

introdução à
ORIENTAÇÃO
a **OBJETOS #6**

BY RAFA

introdução à **ORIENTAÇÃO** *a* **OBJETOS** #6

REVISÃO

MAH QUÊ ISSO? FOR

UML

PROJETO INTEGRADOR

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



ABSTRAÇÃO



ENCAPSULAMENTO



HERANÇA



POLIMORFISMO

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

CLASSE

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










ENCAPSULAMENTO

MODIFICADORES DE ACESSO



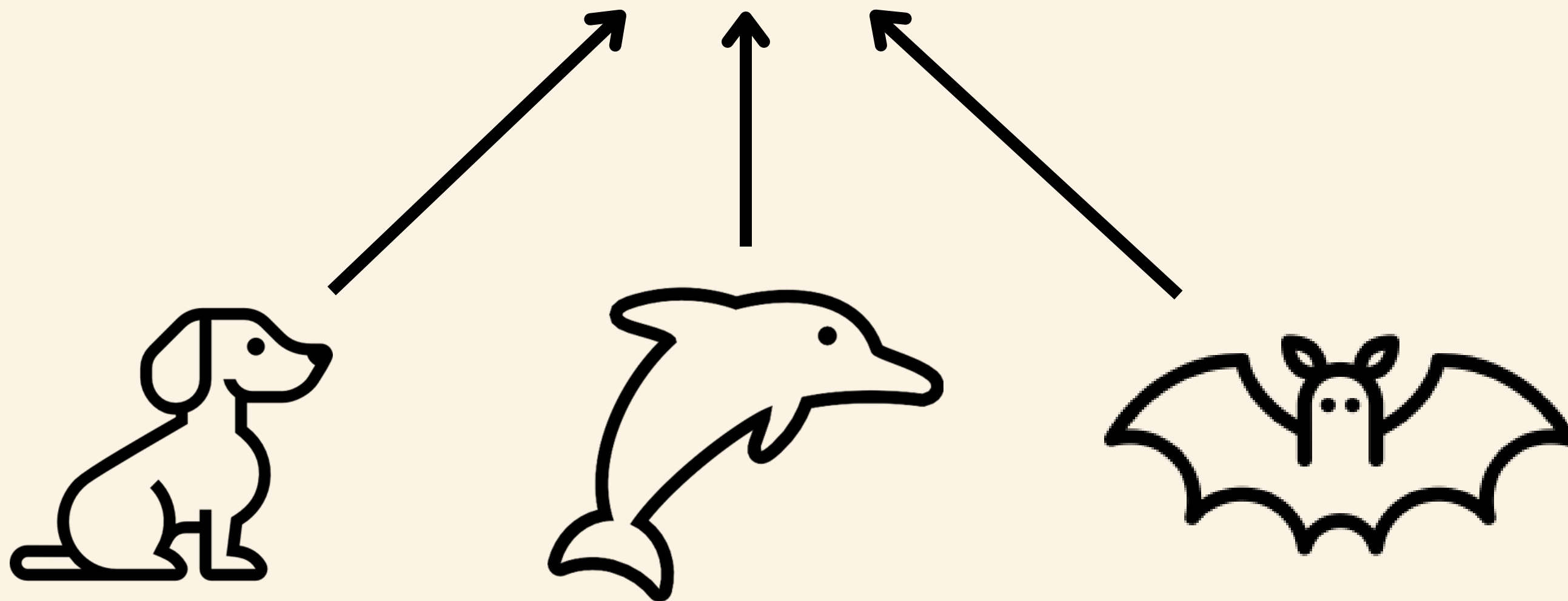
ENCAPSULAMENTO

MODIFICADORES DE ACESSO

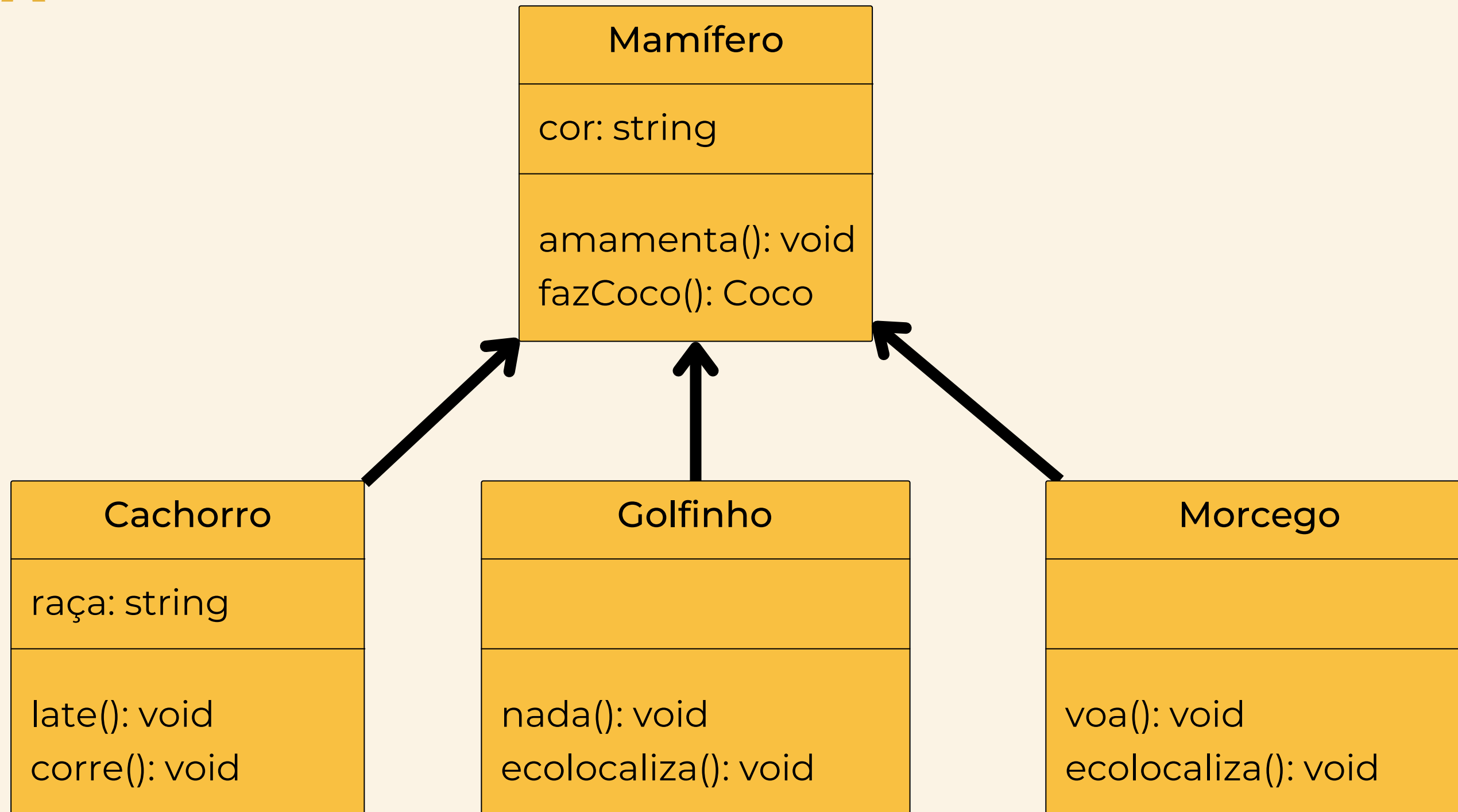
MODIFICADOR	ACESSO INTERNO	ACESSO FILHAS	ACESSO EXTERNO
PRIVATE			
PROTECTED			
PUBLIC			

