INTERFACES

FAPESC – DESENVOLVEDORES PARA TECNOLOGIA DA INFORMAÇÃO

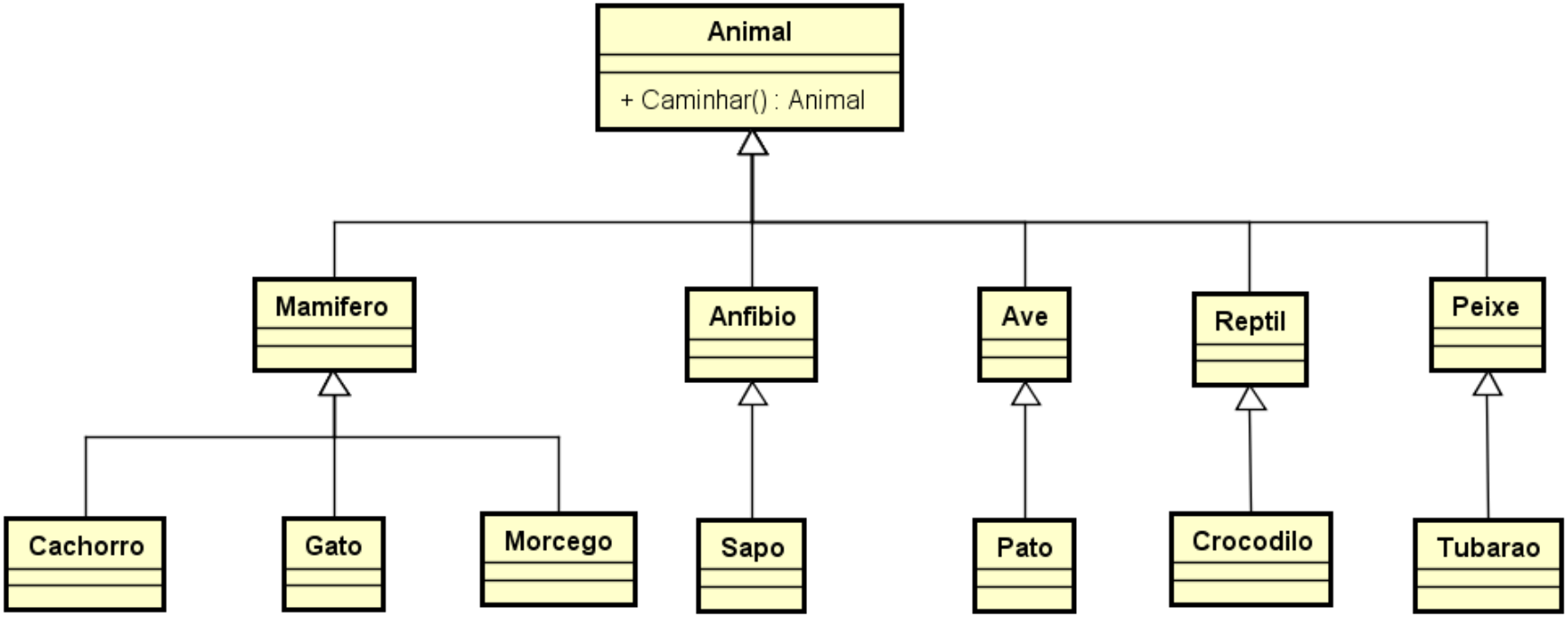
HERCULANO DE BIASI
herculano.debiasi@unoesc.edu.br

Tópicos

- Motivação
- *Interfaces*
- Prática
- *Interfaces* vs. Classes abstratas
- Boas práticas de POO (Programação Orientada a Objetos)
- Exercícios

Motivação

- A natureza é um caos 😉



Motivação

- Aonde encaixar um ornitorrinco?
  - É um mamífero ovíparo, bota ovo como as aves
  - Tem bico e nadadeiras como as aves

