

*programação*  
**ORIENTADA**  
*a* **OBJETOS**  
**REVISÃO**

BY RAFA

# linguagens de PROGRAMAÇÃO



## O que são?

Conjunto de sintaxes utilizados para criar comandos na comunicação humano-máquina

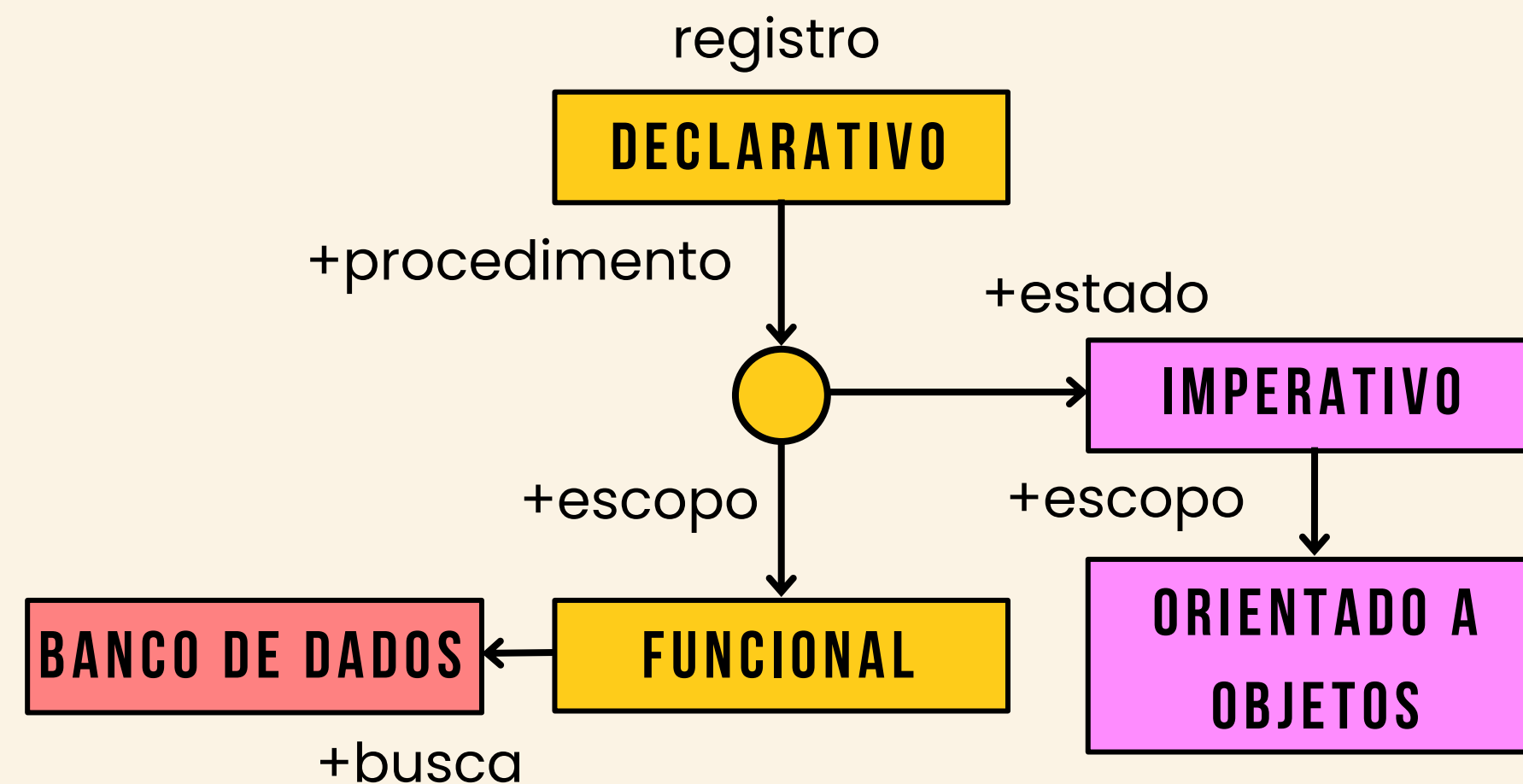
## Como classificar?

São classificadas de BAIXO a ALTO nível sendo que, quanto mais alto, mais próximo à linguagem humana

## Como compreender?

Existem 3 tipos de tradutores: Compiladores, Transpiladores e Interpretadores

# *paradigmas de* **PROGRAMAÇÃO**



# *paradigmas de* **PROGRAMAÇÃO**

## **ORIENTADO A OBJETOS**

Baseado na interação entre **objetos**

Utiliza de **abstrações** para solucionar problemas

Busca poupar código através da **herança** de **classes**

Exemplos: Typescript, C#, Java

*programação*  
**ORIENTADA**  
*a*  
**OBJETOS**

---



**ABSTRAÇÃO**



**ENCAPSULAMENTO**



**HERANÇA**



**POLIMORFISMO**

*programação*

# ORIENTADA *a* OBJETOS

---



The diagram consists of a large light gray rectangle with a black border, divided into four equal quadrants by a horizontal and a vertical line. Each quadrant contains a concept of Object-Oriented Programming in bold black uppercase letters. The top-left quadrant is labeled 'ABSTRAÇÃO' and is circled with a thick pink double-line oval. The top-right quadrant is labeled 'ENCAPSULAMENTO'. The bottom-left quadrant is labeled 'HERANÇA'. The bottom-right quadrant is labeled 'POLIMORFISMO'. Small colored squares are placed at the intersections of the lines: pink at the top-left, teal at the top-right, yellow at the bottom-left, and red at the bottom-right.