HERANÇA

MAMÍFERO



HERANÇA



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();
    }
}
```

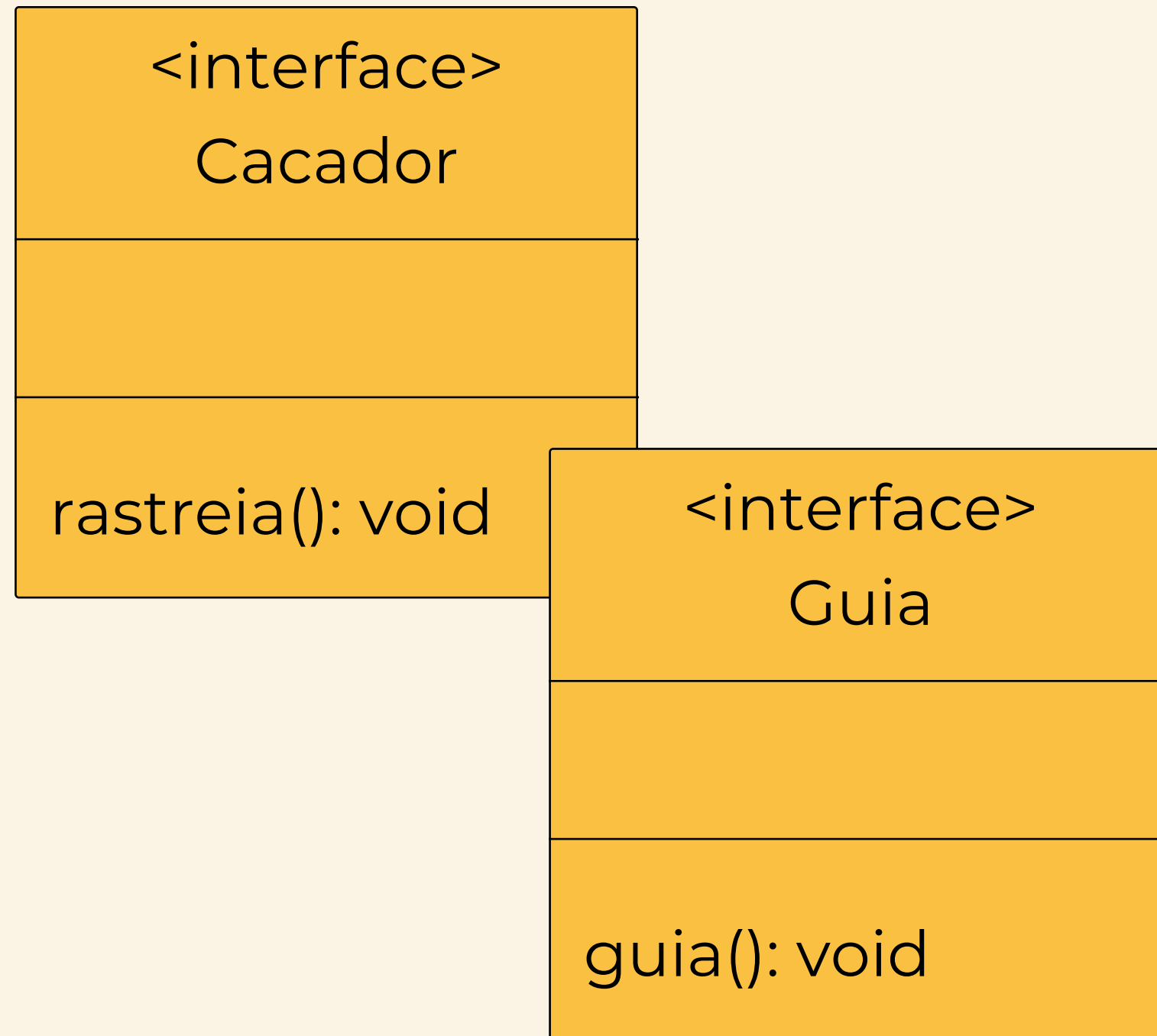
POLIMORFISMO

```
interface Guia {  
    guia(): void;  
}
```

```
interface Cacador {  
    rastreia(): void;  
}
```

```
class Cachorro extends Mamifero implements Guia, Cacador {  
    raca: string;  
  
    constructor(cor: string, raca: string) {  
        super(cor)  
        this.raca = raca;  
    }  
  
    // métodos  
  
    rastreia(): void {  
        // implementação do método  
    }  
  
    guia(): void {  
        // implementação do método  
    }  
}
```

POLIMORFISMO



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.

THIS

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

```
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() {  
    this.life = 0;  
    this.destroyed = true;  
  }  
}
```

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;
    }
}
```

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()  
    }  
}
```

CONSTANTES

Tal qual o tipo **const** do **javascript**, estes valores **não** podem ser **alterados** em tempo de **execução**.

readonly utilizado para definir um **atributo** cujo valor pode ser alterado **apenas** no **construtor**.

static operador utilizado para definir um **atributo** que **não** será **acessível** pelas **instâncias**. Será **visível** apenas a nível da **classe**.

```
class Thing {
  static readonly defaultLife = 1000;

  protected name: string;
  protected life: number;
  protected destroyed: boolean;


  constructor(name: string,
              life: number = Thing.defaultLife) {
    this.life = life;
    this.destroyed = false;
    this.name = name;
  }