UNOESC

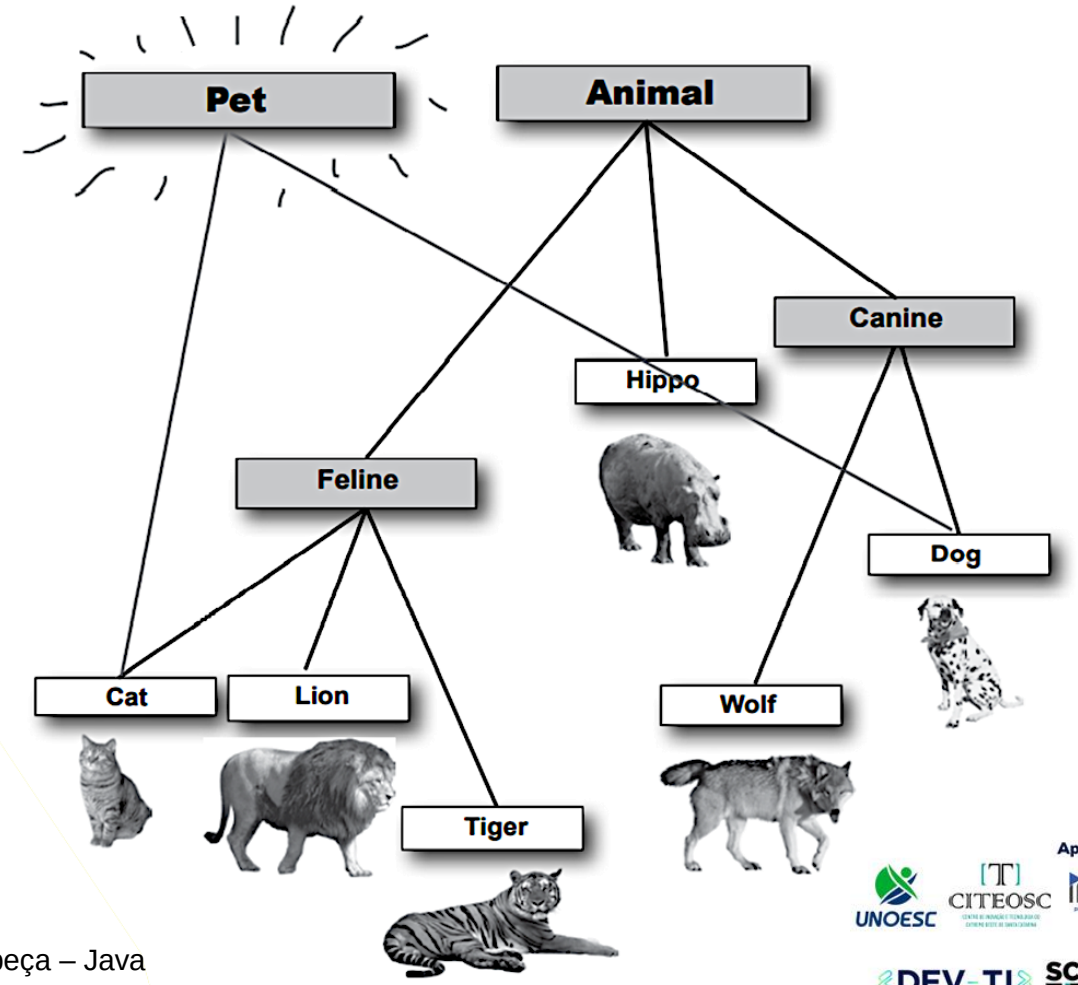
Apoiadores:
UNOESC CITEOSC INOVALE CENTRO DE INOVAÇÃO VIDEIRA
DEV-TI SC TEC fapesc GOVERNO DE SANTA CATARINA

Motivação

- E como modelar felinos?
  - Gatos, leões, tigres, pumas, onças, panteras, leopardos, guepardos, jaguatiricas, etc
  - Guepardos possuem garras semirretráteis, ao contrário dos outros felinos
  - Pumas não rugem, mas tigres, leões, onças e os leopardos, com exceção do leopardo das neves, rugem
  - Leopardos, onças e panteras são semiarborícolas, leões não
  - Leões vivem em grupos, tigres são solitários
  - Linces possuem várias adaptações para viverem na neve
  - Gatos são animais domésticos, os outros felinos não
  - Tigres são excelentes nadadores

