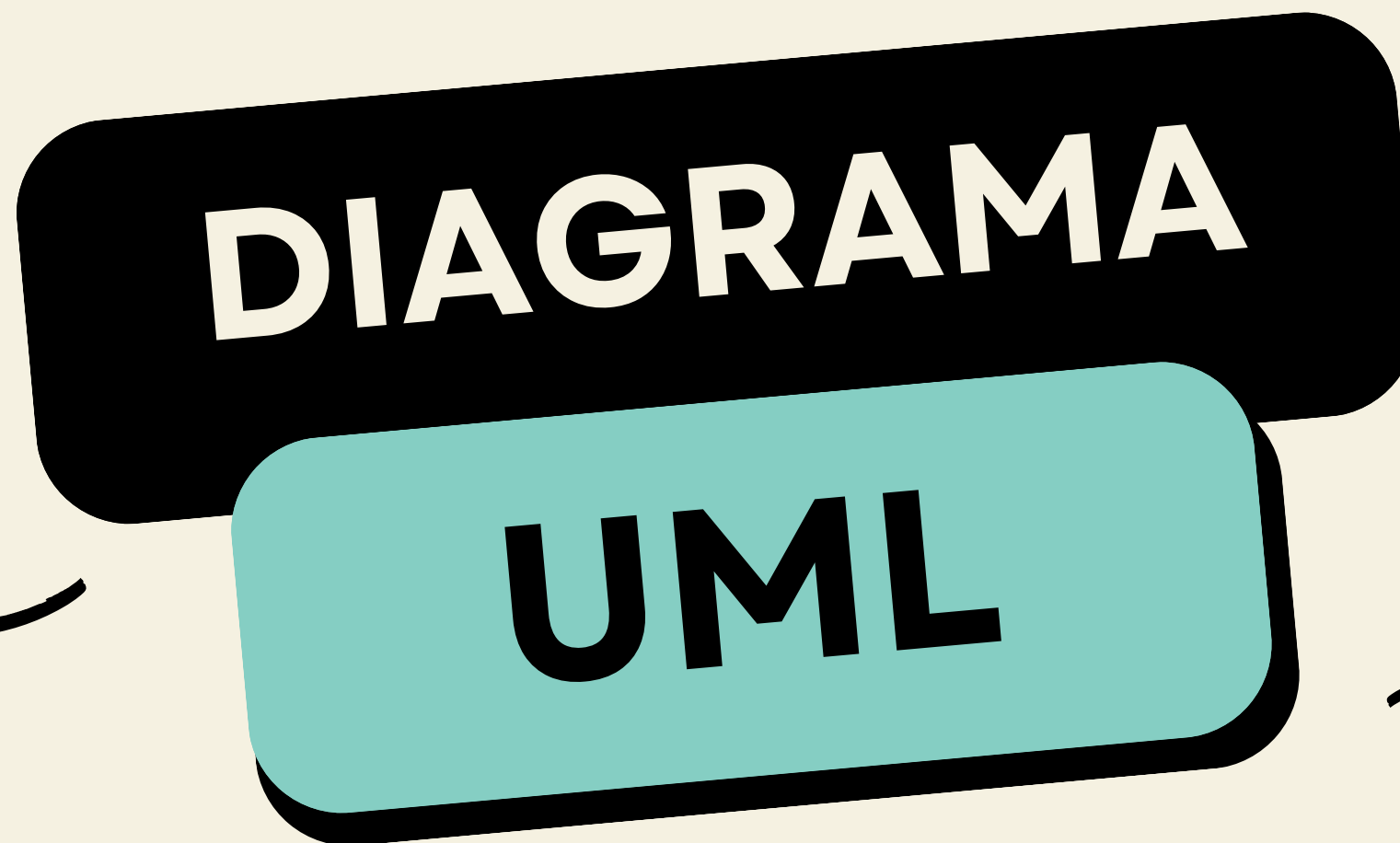
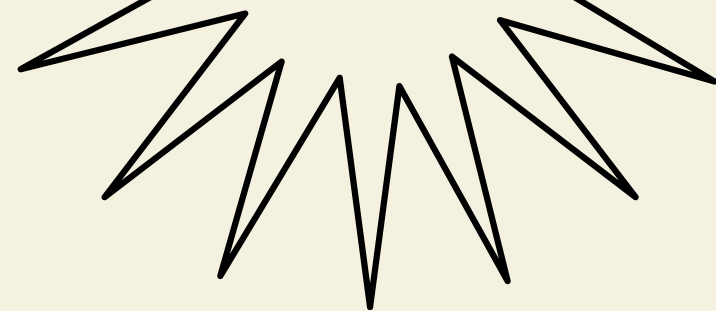