  //métodos
}
```


INFORMAÇÃO

EXTRA

```
for
```

	for	for
<input type="checkbox"/>	for	For Loop
<input type="checkbox"/>	forawaitof	For-Await-Of Loop
<input type="checkbox"/>	foreach =>	For-Each Loop using =>
<input type="checkbox"/>	forin	For-In Loop
<input type="checkbox"/>	forof	For-Of Loop

INFORMAÇÃO EXTRA

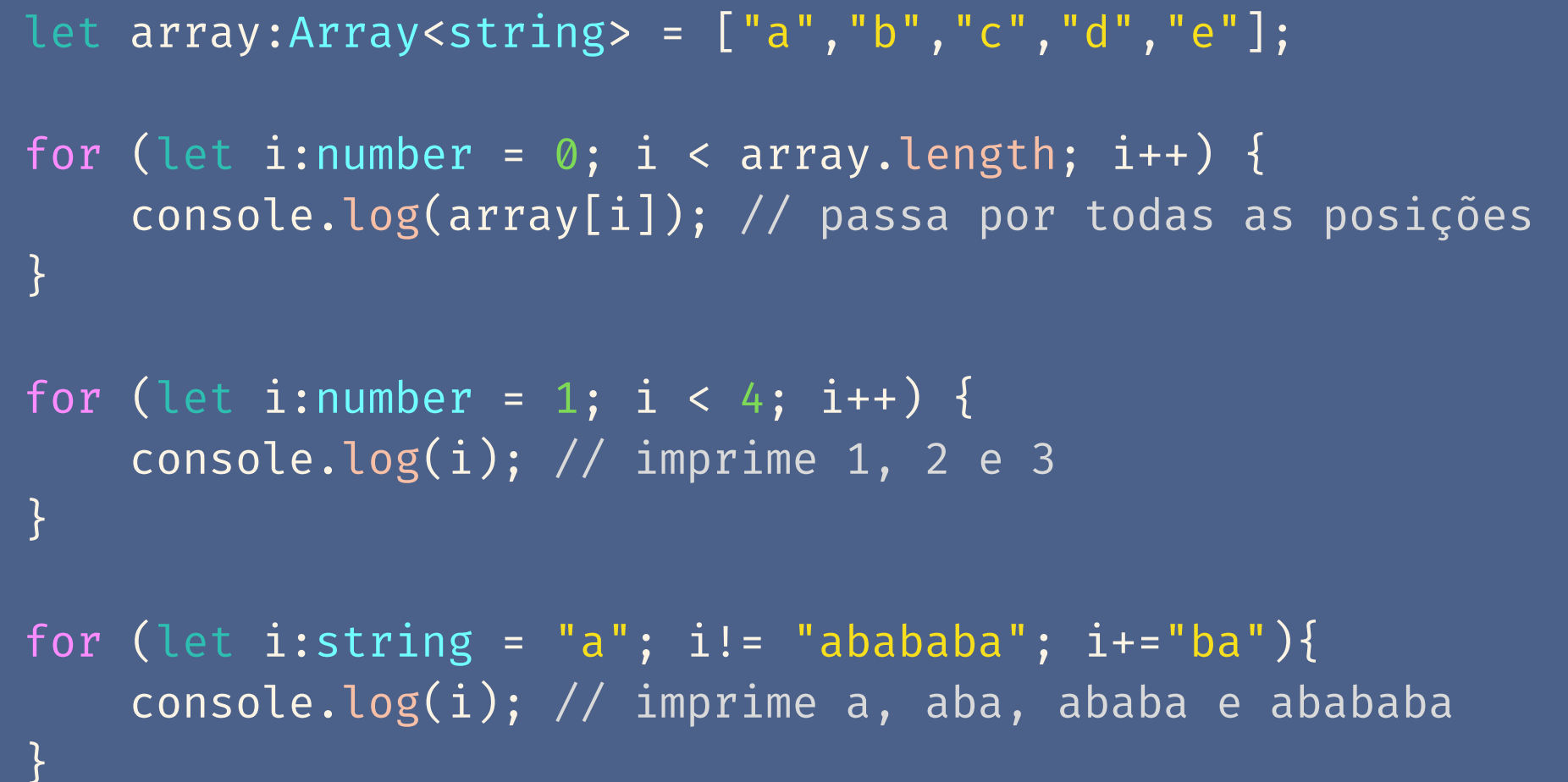
	
FOR LOOP	FOR IN
	
FOR OF	FOR EACH

FOR LOOP

Executa uma iteração com **início** e **término** determinados.

Não precisa estar ligada a um array.

O iterador pode ser de **qualquer** tipo.



```
let array:Array<string> = ["a","b","c","d","e"];

for (let i:number = 0; i < array.length; i++) {
  console.log(array[i]); // passa por todas as posições
}

for (let i:number = 1; i < 4; i++) {
  console.log(i); // imprime 1, 2 e 3
}

for (let i:string = "a"; i!= "abababa"; i+="ba"){
  console.log(i); // imprime a, aba, ababa e abababa
}
```

FOR IN

Executa uma iteração **SEMPRE** com **base** em um **array**

Sempre utiliza um iterador do tipo **string** como **contador**. Ex.: "0", "1", "2", "3",...

Utiliza uma **const** como base pois **recria** a variável **em cada iteração** e não permite a sua alteração.

```
let array:Array<string> = ["a","b","c","d","e"];

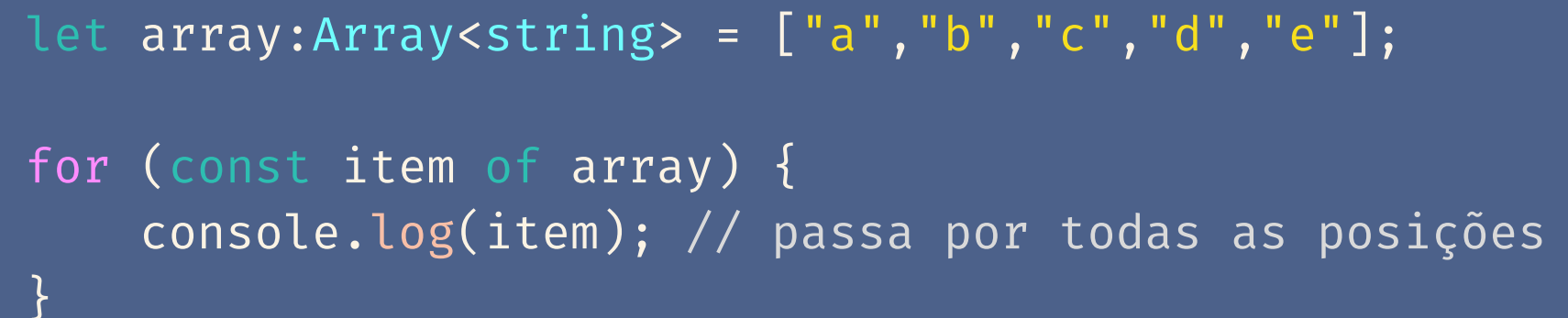
for (const i in array) {
    console.log(array[i]); // passa por todas as posições
}
```

FOR OF

Executa uma iteração **SEMPRE** com **base** em um **array**

Não possui um iterador. **Sempre** processa diretamente os **itens** do array.

Utiliza uma **const** como base pois **recria** a variável **em cada iteração** e não permite a sua alteração.



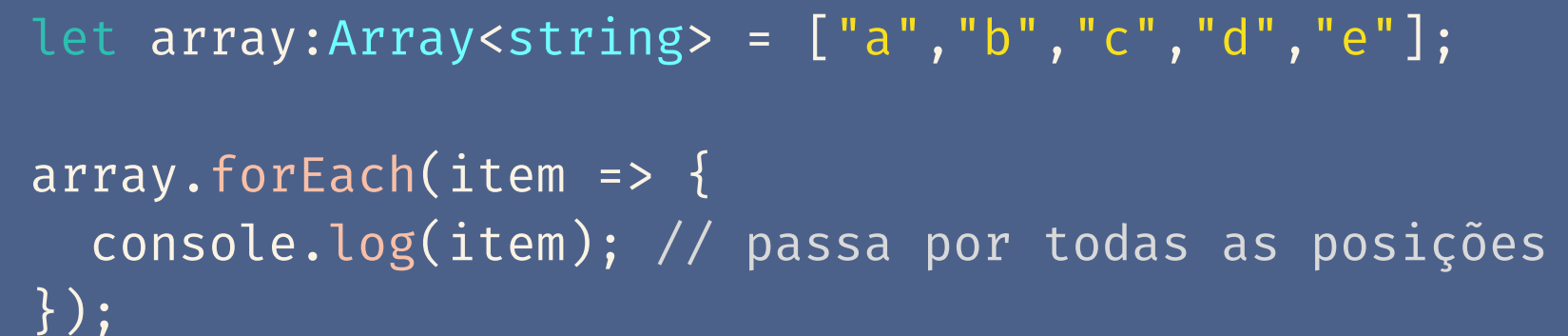
```
let array:Array<string> = ["a","b","c","d","e"];