Motivação

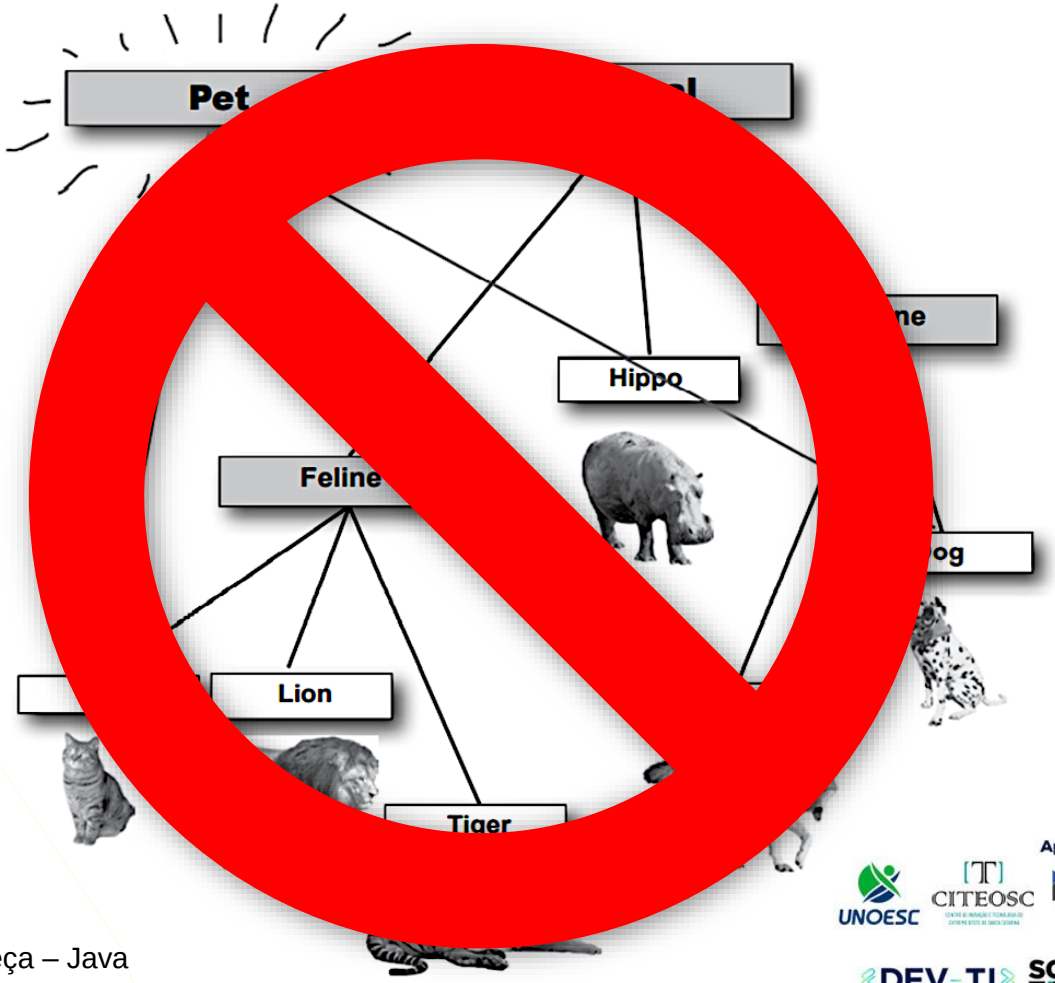
- Herdar de mais de uma classe pai (herança múltipla) resolveria o problema