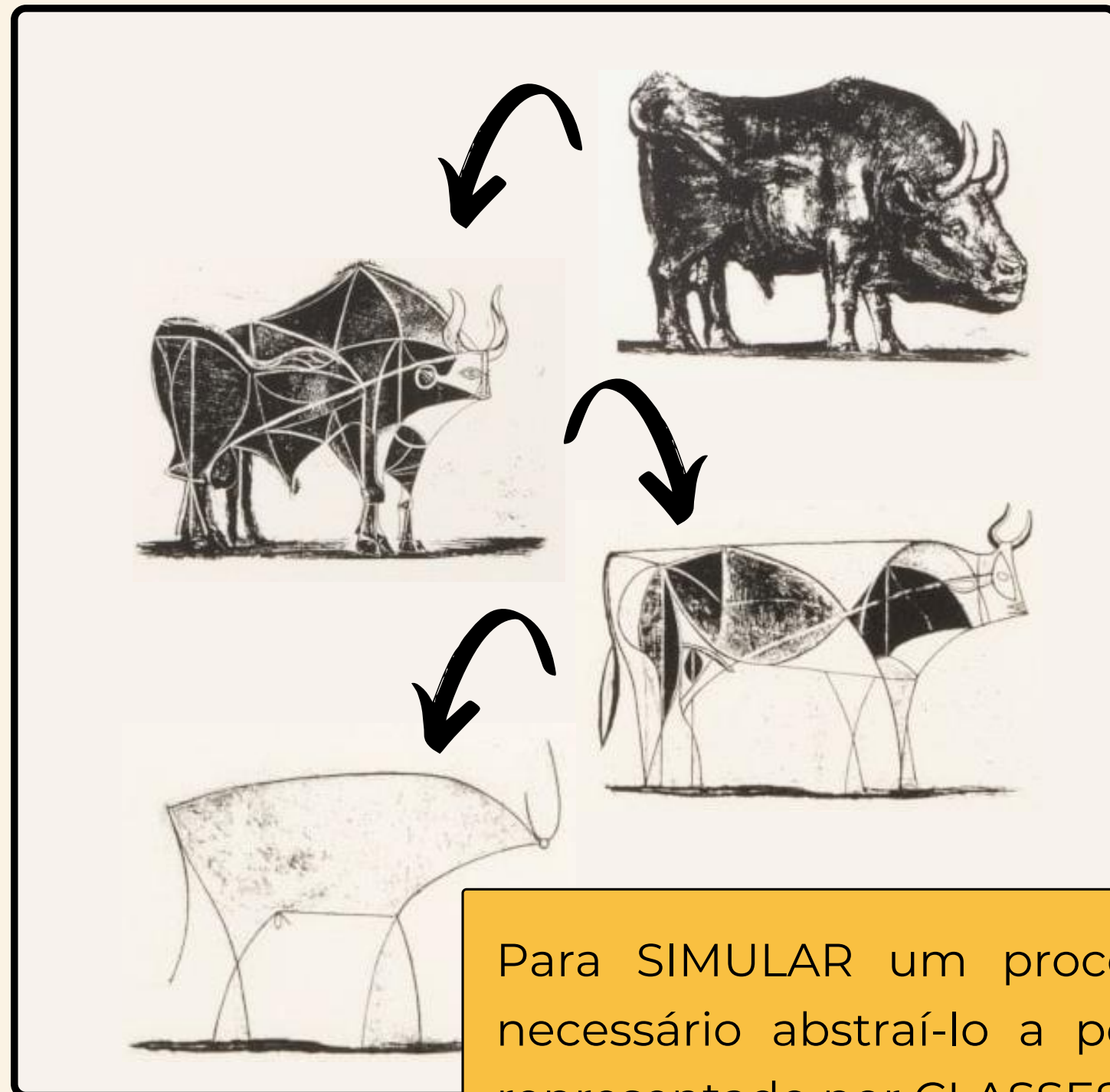
**ABSTRAÇÃO**

**ENCAPSULAMENTO**

**HERANÇA**

**POLIMORFISMO**

# ABSTRAÇÃO



Para SIMULAR um processo real, é necessário abstraí-lo a ponto de ser representado por CLASSES e OBJETOS

## SIMULAÇÃO

A POO tem como objetivo simular PROCESSOS REAIS a partir da ABSTRAÇÃO de elementos

## OBJETOS

São representações dos elementos contidos no processo a ser simulado. São utilizados para INTERAGIREM com outros objetos SIMULANDO o processo real

## CLASSES

São ABSTRAÇÕES aplicadas aos OBJETOS. Um objeto é a forma "CONCRETA" de uma classe



**ABSTRAÇÃO**

**OBJETO**





# ABSTRAÇÃO

# OBJETO



Sandijunior
raça: SRD cor: preto
late(): void corre(): void fazCoco(): Coco

# ABSTRAÇÃO

Sandijunior
raça: SRD cor: preto
late(): void corre(): void fazCoco(): Coco



# OBJETO

Fred
raça: Beagle cor: bege
late(): void corre(): void fazCoco(): Coco

# ABSTRAÇÃO

# OBJETO

Sandijunior
raça: SRD cor: preto
late(): void corre(): void fazCoco(): Coco

Fred
raça: Beagle cor: bege
late(): void corre(): void fazCoco(): Coco

Cachorro
raça: string cor: string
late(): void corre(): void fazCoco(): Coco



# ABSTRAÇÃO

## OBJETO

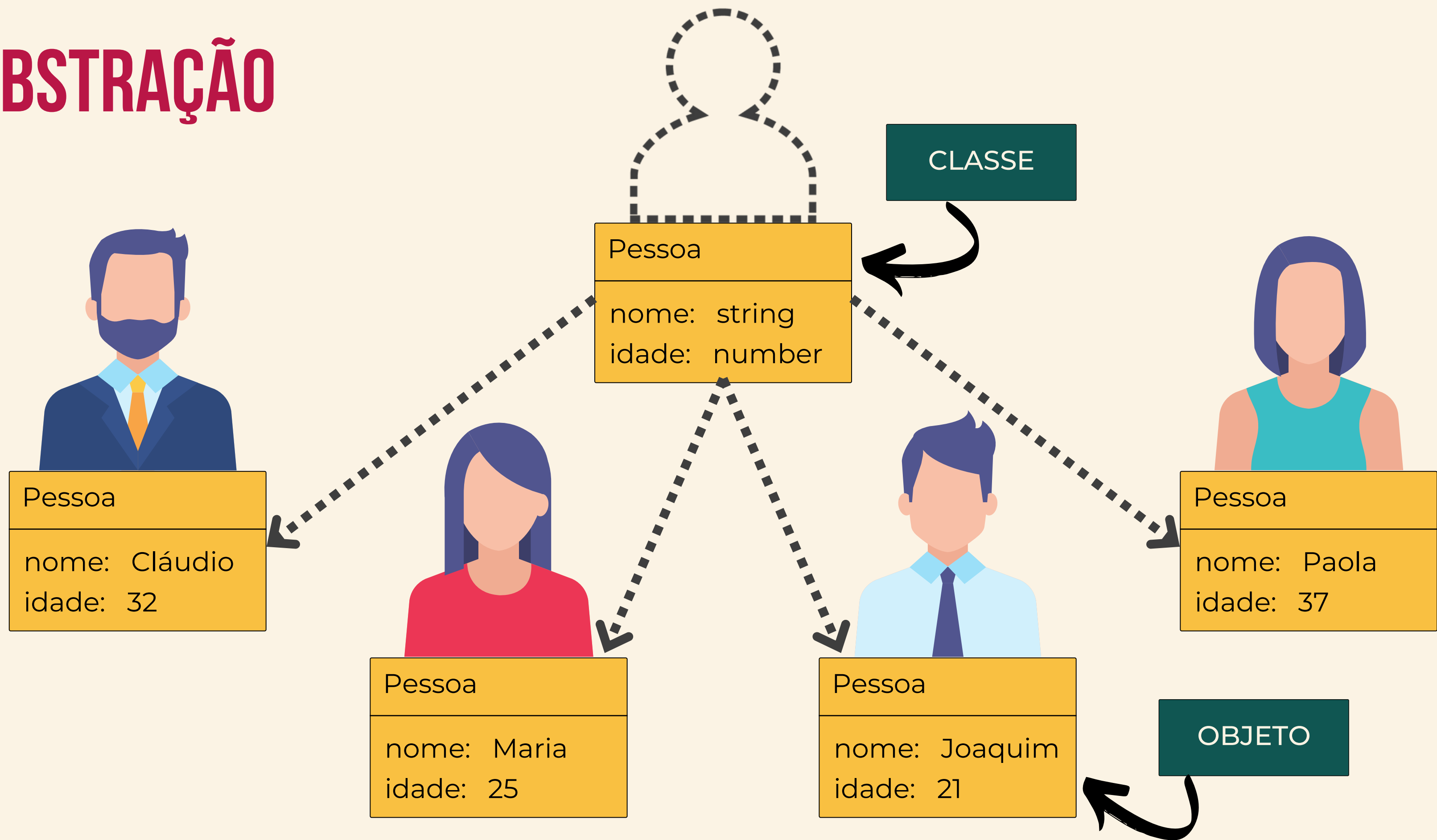
Sandijunior
raça: SRD cor: preto
late(): void corre(): void fazCoco(): Coco