for (const item of array) {
  console.log(item); // passa por todas as posições
}
```

FOR EACH

Executa uma iteração **SEMPRE** com **base** em um **array**

Não possui um iterador. **Sempre** processa diretamente os **itens** do array.

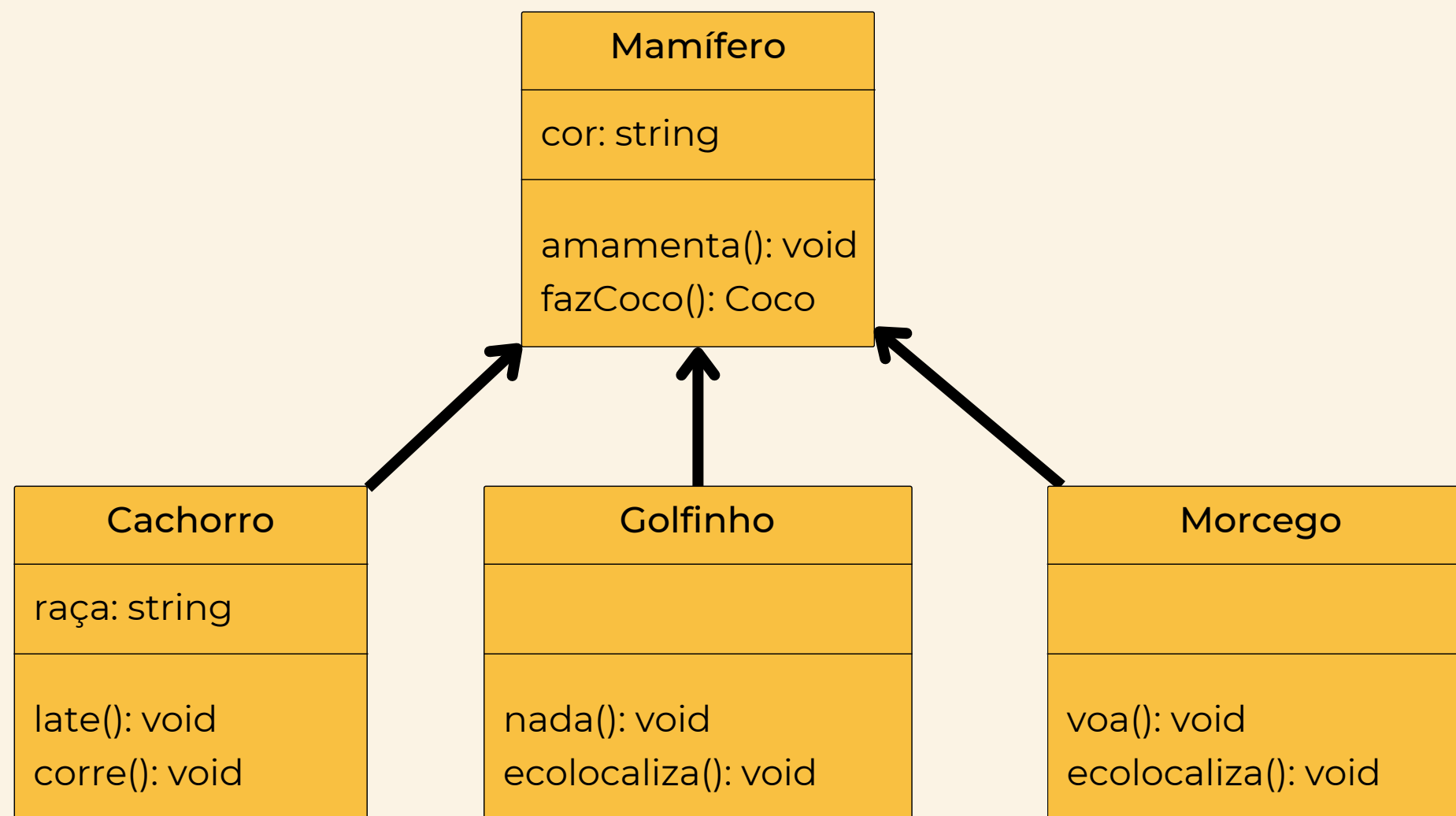
A code editor window with a dark blue background and a white title bar containing three colored window control buttons (red, yellow, green). The code is written in Kotlin and demonstrates the use of the `forEach` method on an array of strings.

```
let array:Array<string> = ["a","b","c","d","e"];