Fonte: Use a Cabeça – Java

Motivação

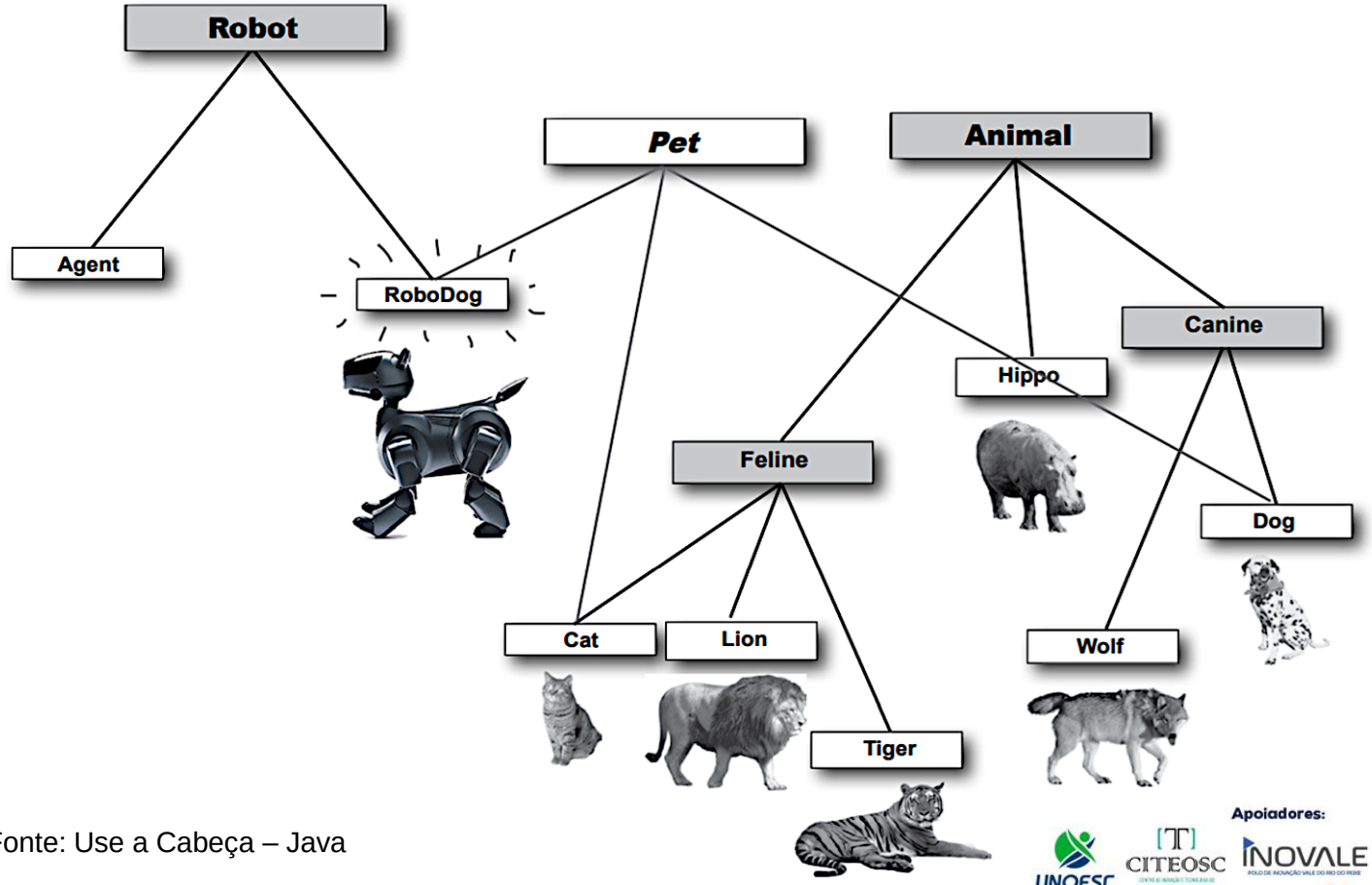
- Mas Java não possui herança múltipla



Fonte: Use a Cabeça – Java

INTERFACES

- Solução: *Interfaces*, que podem ser utilizadas até mesmo em árvores diferentes