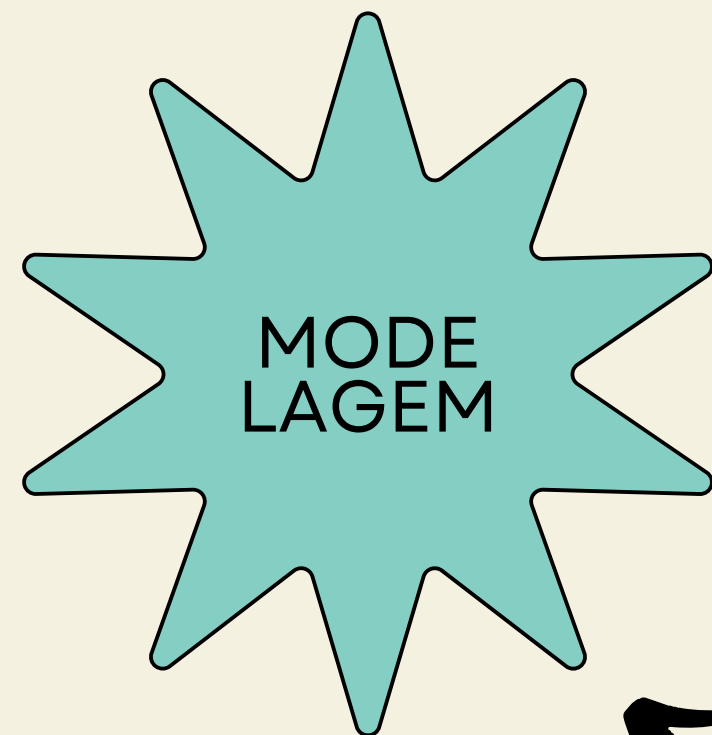
- ◆ Array, Map, Set

