



INTERFACES

FAPESC – DESENVOLVEDORES PARA TECNOLOGIA DA INFORMAÇÃO

HERCULANO DE BIASI

herculano.debiasi@unoesc.edu.br

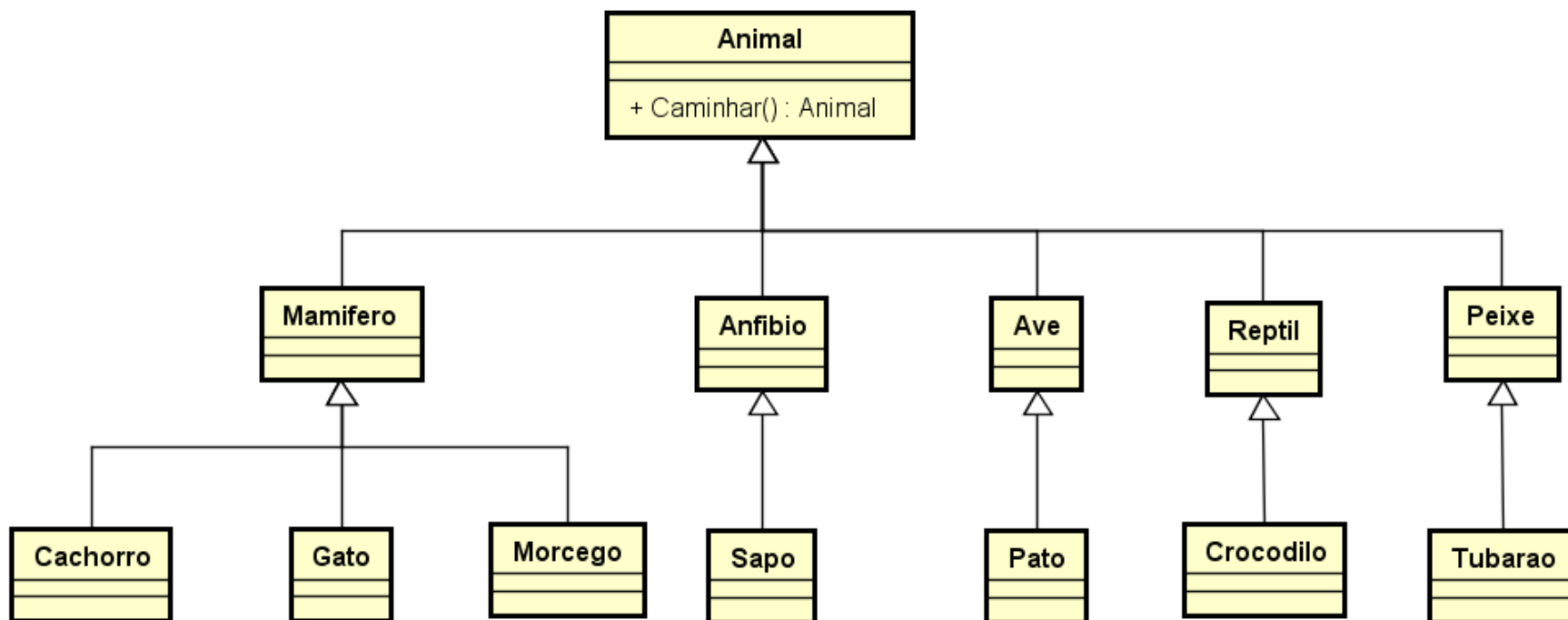


TÓPICOS

- 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