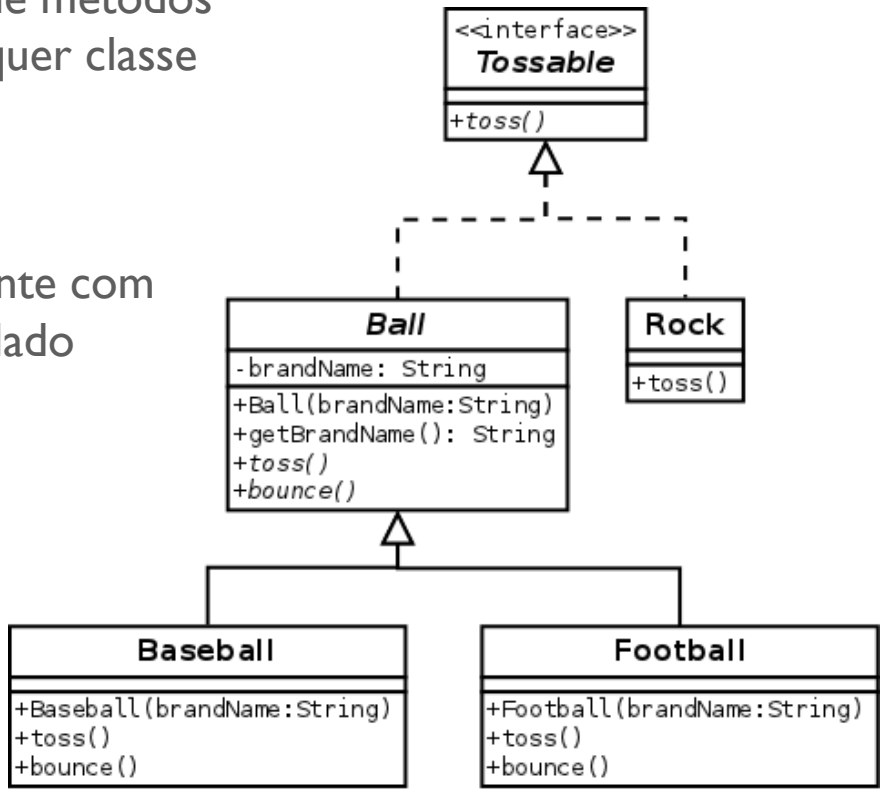
Fonte: Use a Cabeça – Java

INTERFACES

- Uma *interface* é um contrato, ela define métodos com uma assinatura específica e qualquer classe que a implemente precisa defini-los

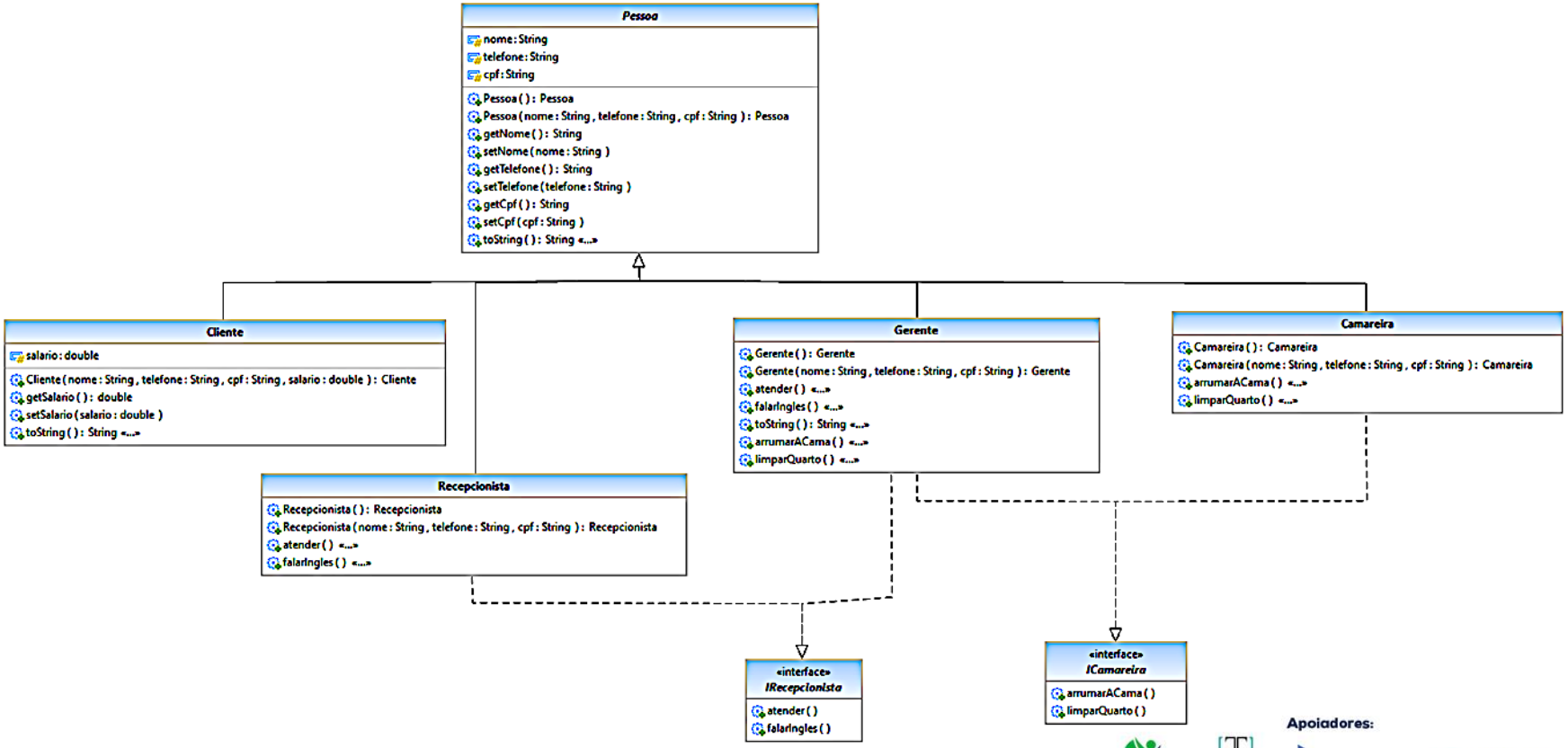
- É possível combinar *interface* juntamente com classes abstratas, como mostrado ao lado