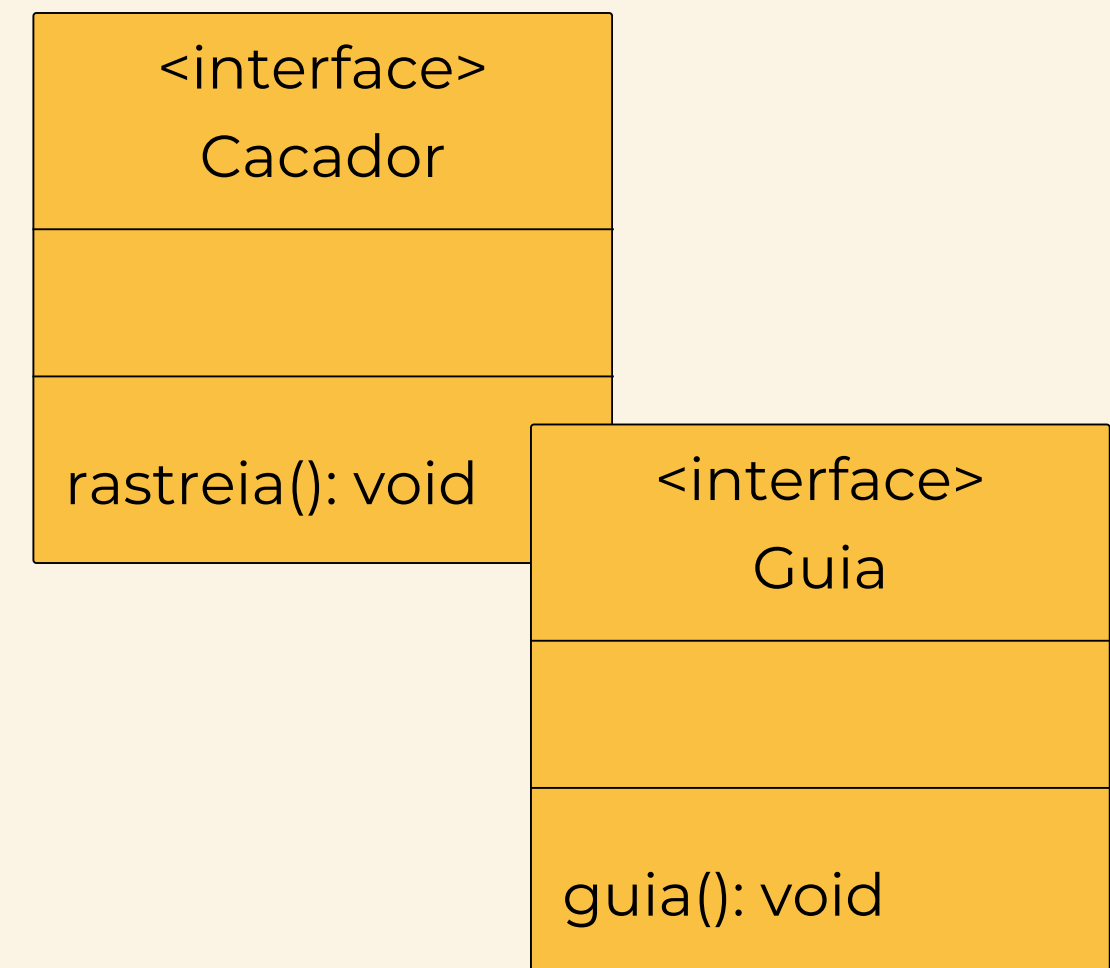
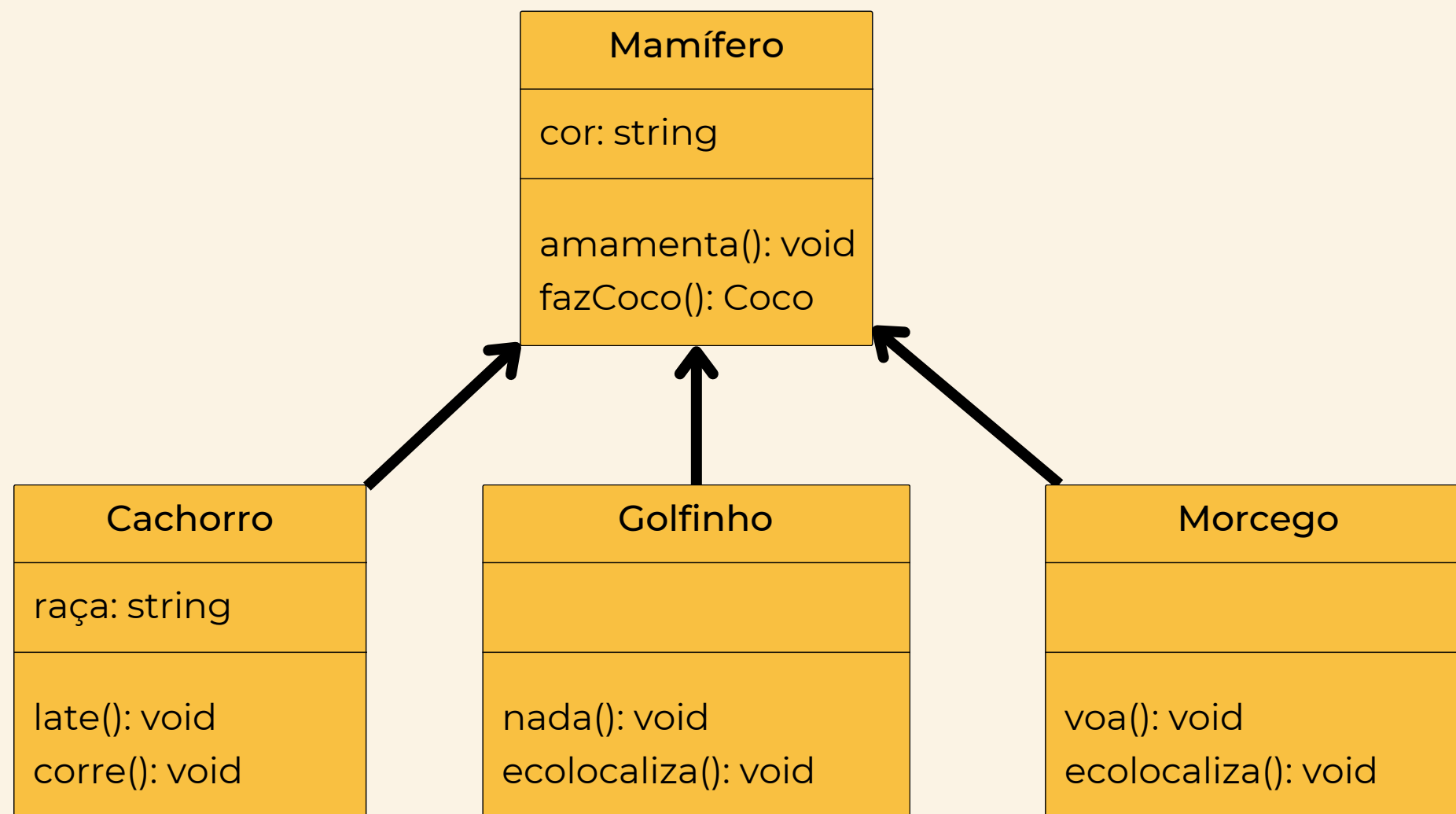
array.forEach(item => {
    console.log(item); // passa por todas as posições
});
```

diagramas de **CLASSE**

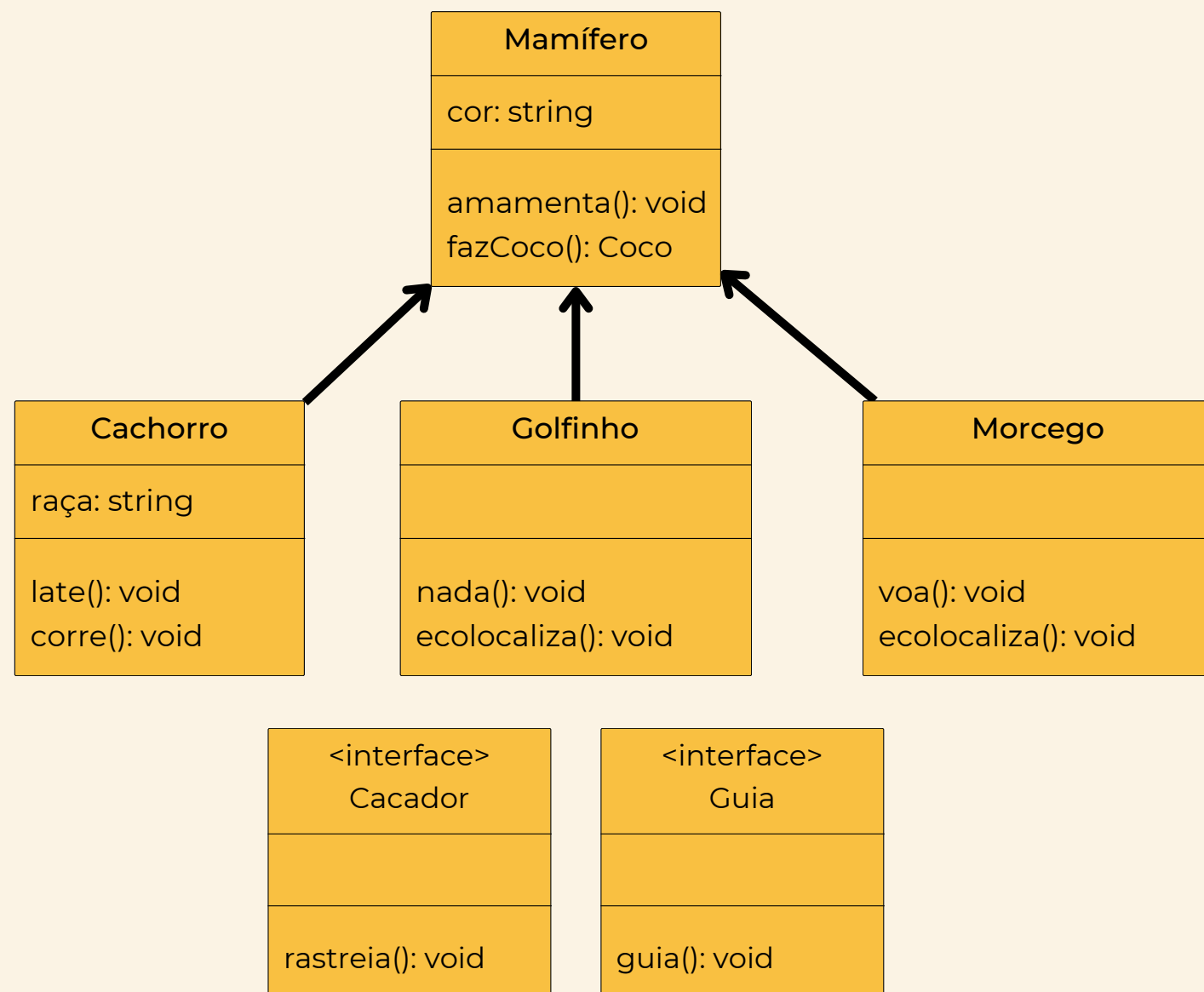
diagramas de CLASSE



diagramas de CLASSE



diagramas de CLASSE



O que são?

Formas de representar classes de sistemas de informação de maneira visual seguindo as normas da UML.

UML?

Linguagem de Modelagem Unificada (Unified Modeling Language). Conjunto de normas de representação de estruturas de sistemas de informação.

Normas?

As normas da UML contemplam diferentes tipos de diagramas que podem ser observados com mais profundidade [aqui](#)