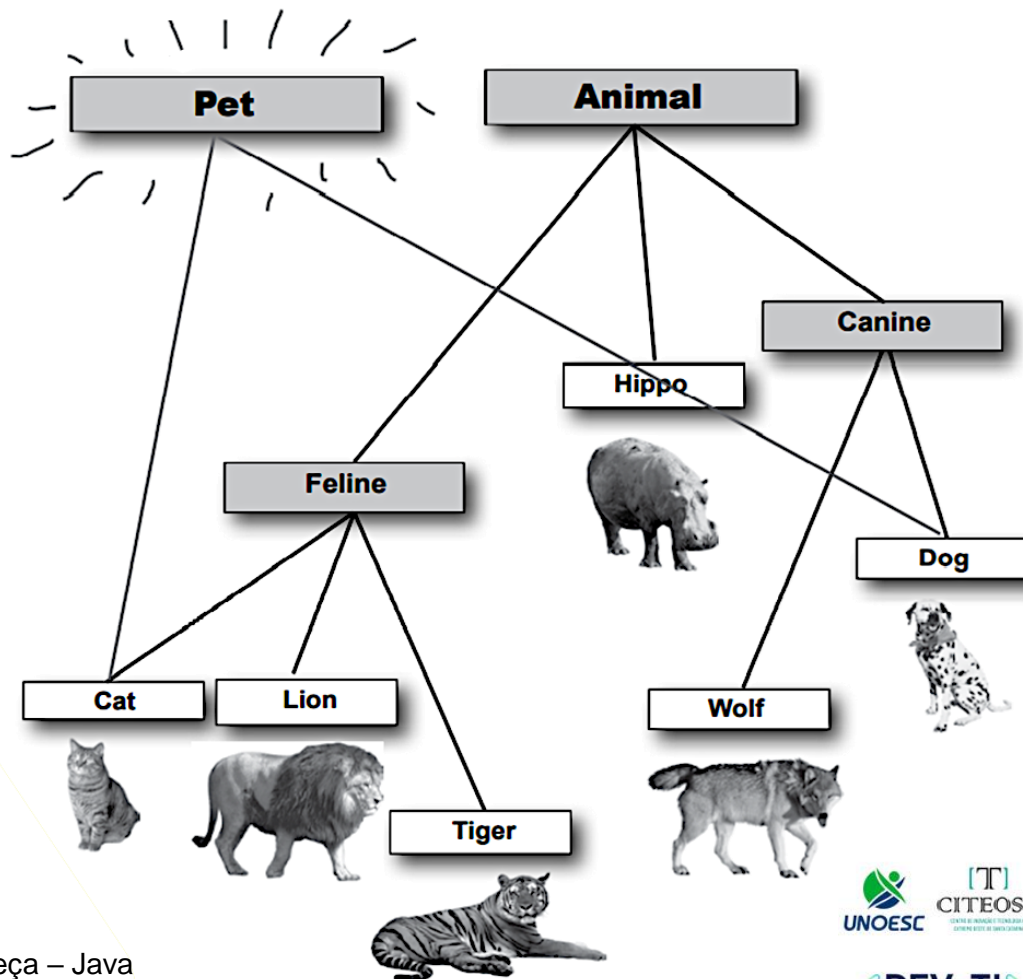


MOTIVAÇÃO

- 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
 - Lince possuem várias adaptações para viverem na neve
 - Gatos são animais domésticos, os outros felinos não
 - Tigres são excelentes nadadores