- Legenda do diagrama
  - A linha pontilhada indica que está sendo usada herança de interface (Java usa a palavra `implements`)
  - A linha sólida indica o uso de herança por especialização (Java usa a palavra `extends`)



PRÁTICA

- Prática: Implementar o modelo orientado a objetos das funções de um hotel, representando as *interfaces* e classes mostradas no modelo abaixo



PRÁTICA

- *Interfaces*

herculano - ICamareira.java

```java
package interfaces;

public interface ICamareira {
    void arrumarACama();
    void limparQuarto();
}
```

herculano - IRecepcionista.java

```java
package interfaces;

public interface IRecepcionista {
    void atender();
    void falarIngles();
}
```

PRÁTICA

- Classe `Pessoa`

herculano - Pessoa.java

```java
package model;

public abstract class Pessoa {
    protected String nome;
    protected String telefone;
    protected String cpf;

    public Pessoa(String nome, String telefone, String cpf) {
        super();
        this.nome = nome;
        this.telefone = telefone;
        this.cpf = cpf;
    }

    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }

    public String getTelefone() { return telefone; }
    public void setTelefone(String telefone) { this.telefone = telefone; }

    public String getCpf() { return cpf; }
    public void setCpf(String cpf) { this.cpf = cpf; }

    @Override
    public String toString() {
        return "Pessoa [nome=" + nome + ", telefone=" + telefone + ", cpf=" + cpf + "]";
    }
}
```

UNOESC

PRÁTICA

- Classe `Cliente`

modulo2 - TesteConversoes.java

```java
package model;

public class Cliente extends Pessoa {
    protected double salario;

    public Cliente(String nome, String telefone, String cpf, double salario) {
        super(nome, telefone, cpf);
        this.salario = salario;
    }

    public double getSalario() {
        return salario;
    }

    public void setSalario(double salario) {
        this.salario = salario;
    }

    @Override
    public String toString() {
        return "Nome: " + nome + "\nCliente [salario=" + salario + "]";
    }
}
```

UNOESC

PRÁTICA

- Classe `Camareira`

modulo2 - TesteConversoes.java

```java
package model;

import interfaces.ICamareira;

public class Camareira extends Pessoa implements ICamareira {
    public Camareira() {
        super();
    }

    public Camareira(String nome, String telefone, String cpf) {
        super(nome, telefone, cpf);
    }

    @Override
    public void arrumarACama() {
        System.out.println("Arrumo camas...");
    }

    @Override
    public void limparQuarto() {
        System.out.println("Limpo quartos...");
    }
}
```

UNOESC

PRÁTICA

- Classe `Recepcionista`

modulo2 - TesteConversoes.java

```java
package model;

import interfaces.IRecepcionista;

public class Recepcionista extends Pessoa implements IRecepcionista {
    public Recepcionista() {
        super();
    }

    public Recepcionista(String nome, String telefone, String cpf) {
        super(nome, telefone, cpf);
    }

    @Override
    public void atender() {
        System.out.println("Recebo clientes no hotel...");
    }

    @Override
    public void falarIngles() {
        System.out.println("Falo inglês meia boca...");
    }
}
```