E métodos para manipular coleções:

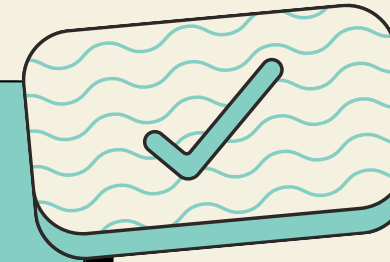
- ◆ .forEach(), .map(), .filter(), .find()

```
class Pessoa {  
  constructor(nome) {  
    this.nome = nome;  
  }  
}  
  
const grupo = [new Pessoa("Ana"), new  
Pessoa("Bia"), new Pessoa("Caio")];  
grupo.forEach(p => console.log(p.nome));  
// Ana, Bia, Caio
```



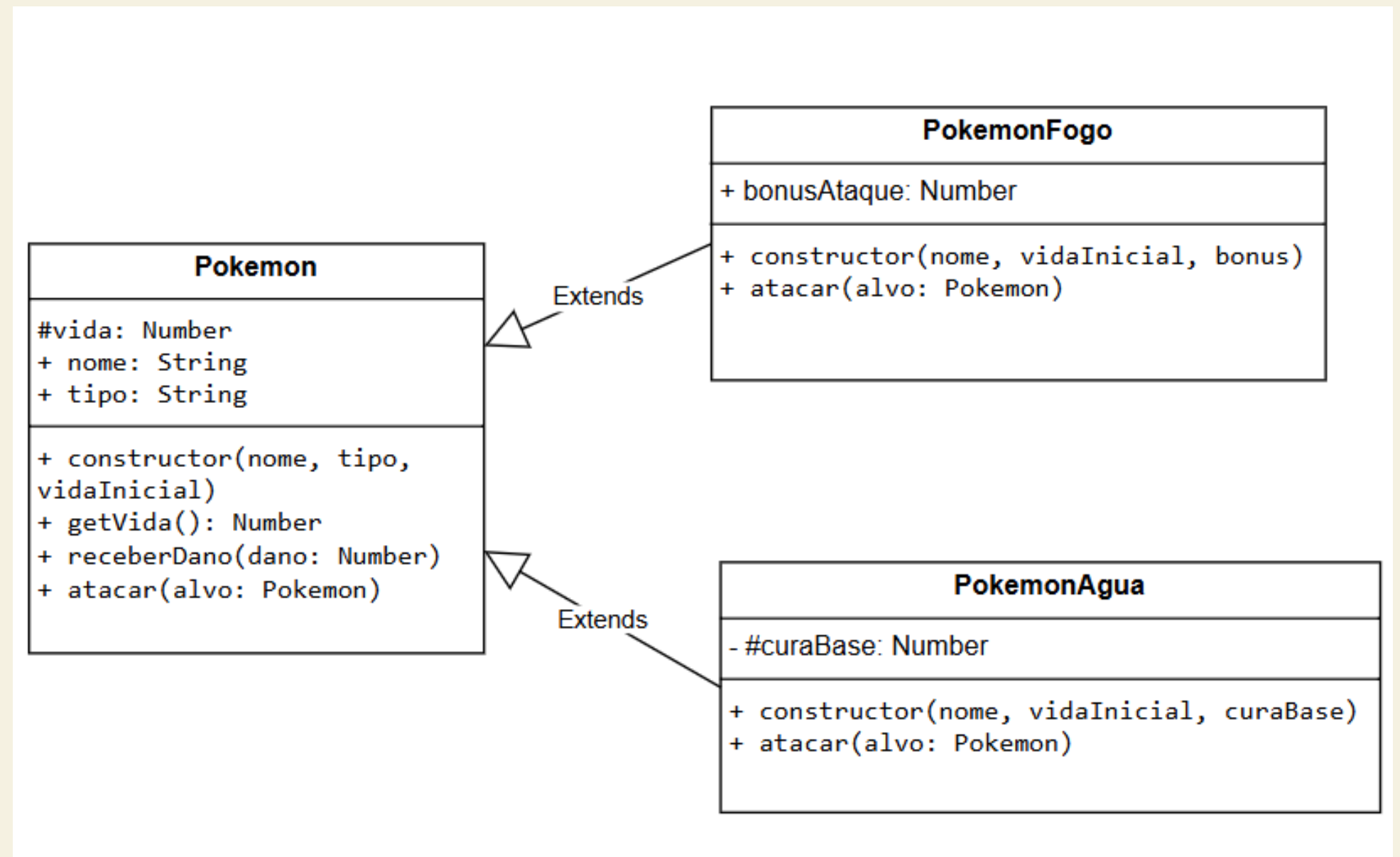


# QUESTÃO 1

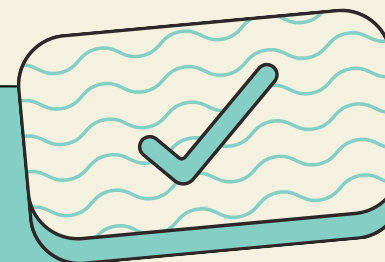


Cenário: Em uma batalha Pokémon, todos os Pokémon herdam características básicas, mas seus ataques são polimórficos. A vida deve ser protegida de alterações externas.

Vamos simular uma batalha Pokémon. A classe Pokémon define as características básicas e o ataque genérico. Precisamos garantir que a vida do Pokémon seja modificada apenas internamente. Crie subclasses para ataques especializados e demonstre que, ao chamar o método atacar do Pokémon, cada um executa sua lógica única."