Fred
raça: Beagle cor: bege
late(): void corre(): void fazCoco(): Coco

## CLASSE

Cachorro
raça: string cor: string
late(): void corre(): void fazCoco(): Coco

# ABSTRAÇÃO



# ABSTRAÇÃO

# CONSTRUTOR

Utilizado para **criar** uma instância de uma **classe**. Tem como objetivo, **inicializar** um objeto populando os atributos da classe.

Caso **não declarado**, é considerada com um construtor **vazio**, ou seja, **sem argumentos**, inicializando um objeto cujos atributos serão todos ***undefined***

# THIS

Uma referência à **instância** corrente, ou seja, ao **objeto** em questão. Usado para referenciar os **atributos** e **métodos** daquela instância

```
class Autor{
  nome: string;
  cpf: string;

  constructor(nome: string,
              cpf: string) {
    this.nome = nome;
    this.cpf= cpf;
  }
}
```



*programação*

# ORIENTADA *a* OBJETOS

---



A diagram illustrating the four principles of Object-Oriented Programming (OOP). It consists of a large rectangle divided into four quadrants by a horizontal and a vertical line. Each quadrant contains a principle name in bold, uppercase letters. Small colored squares are placed at the intersections of the lines: a pink square at the top-left, a yellow square at the bottom-left, a teal square at the top-right, and a pink square at the bottom-right. The word 'ENCAPSULAMENTO' in the top-right quadrant is circled with a thick, hand-drawn pink oval.

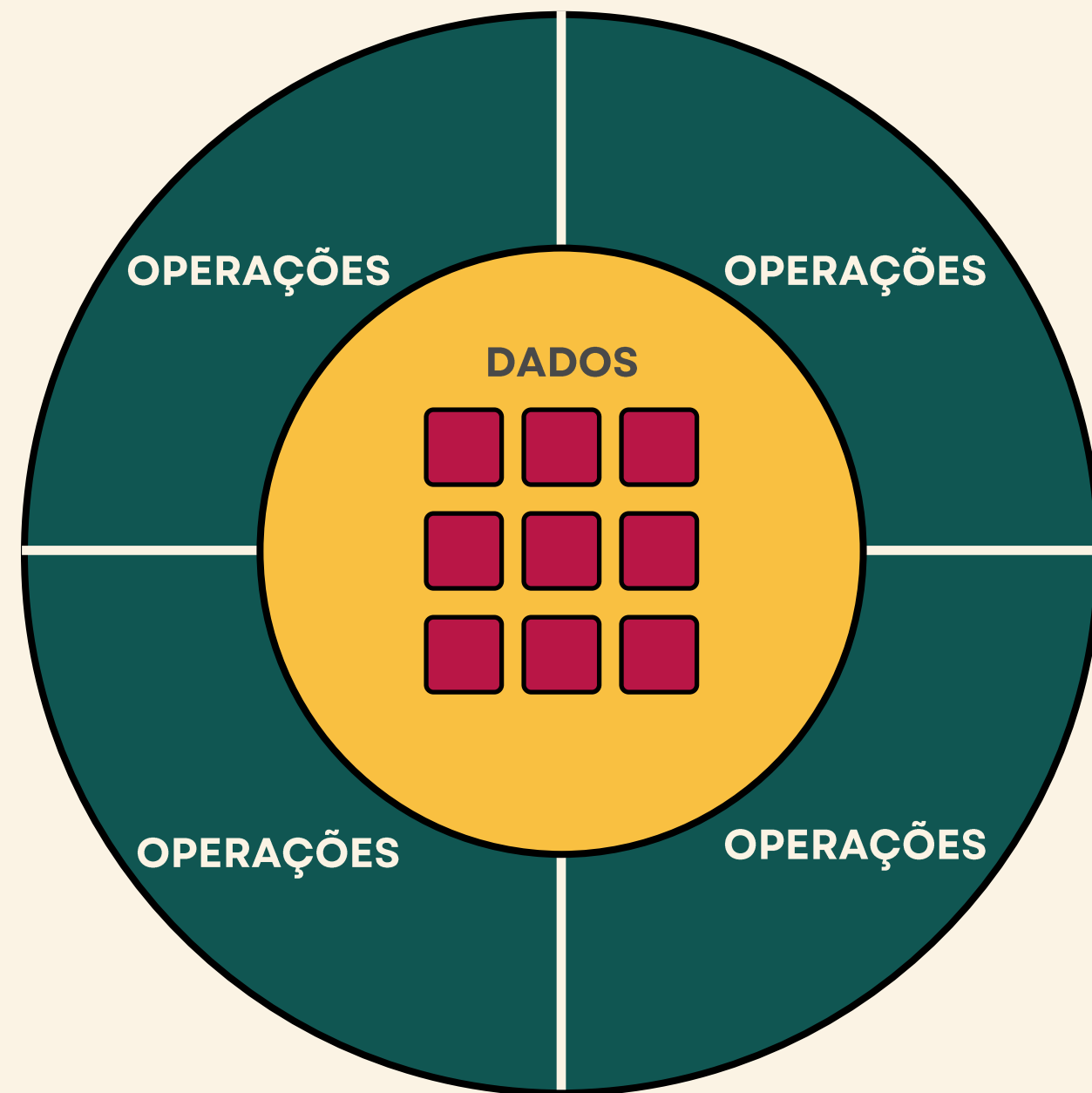
**ABSTRAÇÃO**

**ENCAPSULAMENTO**

**HERANÇA**

**POLIMORFISMO**










# ENCAPSULAMENTO



Biblioteca
nome: string livros: Array<Livro> emprestimos: Array<Emprestimo>
getLivros(): Array<Livro> addLivro(livro: Livro): void empresta(livro: Livro, leitor: Leitor): void

# ENCAPSULAMENTO

## MODIFICADORES DE ACESSO

MODIFICADOR	ACESSO INTERNO	ACESSO FILHAS	ACESSO EXTERNO
PRIVATE			
PROTECTED			
PUBLIC			

# ENCAPSULAMENTO GET/SET

Utilizado para **expor atributos protegidos** por modificadores de acesso. É importante que seja **usado com cautela**. Não há sentido em um modificador de acesso se implementados ambos get/set públicos.