UNOESC

PRÁTICA

- Classe `Gerente`

herculano - Gerente.java

```java
package model;

import interfaces.ICamareira;
import interfaces.IRecepcionista;

public class Gerente extends Pessoa implements ICamareira, IRecepcionista {

    public Gerente(String nome, String telefone, String cpf) {
        super(nome, telefone, cpf);
    }

    @Override
    public void arrumarACama() {
        System.out.println("Sei como deve ficar uma cama arrumada...");
    }

    @Override
    public void limparQuarto() {
        System.out.println("Sei como deve ficar um quarto limpo...");
    }

    @Override
    public void atender() {
        System.out.println("Sei mais ou menos como atender...");
    }

    @Override
    public void falarIngles() {
        System.out.println("Sei falar inglês muito bem...");
    }
}
```

UNOESC

PRÁTICA

- Classe `Principal`

herculano - Principal.java

```java
import model.Camareira;
import model.Cliente;
import model.Gerente;
import model.Recepcionista;

public class Principal {

    public static void main(String[] args) {
        System.out.println("Hotel Java\n");

        Cliente cliente = new Cliente("Cliente", "(11) 2222-3333", "111.111.111-11");
        Recepcionista recepcionista = new Recepcionista("Recepcionista", "(22) 1234-5678", "222.222.222-22");
        Camareira camareira = new Camareira("Camareira", "(33) 4321-8765", "333.333.333-33");
        Gerente gerente = new Gerente("Gerente", "(44) 6666-6666", "444.444.444-44");

        System.out.println("Recepcionista...");
        recepcionista.atender();
        recepcionista.falarIngles();
        System.out.println();

        System.out.println("Camareira...");
        camareira.arrumarACama();
        camareira.limparQuarto();
        System.out.println();

        System.out.println("Gerente...");
        gerente.arrumarACama();
        gerente.falarIngles();
        gerente.atender();
        gerente.limparQuarto();
    }

}
```

UNOESC

INTERFACES VS. CLASSES ABSTRATAS

- Cuidado com a herança de implementação (`extends`) !

Uma vez fui a uma reunião do grupo de usuários Java, onde James Gosling (inventor do Java) foi o orador de destaque. Durante o memorável Q&A sessão, alguém lhe perguntou: "Se você pudesse fazer Java novamente, o que você mudaria?" "Eu deixaria de fora as Classes", ele respondeu. Após sessarem os risos, ele explicou que o verdadeiro problema não era as Classes em si, mas sim a Herança de Implementação (extends). Herança de Interface (implements) é preferível. Você deve evitar a Herança de Implementação, sempre que possível.

Why extends is evil — JavaWorld

Interfaces vs. Classes Abstratas

Interface ou classe abstrata?

Várias vezes já me fizeram a pergunta: *"quando devo utilizar uma classe abstrata e quando devo utilizar uma interface?"*. Tanto as classes abstratas quanto as interfaces podem definir métodos abstratos que precisam ser implementados pelas classes que respectivamente a estende ou implementa. Porém apenas as classes abstratas podem possuir métodos concretos e atributos. Apesar dessa diferença, a resposta para pergunta é mais conceitual do que relacionada com questões de linguagem.

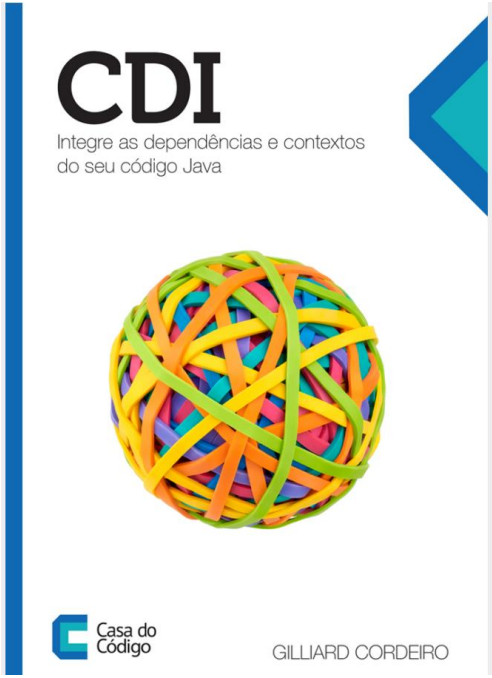
Quando a abstração que precisar ser criada for um conceito, ou seja, algo que possa ser refinado e especializado, deve-se utilizar uma classe abstrata. Quando a abstração é um comportamento, ou seja, algo que uma classe deve saber fazer, então a melhor solução é a criação de uma interface. Imagine um jogo no qual existem naves que se movem. Se sua abstração representa uma nave, então você está representando um conceito e deve utilizar uma classe abstrata. Por outro lado, se sua abstração representa algo que se move, então o que está sendo abstraído é um comportamento e a melhor solução é usar uma interface.