CLASSE

Cachorro

+ `raca: string`

+ `late(): void`

+ `corre(): void`

CLASSE

Declaração da classe

Cachorro

+ `raca: string`

+ `late(): void`

+ `corre(): void`

CLASSE

Declaração da classe

Cachorro

+ `raca: string`

+ `late(): void`

+ `corre(): void`

Definição de atributos

CLASSE

Declaração da classe

Cachorro

+ `raca: string`

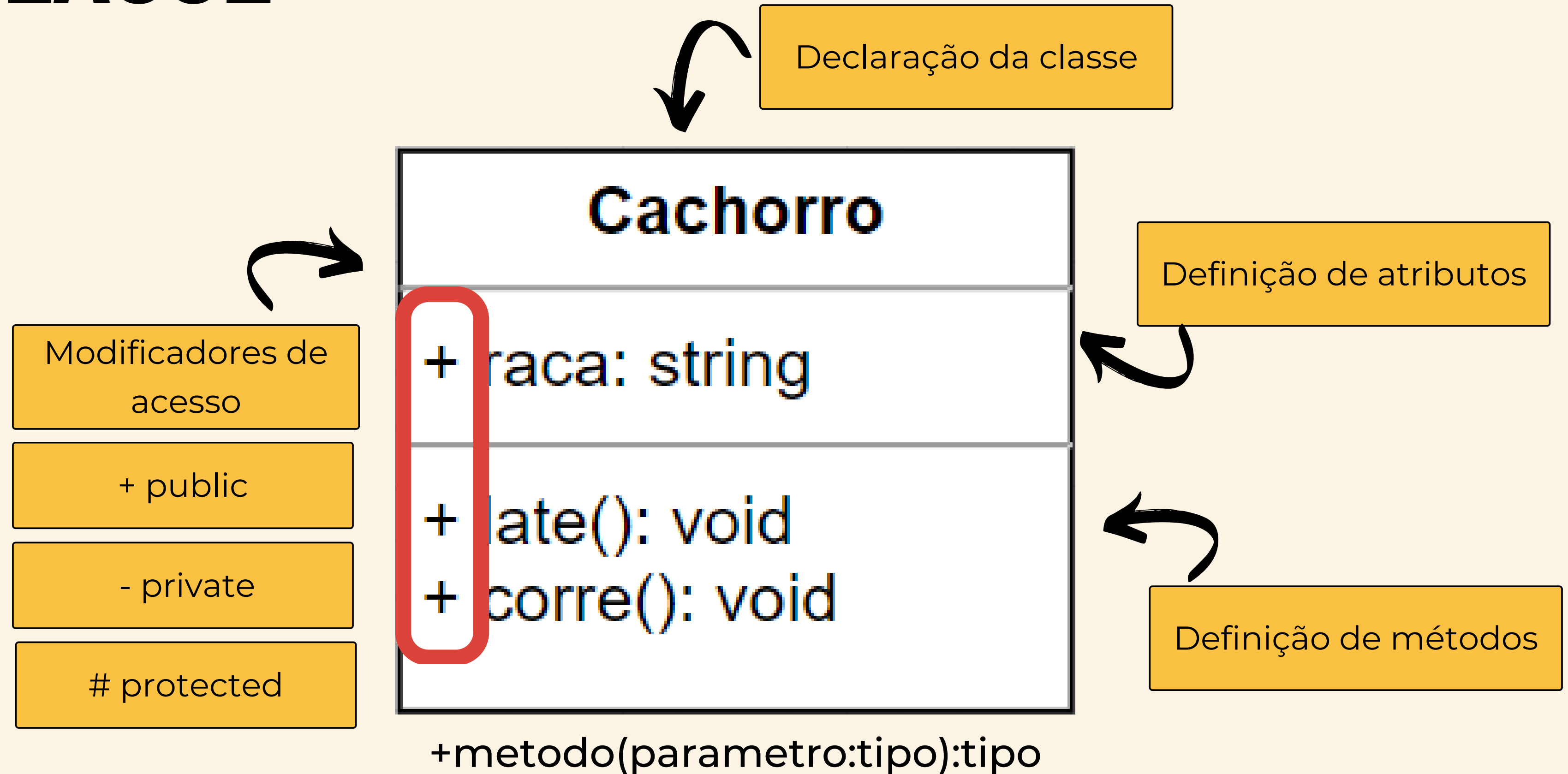
+ `late(): void`

+ `corre(): void`

Definição de atributos

Definição de métodos

CLASSE



CLASSE ABSTRATA

<<Abstract>>
Mamifero

+ cor: string

+ amamenta(): void

+ fazCoco(): void

+ *comunica(): void*

+ *desloca(): void*