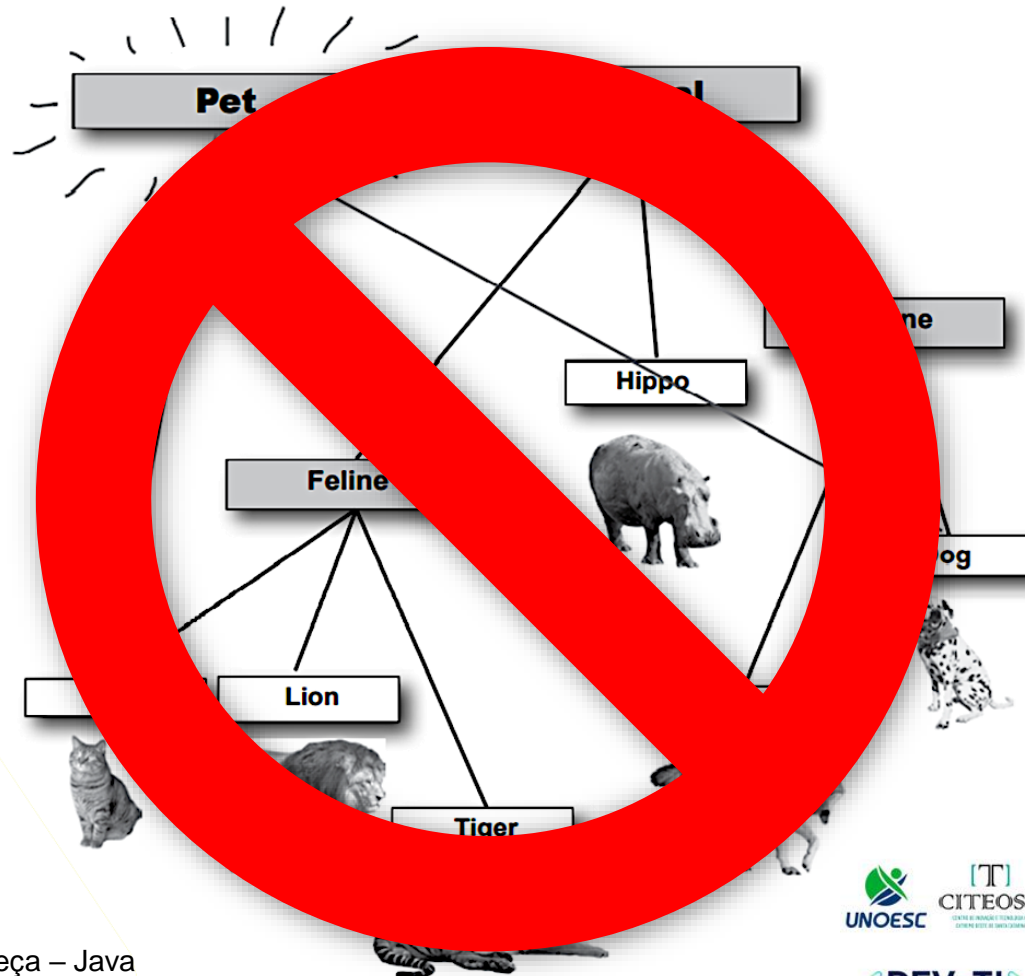
MOTIVAÇÃO

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



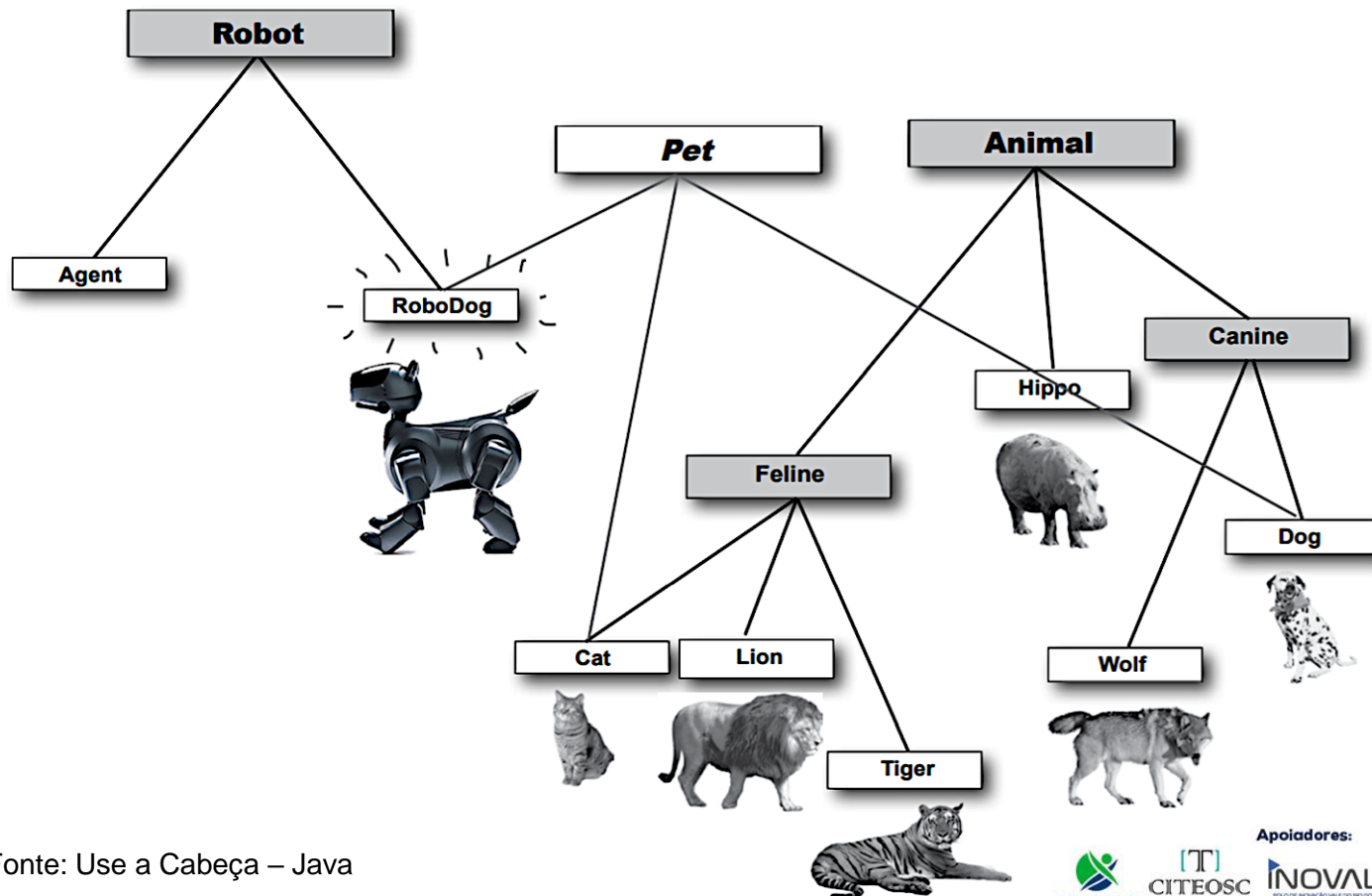
MOTIVAÇÃO