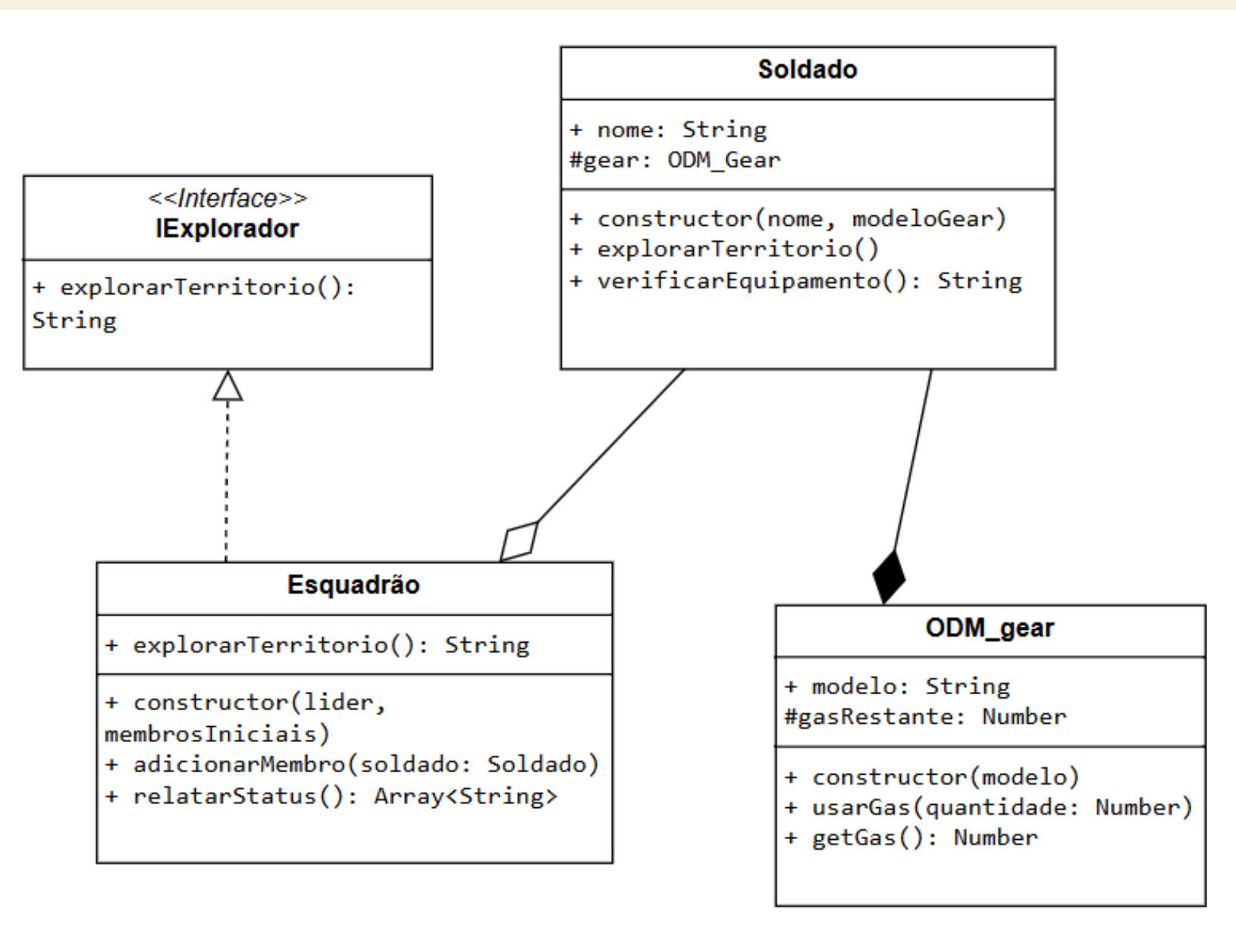


# QUESTÃO 2

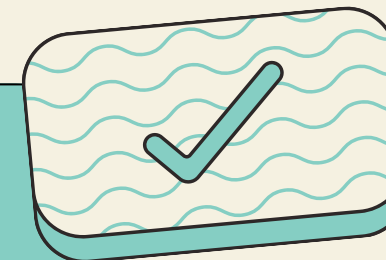


Cenário: Um Esquadrão do Capitão Levi é composto por Soldados, cada um equipado com um ODM\_Gear essencial para a mobilidade.

Modele a estrutura do Esquadrão. O ODM\_Gear é um componente fundamental do Soldado. Além disso, o Esquadrão deve agregar vários Soldados. Precisamos garantir que todo soldado implemente o contrato IExplorador para ser útil. Demonstre chamando a função do ODM\_Gear através do Soldado."