CLASSE ABSTRATA

Modificador de classe

<<Abstract>>
Mamifero

+ cor: string

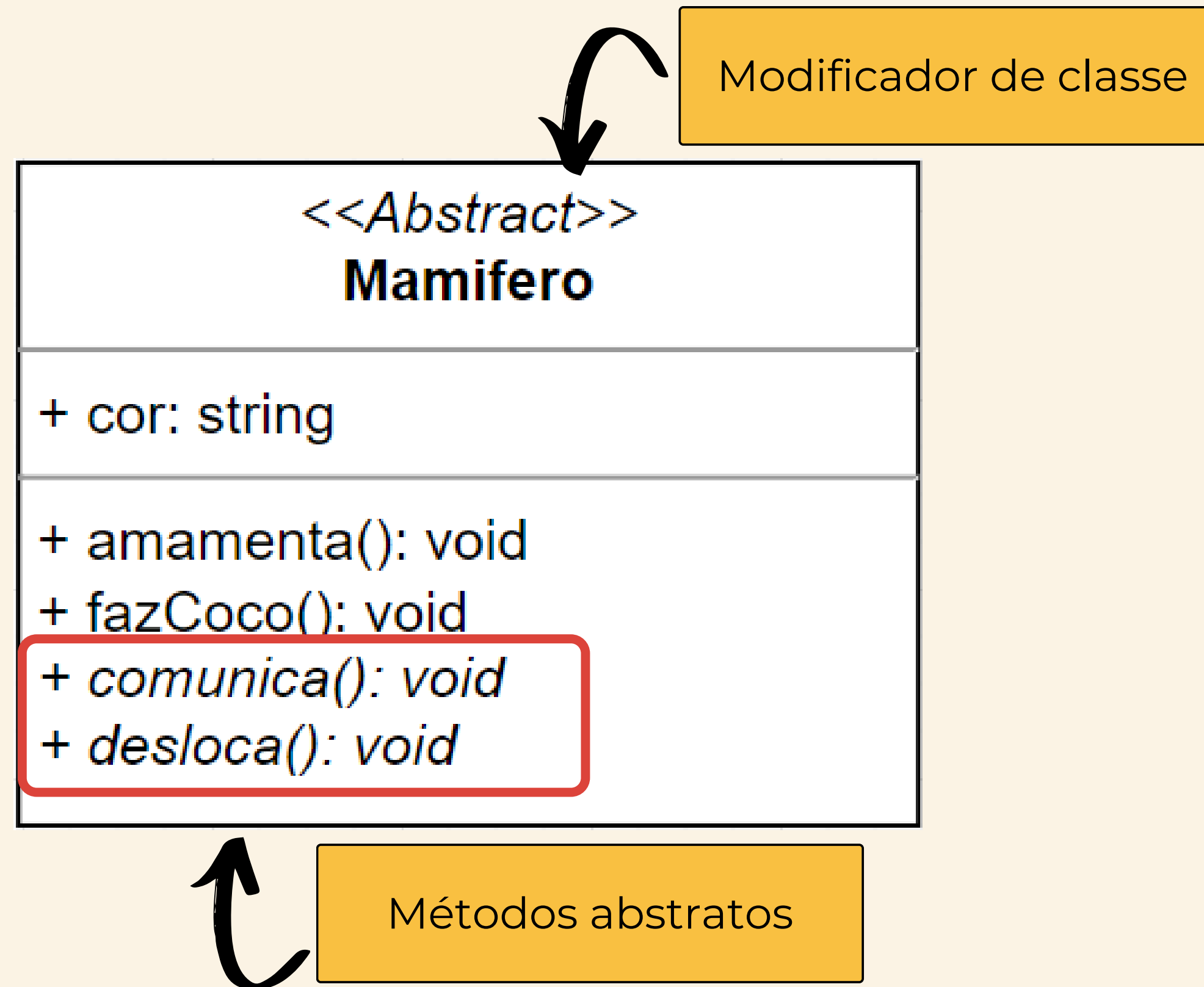
+ amamenta(): void

+ fazCoco(): void

+ *comunica(): void*

+ *desloca(): void*

CLASSE ABSTRATA



INTERFACE

<<Interface>>

Guia

+ guia(): void

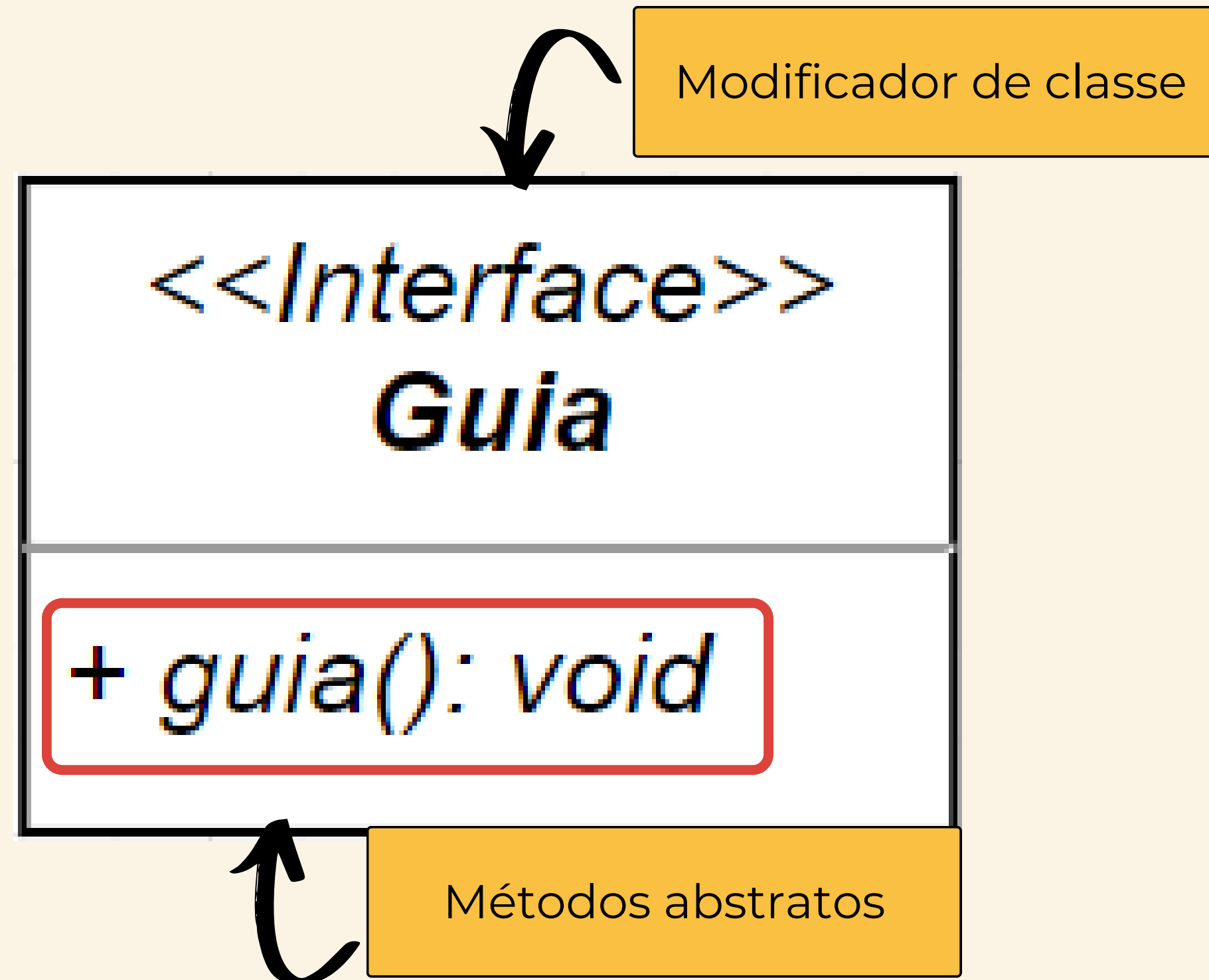
INTERFACE

Modificador de classe

<<Interface>>
Guia

+ *guia(): void*

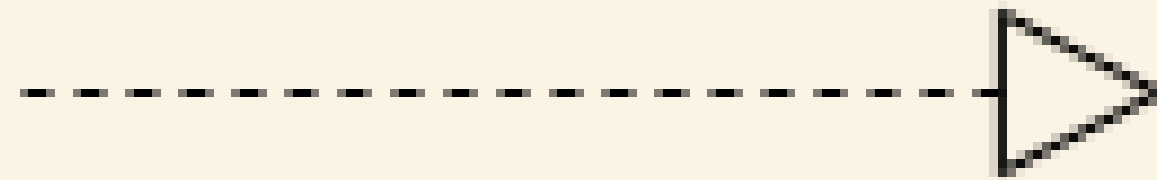
INTERFACE



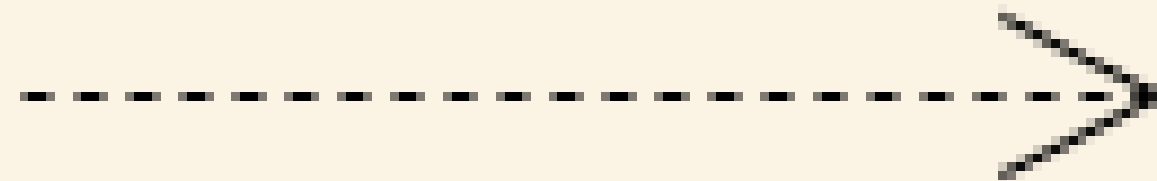
RELACIONAMENTOS



Herança

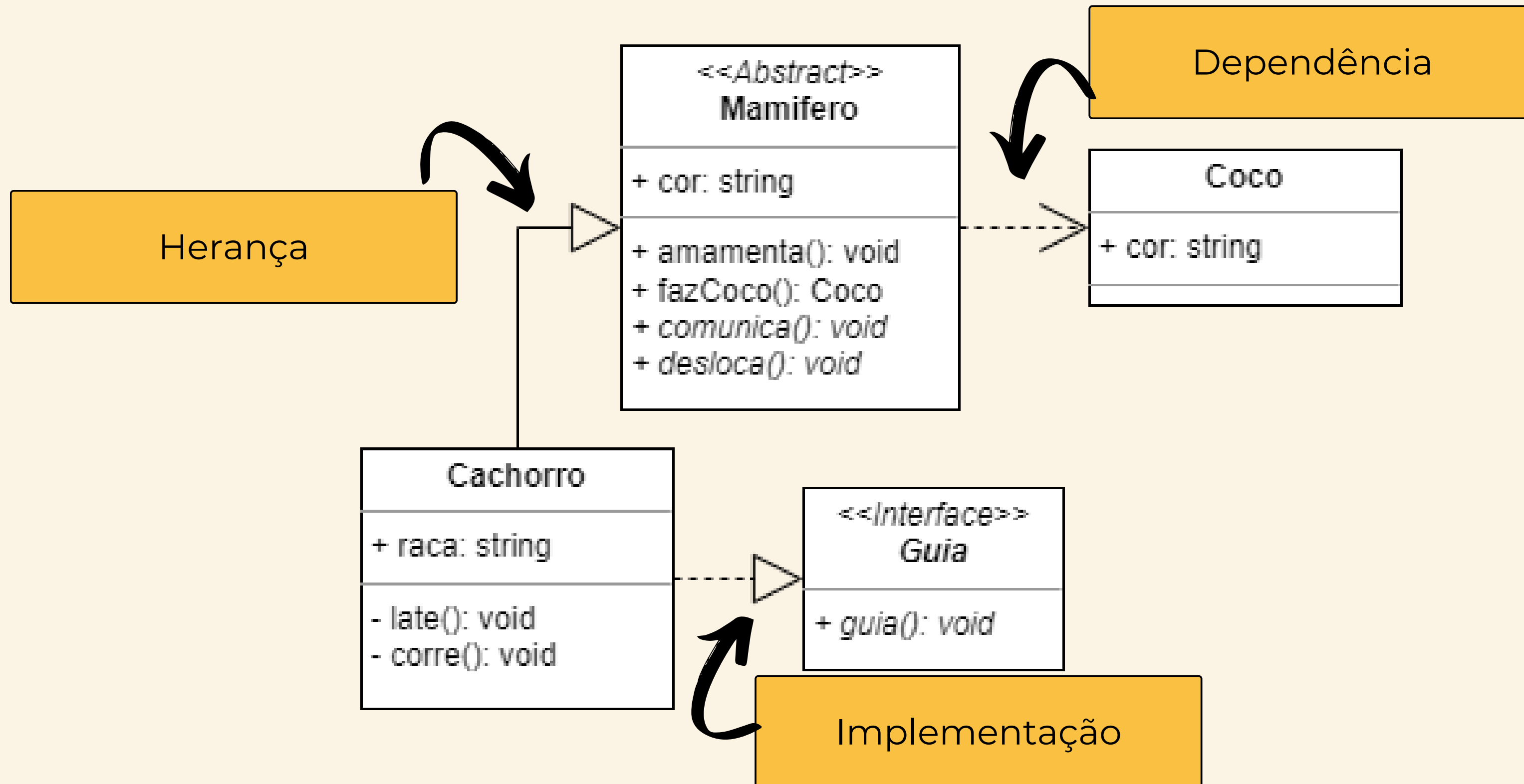


Implementação



Dependência

RELACIONAMENTOS



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

```
interface Guia {  
    guia(): void;  
}
```

```
interface Cacador {  
    rastreia(): void;  
}
```

```
class Cachorro extends Mamifero implements Guia, Cacador {  
    raca: string;  
  
    constructor(cor: string, raca: string) {  
        super(cor)  
        this.raca = raca;  
    }  
  
    // métodos  
  
    rastreia(): void {  
        // implementação do método  
    }  
  
    guia(): void {  
        // implementação do método  
    }  
}
```

projeto
INTEGRADOR

projeto **INTEGRADOR** **POO + BD**

Modelar classes e entidades necessárias ao sistema.

A entrega de POO deve ser feita via
projeto TS e/ou diagrama de classes