- Mas Java não possui herança múltipla



INTERFACES

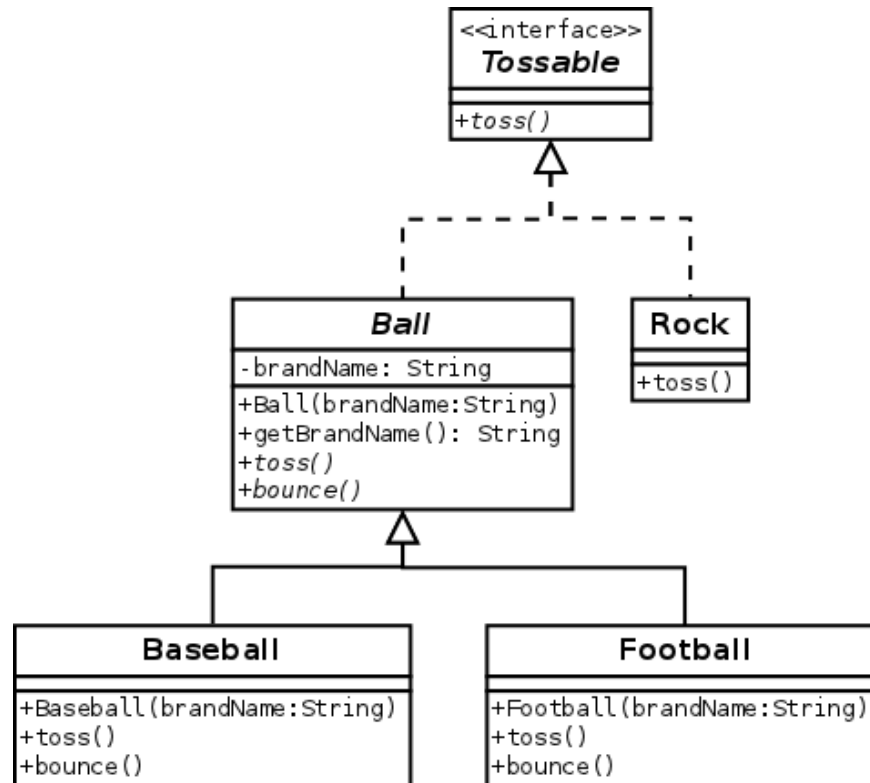
- Solução: *Interfaces*, que podem ser utilizadas até mesmo em árvores diferentes