**ATENÇÃO:** getters para **atributos** do tipo **boolean** possuem o **prefixo "is"** ao invés de "get".

```
class Thing {  
    protected name: string;  
    protected life: number;  
    protected destroyed: boolean;  
  
    constructor(name: string,  
                life: number) {  
        this.life = life;  
        this.destroyed = false;  
        this.name = name;  
    }  
  
    destroy(): void {  
        this.life = 0;  
        this.destroyed = true;  
    }  
  
    getLife(): number {  
        return this.life;  
    }  
  
    takeDamage(damage: number): void {  
        this.life -= damage;  
        if (this.life <= 0) {  
            this.destroy();  
        }  
    }  
  
    isDestroyed(): boolean {  
        return this.destroyed;  
    }  
}
```

*programação*  
**ORIENTADA  
a  
OBJETOS**

---



A diagram showing four OOP concepts arranged in a 2x2 grid. The top row contains 'ABSTRAÇÃO' and 'ENCAPSULAMENTO'. The bottom row contains 'HERANÇA' and 'POLIMORFISMO'. The 'HERANÇA' cell is circled in red. Small colored squares (pink, teal, yellow, pink) are placed at the intersections of the grid lines.

**ABSTRAÇÃO**

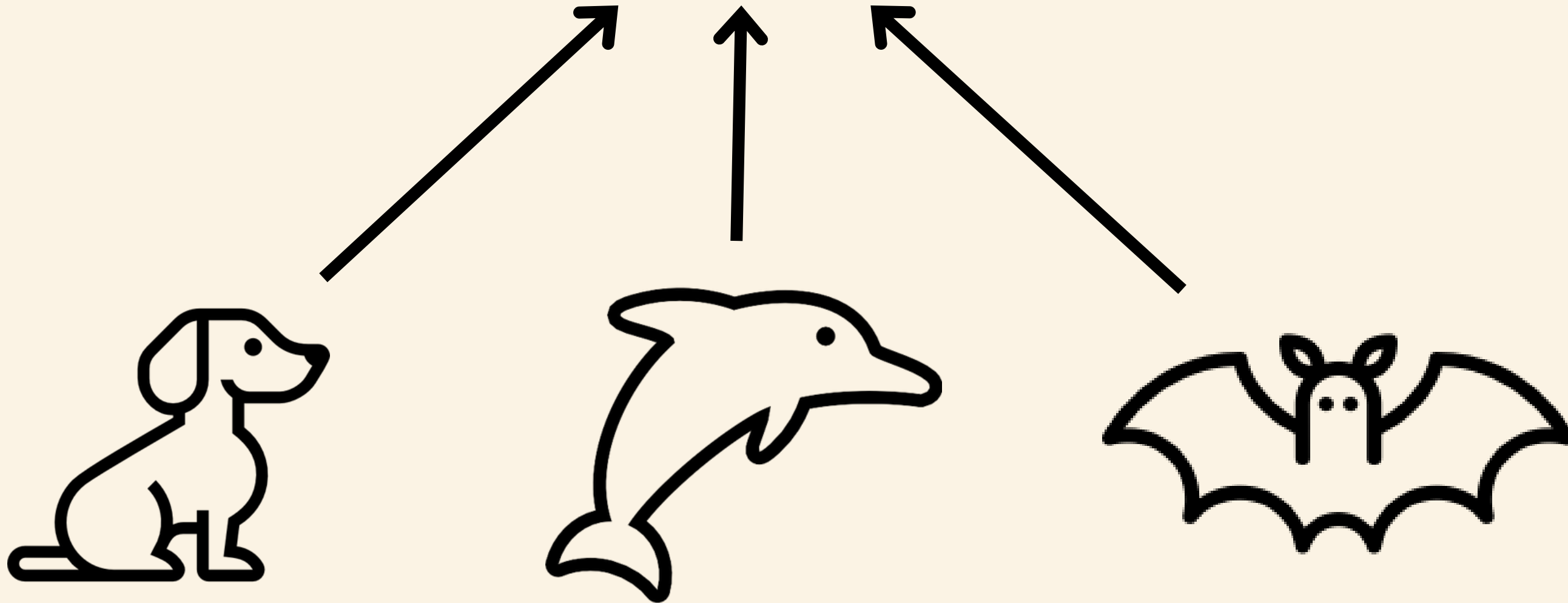
**ENCAPSULAMENTO**

**HERANÇA**

**POLIMORFISMO**

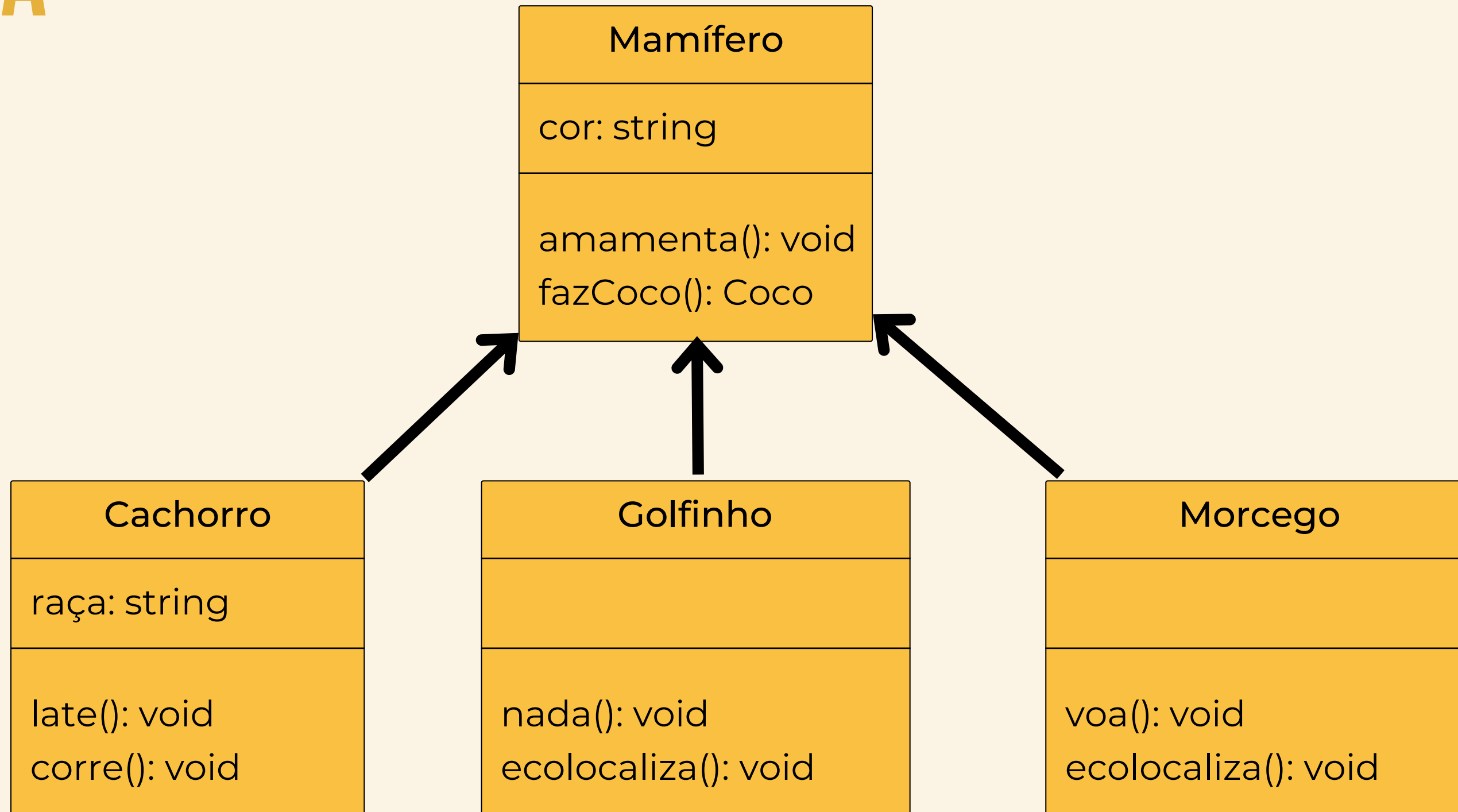
HERANÇA

**MAMÍFERO**

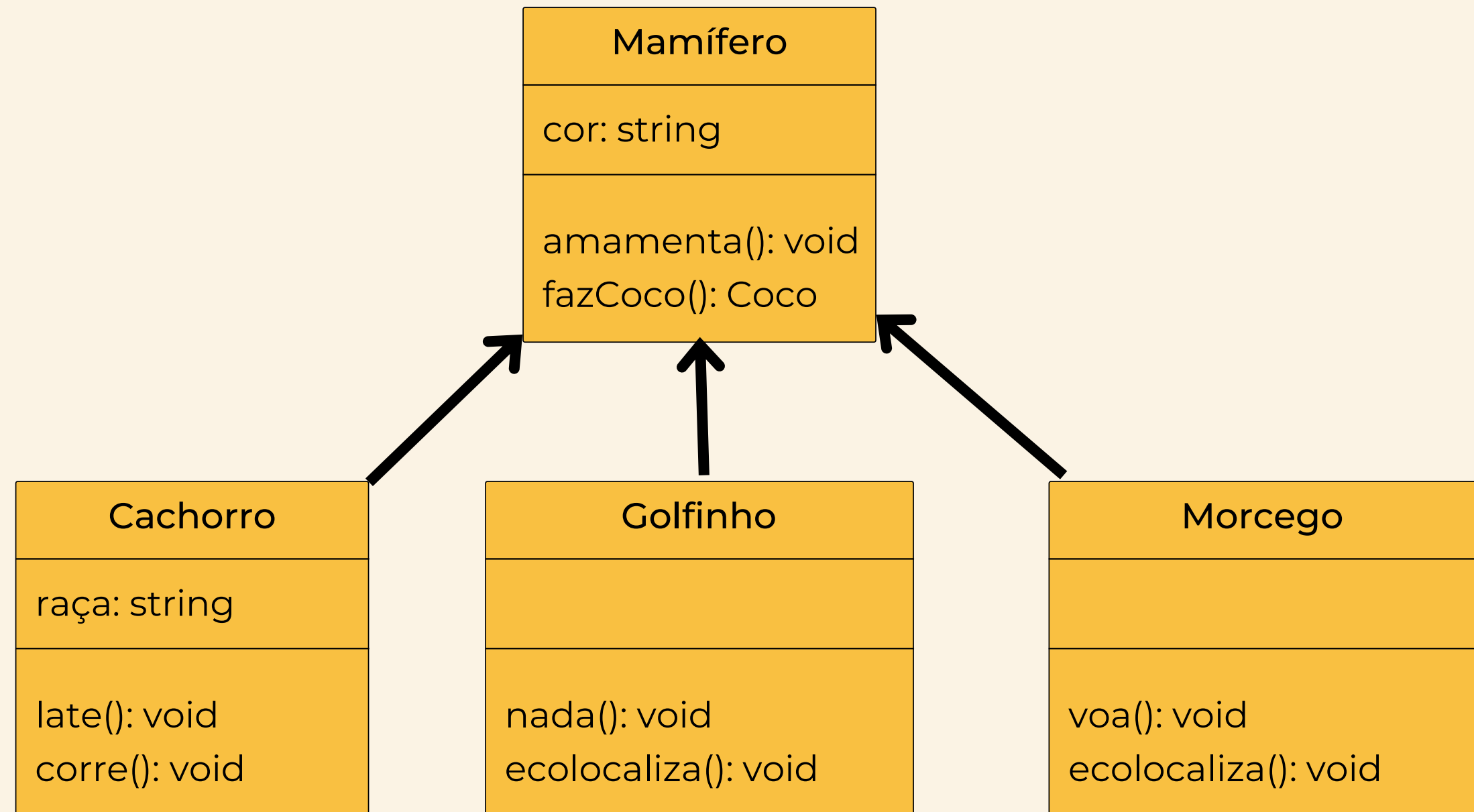




# HERANÇA



# HERANÇA



Uma classe, ao ter classes que a herdaram é dita como classe **PAI/MÃE** ou **SUPER CLASSE** enquanto as classes que a herdaram são ditas como classes **FILHAS**



Permite o **APROVEITAMENTO** de propriedades de outras classes

*programação*

# ORIENTADA *a* OBJETOS

---



A diagram showing four OOP principles arranged in a 2x2 grid. The top row contains 'ABSTRAÇÃO' and 'ENCAPSULAMENTO', and the bottom row contains 'HERANÇA' and 'POLIMORFISMO'. Each principle is centered in its respective cell. Above each principle is a small colored square: pink for 'ABSTRAÇÃO', teal for 'ENCAPSULAMENTO', yellow for 'HERANÇA', and pink for 'POLIMORFISMO'. The 'POLIMORFISMO' text is circled with a thick pink oval.