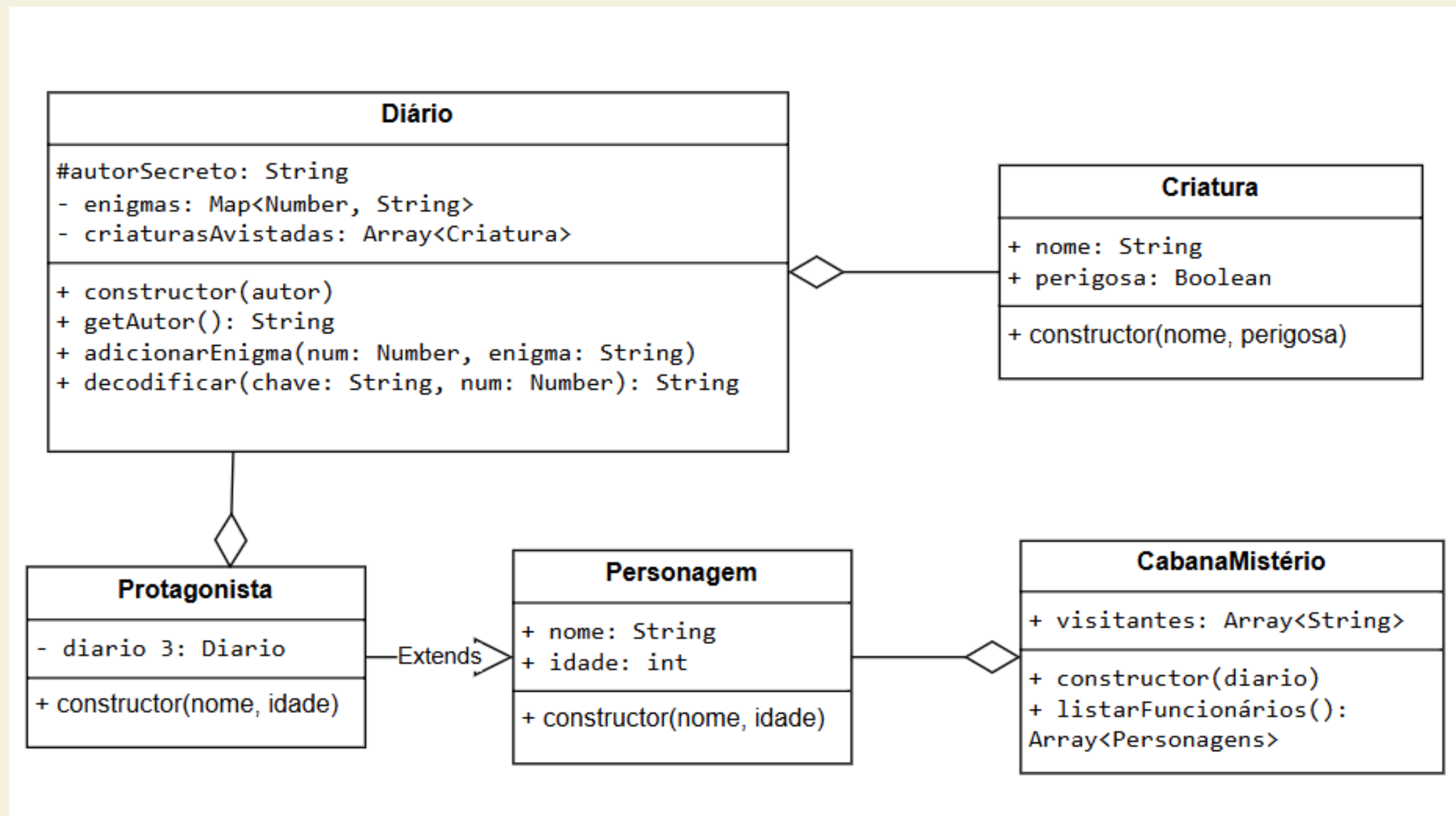
# QUESTÃO 3



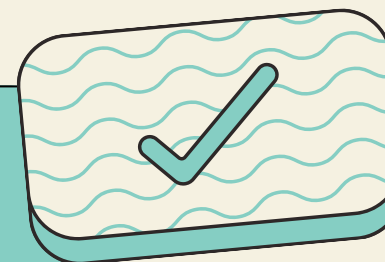
Cenário: Em Gravity Falls, O Protagonista (Dipper) usa o Diário, que agrega dados sobre Criaturas. Enquanto a CabanaMisterio tem Personagens (funcionários) que trabalham nela.