Fonte: Use a Cabeça – Java

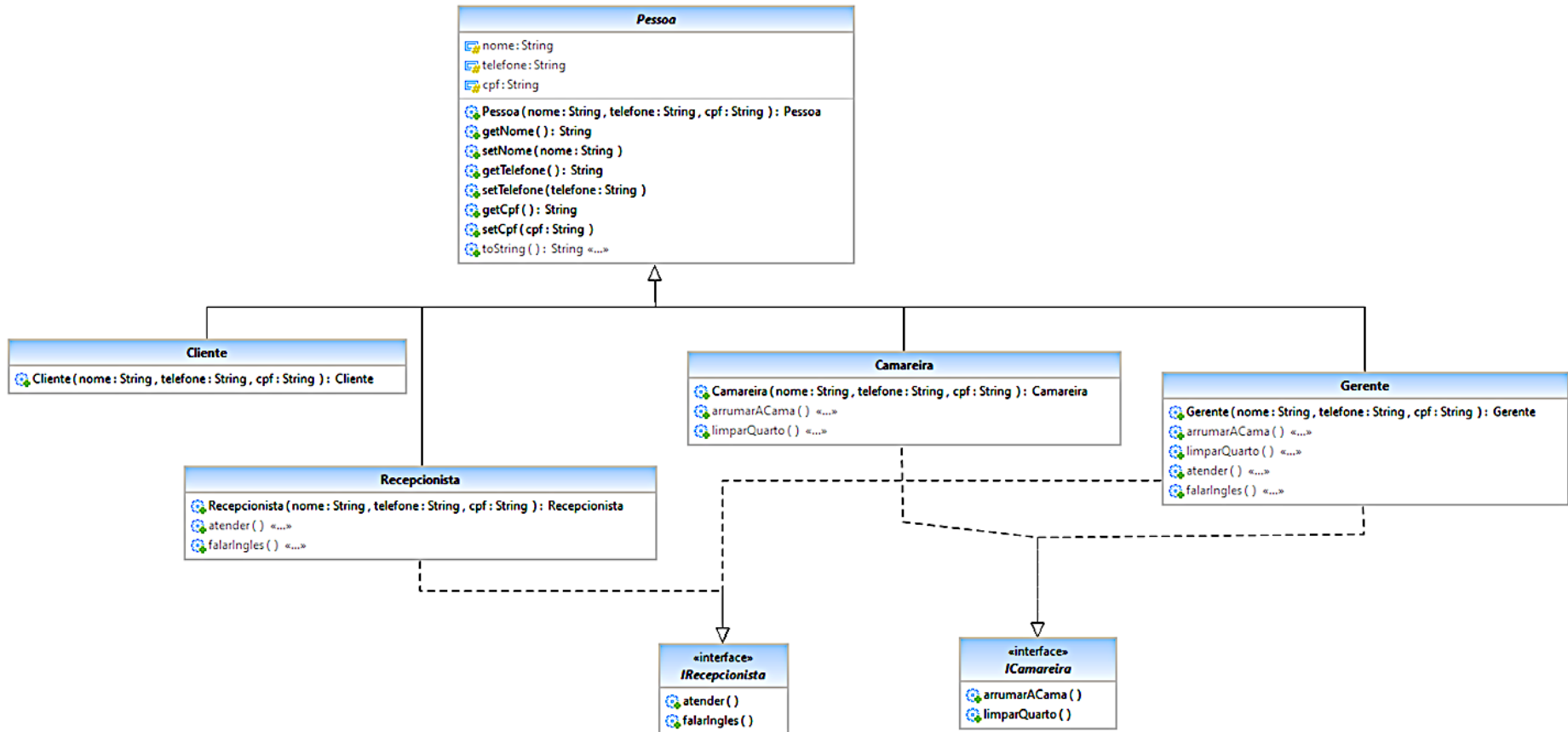
INTERFACES

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



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

```
1 package interfaces;
2
3 public interface ICamareira {
4     void arrumarACama();
5     void limparQuarto();
6 }
```

herculano - IRecepcionista.java

```
1 package interfaces;
2
3 public interface IRecepcionista {
4     void atender();
5     void falarIngles();
6 }
```

PRÁTICA

■ Classe Pessoa

```
herculano - Pessoa.java
1 package model;
2
3 public abstract class Pessoa {
4     protected String nome;
5     protected String telefone;
6     protected String cpf;
7
8     public Pessoa(String nome, String telefone, String cpf) {
9         super();
10        this.nome = nome;
11        this.telefone = telefone;
12        this.cpf = cpf;
13    }
14
15    public String getNome() { return nome; }
16    public void setNome(String nome) { this.nome = nome; }
17
18    public String getTelefone() { return telefone; }
19    public void setTelefone(String telefone) { this.telefone = telefone; }
20
21    public String getCpf() { return cpf; }
22    public void setCpf(String cpf) { this.cpf = cpf; }
23
24    @Override
25    public String toString() {
26        return "Pessoa [nome=" + nome + ", telefone=" + telefone + ", cpf=" + cpf + "];"
27    }
28 }
```

PRÁTICA

■ Classe Cliente

```
herculano - Cliente.java
1 package model;
2
3 public class Cliente extends Pessoa {
4
5     public Cliente(String nome, String telefone, String cpf) {
6         super(nome, telefone, cpf);
7     }
8
9 }
```