**ABSTRAÇÃO**

**ENCAPSULAMENTO**

**HERANÇA**

**POLIMORFISMO**

# POLIMORFISMO

```
abstract class Mamifero {
    cor: string;

    constructor(cor: string) {
        this.cor = cor;
    }

    amamenta(): void {
        // implementação do método
    }

    fazCoco(): Coco {
        // implementação do método
    }

    abstract comunica(): void;
    abstract desloca(): void;
}
```

```
class Cachorro extends Mamifero {
    raca: string;

    constructor(cor: string, raca: string) {
        super(cor);
        this.raca = raca;
    }

    private late(): void {
        // implementação do método
    }

    private corre(): void {
        // implementação do método
    }

    comunica(): void {
        this.late();
    }

    desloca(): void {
        this.corre();
    }
}
```

```
class Morcego extends Mamifero {
    constructor(cor: string) {
        super(cor);
    }

    private ecolocaliza(): void {
        // implementação do método
    }

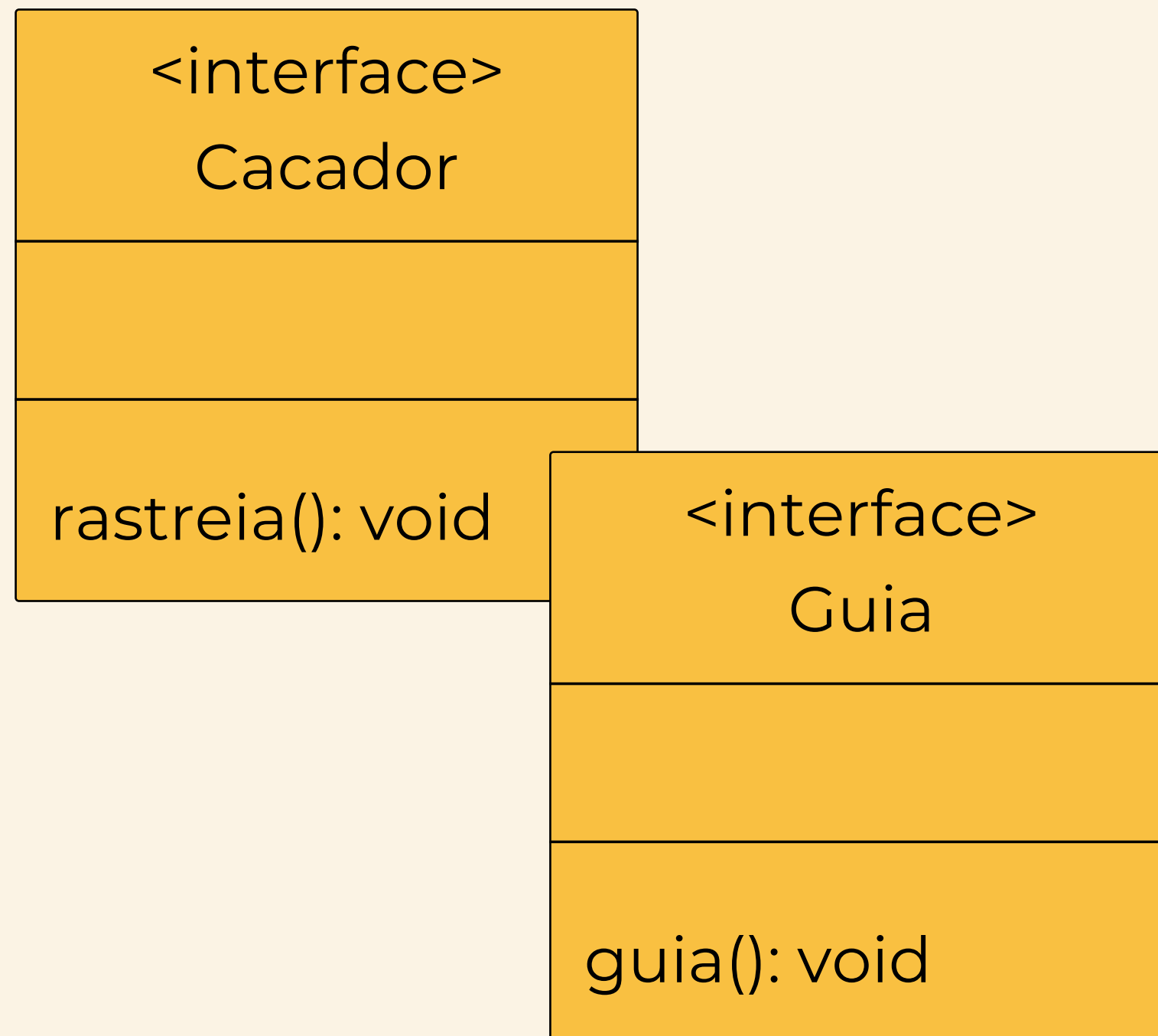
    private voa(): void {
        // implementação do método
    }

    comunica(): void {
        this.ecolocaliza();
    }

    desloca(): void {
        this.voa();
    }
}
```

# POLIMORFISMO

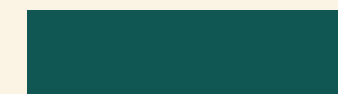
## INTERFACES



INTERFACES podem ser vistas como classes **ABSTRATAS** mais **LIMITADAS**. Também não podem ser instanciadas diretamente.



**NÃO** permite a **IMPLEMENTAÇÃO** de métodos. **TODOS** os métodos são **ABSTRATOS**.



**NÃO** permite a definição de atributos. **APENAS** constantes estáticas são permitidas.

# POLIMORFISMO EQUALS

Método utilizado para **comparar** dois **objetos**.

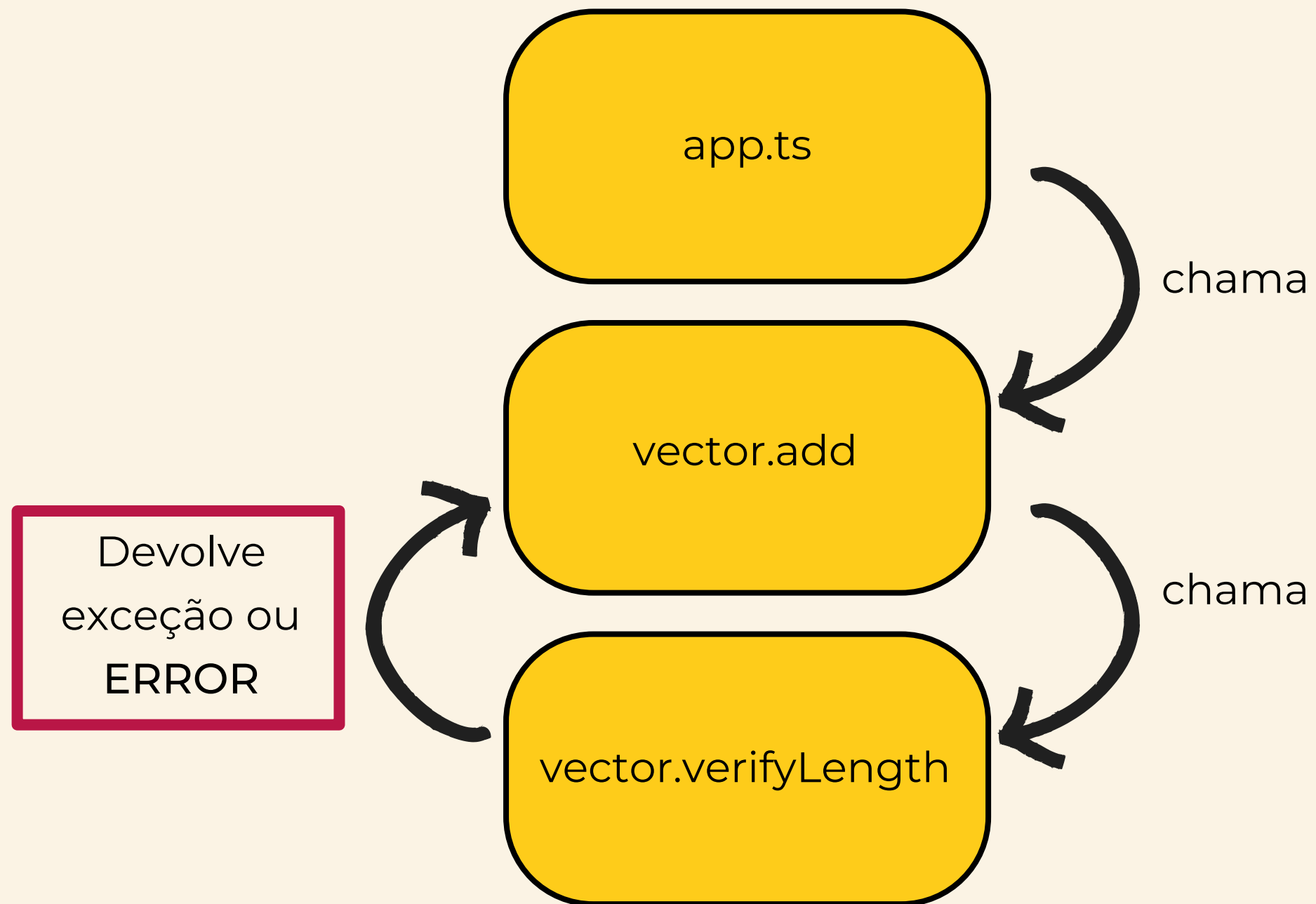
Ao utilizar objetos, deve-se **evitar** a comparação por meio dos operadores "==" e "===". Apesar de funcionar, estes operadores **não comparam o conteúdo** dos objetos e sim o **enderço de memória** ocupado por ele.