UNOESC

Apoiadores:
UNOESC CITEOSC INOVALE CENTRO DE INOVAÇÃO VIDEIRA DEV-TI SC TEC fapesc SANTA CATARINA

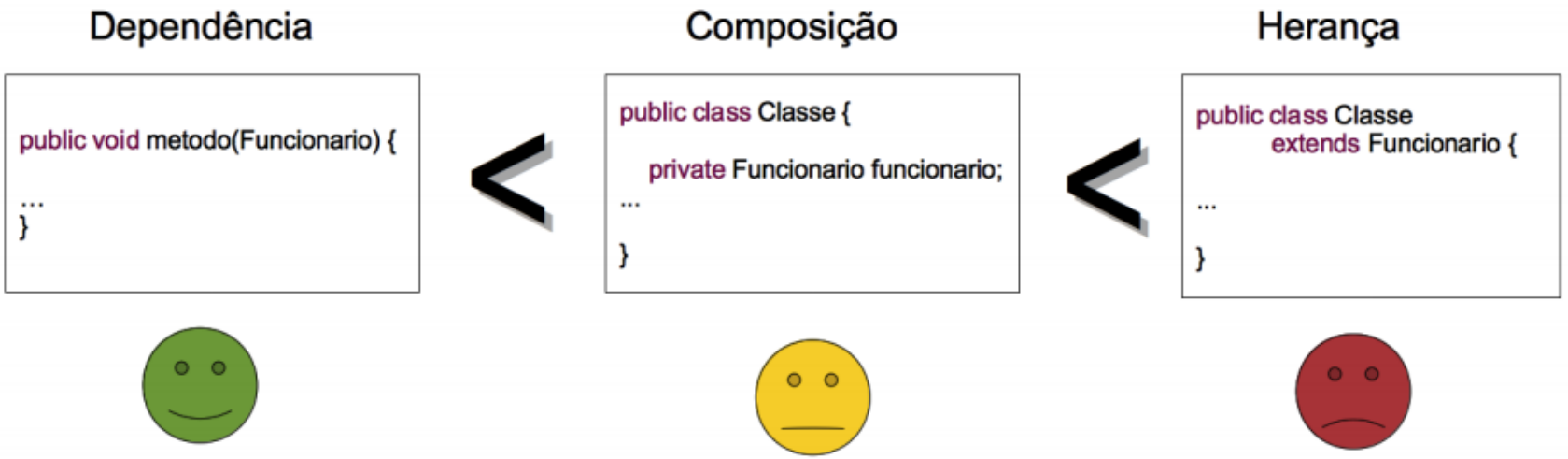
Boas práticas de POO

- É desejável que um modelo orientado a objetos apresente as características de alta coesão e baixo acoplamento
  - Coesão: Objeto deve estar focado em fazer o que se propõe
    - Ou seja, devem fazer apenas uma tarefa (e bem feita)
    - Esta característica é chamada de princípio da responsabilidade única (SRP), sendo um dos postulados SOLID
    - Quanto mais coeso um objeto, melhor
    - Um método com o nome `imprimirSoma()` deveria somente imprimir a soma e não calculá-la e imprimi-la



Boas práticas de POO

- É desejável um modelo com alta coesão e baixo acoplamento
  - Acoplamento: O quanto uma classe depende de outra para funcionar
    - Quanto maior for a dependência entre as classes, mais fortemente acoplada elas estão
    - Ou seja, quanto menor for o acoplamento, melhor



Fonte: CDI - Integre as dependências e contextos do seu código Java

Exercícios

1. Adicione, no exemplo feito do sistema de hotel, no mínimo mais três *interfaces* e três classes que implementem essas *interfaces*; se possível utilize essas *interfaces* recém-criadas nas classes já existentes. Teste essas classes no programa principal.

   Exemplos de *interfaces:*

   - `ICozinheira`
   - `IFazTudo`
   - `IAuxiliarGeral`

2. Crie classes e *interfaces* de forma a modelar a hierarquia de felinos descrita no *slide* 5. Crie um programa principal para testar essas classes.