Implemente as classes do diagrama.

- 1.O Diário deve usar um Map para enigmas e o método decodificar(chave, num) deve validar o acesso usando o #autorSecreto e o indice do enigma decodificado.
- 2.Implemente listarFuncionários() na CabanaMisterio. Este método deve acessar a lista de personageans funcioários
- 3.listarFuncionários() deve retornar a coleção de Personagens agregada."

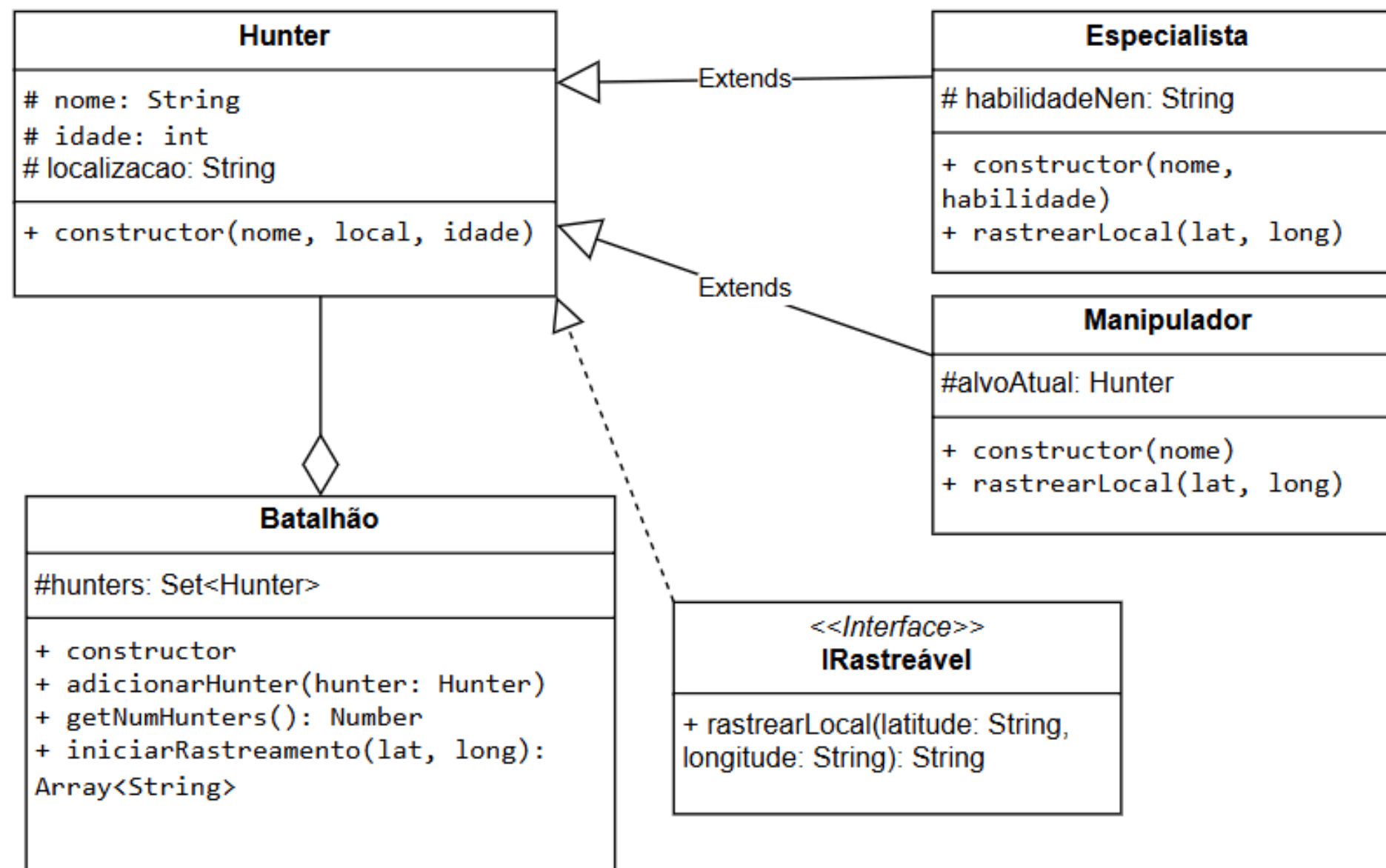


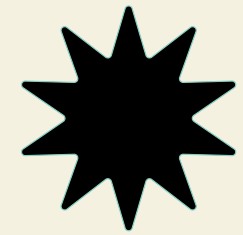
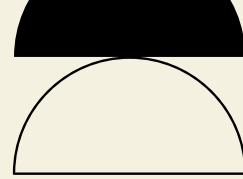
# QUESTÃO 4



Cenário: A Associação Hunter precisa rastrear Hunters que possuem diferentes especialização. Os Hunters Rastreáveis implementam o método rastrearLocal(). O Batalhão não permite Hunters duplicados.