Implementando um método **equals** é possível seguir a **regra** cabível ao contexto da aplicação.

```
class Thing {  
    protected name: string;  
    protected life: number;  
    protected destroyed: boolean;  
  
    constructor(name: string,  
                life: number) {  
        this.life = life;  
        this.destroyed = false;  
        this.name = name;  
    }  
  
    //métodos  
  
    getName(): string {  
        return this.name;  
    }  
  
    equals(thing: Thing): boolean {  
        return this.name === thing.getName()  
    }  
}
```



# EXCEÇÕES



## O que são?

Evento que ocorrem durante a execução de programas que fogem do fluxo esperado de suas instruções.

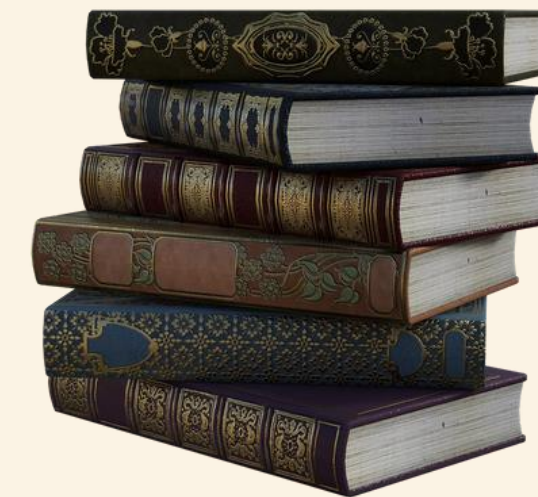
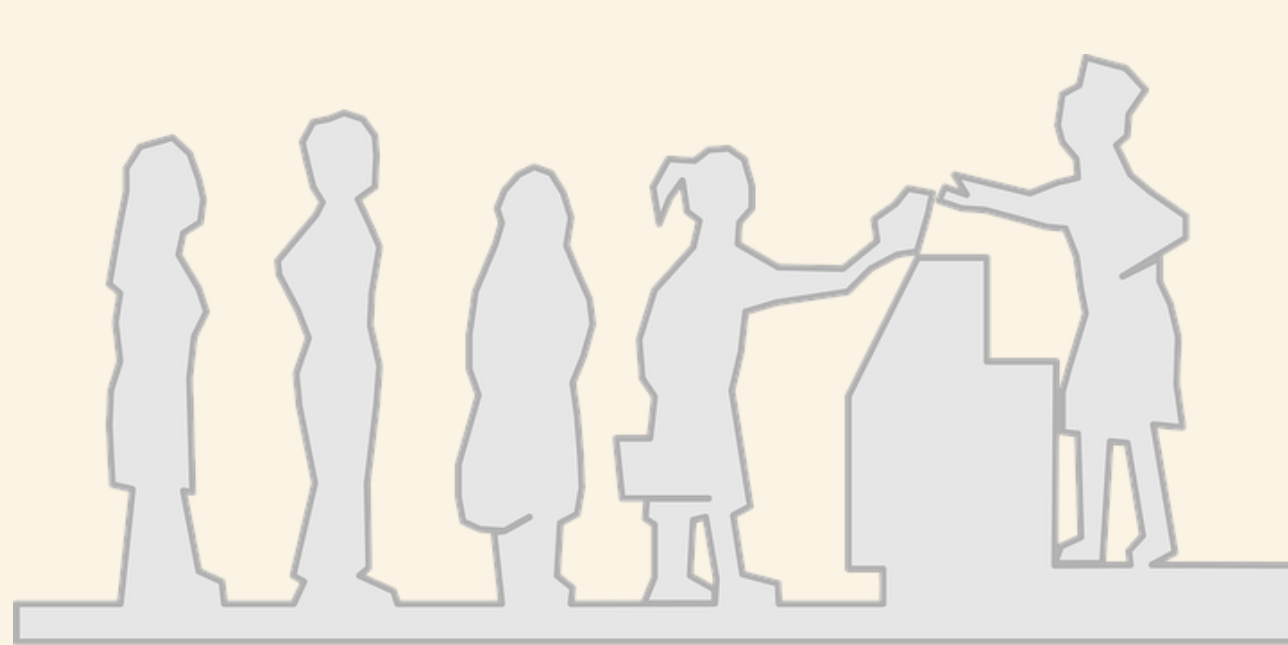
## Pra que servem?

Ao disparar uma exceção, o programa permite que outras partes do próprio programa ou até mesmo outros programas compreendam a falha no fluxo esperado

## Como tratar?

São disparadas a partir do uso do termo "throw" e podem ser identificadas e devidamente tratadas usando "try...catch"

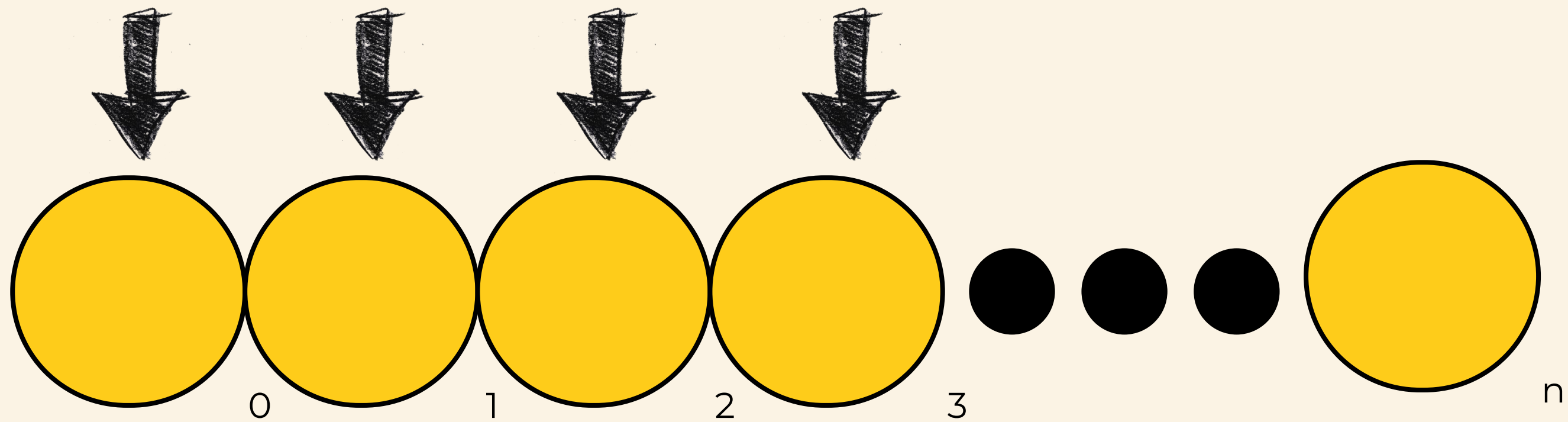
# ESTRUTURAS DE DADOS



## O que são?

Tipos não-primitivos de organização de dados para atender aos diferentes requisitos de processamento

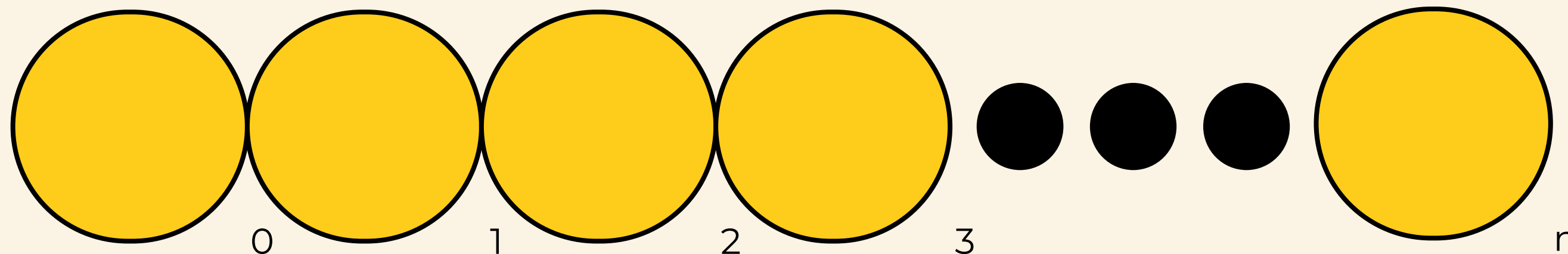
# ARRAY



Posições bem definidas  
Tamanho limitado (nem sempre)  
Itens podem ser adicionados em qualquer  
posição

# LISTA

Array de "luxo"



Posições bem definidas

Tamanho se adapta à posição ocupada de maior índice

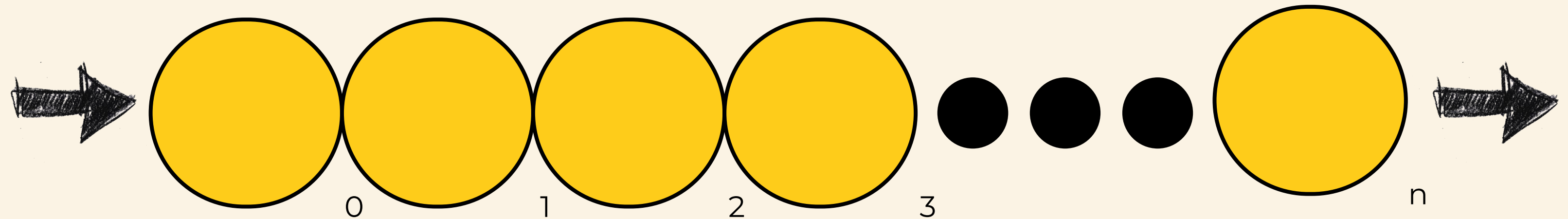
Itens podem ser adicionados em qualquer posição

Não possui ordem de saída definida

```
export interface IList {  
  add(item: string): void;  
  remove(index: number): void;  
  get(index: number): void;  
  set(index: number, item: string): void;  
  contains(item: string): boolean;  
  size(): number;  
  isEmpty(): boolean;  
}
```

# FILA

Primeiro a Chegar Primeiro a Sair (PCPS)  
First In, First Out (FIFO)

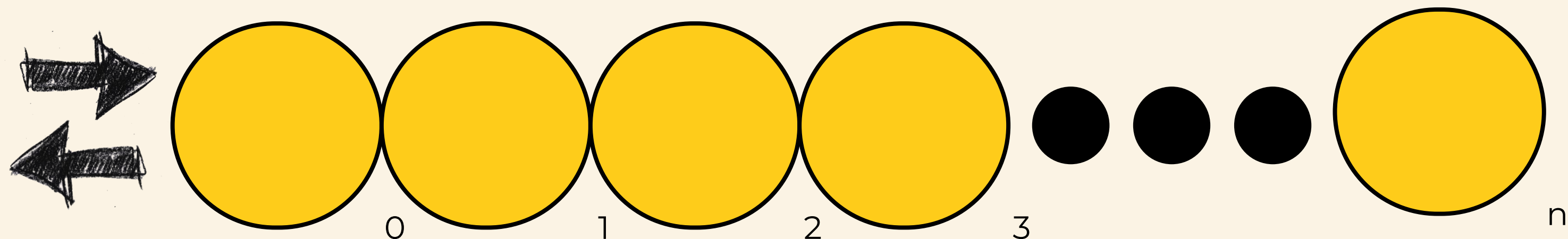


Posições bem definidas  
Tamanho se adapta à posição ocupada de maior índice  
Itens podem ser adicionados apenas no menor índice (zero)  
Saída deve ser feita pelo maior índice ocupado

```
export interface IQueue {  
  enqueue(item: string): void;  
  dequeue(): string;  
  size(): number;  
  isFull(): boolean;  
}
```

# PILHA

Primeiro a Chegar Último a Sair (PCUS)  
First In, Last Out (FILO)



Posições bem definidas

Tamanho se adapta à posição ocupada de maior índice

Itens podem ser adicionados apenas no menor índice (zero)

Saída deve ser feita pelo menor índice ocupado

```
export interface IStack {  
  push(item: string): void;  
  pop(): string;  
  size(): number;  
  isFull(): boolean;  
}
```





**OBRIGADO!**