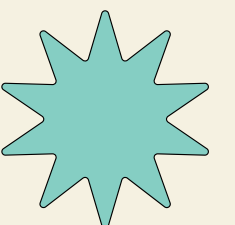
Crie classes para Hunters (Especialista, com sua habilidade e Manipulador, com seu alvo), . A classe Batalhao deve usar um Set para armazenar Hunters, garantindo que não haja duplicatas (identificadas pelo nome). Implemente um método que itere sobre o Set de Hunters e chame o método polimórfico rastrearLocal() em cada um, exibindo sua localização."





# GITHUB

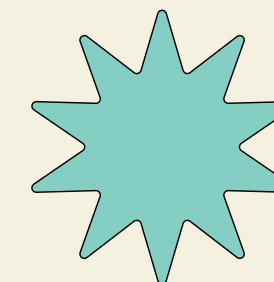
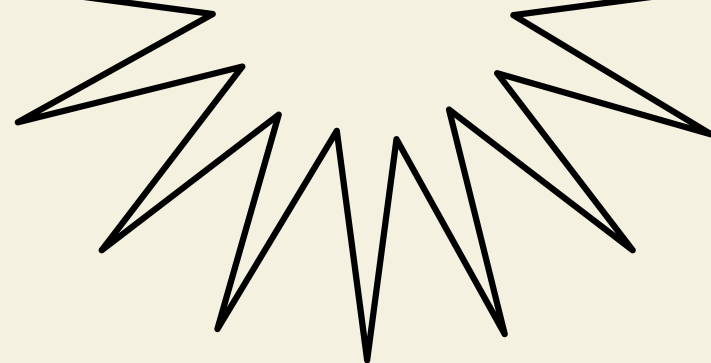
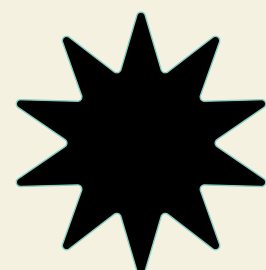
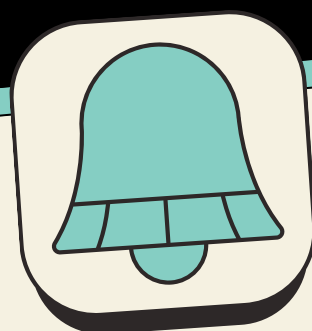
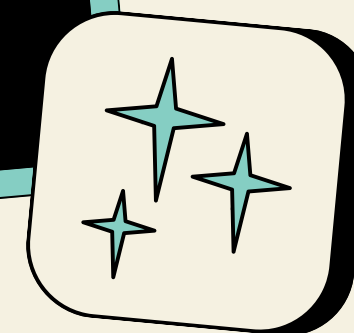
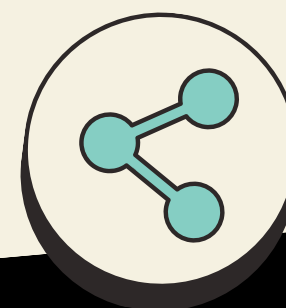
<https://github.com/zSh3rl0cK/S01-monitoria>







**FIM DA AULA**



**LOG IN**