PRÁTICA

■ Classe Camareira

```
herculano - Camareira.java
1 package model;
2
3 import interfaces.ICamareira;
4
5 public class Camareira extends Pessoa implements ICamareira {
6
7     public Camareira(String nome, String telefone, String cpf) {
8         super(nome, telefone, cpf);
9     }
10
11     @Override
12     public void arrumarACama() {
13         System.out.println("Sei arrumar a cama...");
14     }
15
16     @Override
17     public void limparQuarto() {
18         System.out.println("Sei arrumar o quarto...");
19     }
20
21 }
```

PRÁTICA

■ Classe Recepcionista

```
herculano - Recepcionista.java

1 package model;
2
3 import interfaces.ICamareira;
4 import interfaces.IRecepcionista;
5
6 public class Recepcionista extends Pessoa implements IRecepcionista {
7
8     public Recepcionista(String nome, String telefone, String cpf) {
9         super(nome, telefone, cpf);
10    }
11
12    @Override
13    public void atender() {
14        System.out.println("Sei atender...");
15    }
16
17    @Override
18    public void falarIngles() {
19        System.out.println("Sei falar inglês...");
20    }
21
22 }
```

PRÁTICA

■ Classe Gerente

```
herculano - Gerente.java
1 package model;
2
3 import interfaces.ICamareira;
4 import interfaces.IRecepcionista;
5
6 public class Gerente extends Pessoa implements ICamareira, IRecepcionista {
7
8     public Gerente(String nome, String telefone, String cpf) {
9         super(nome, telefone, cpf);
10    }
11
12    @Override
13    public void arrumarACama() {
14        System.out.println("Sei como deve ficar uma cama arrumada...");
15    }
16
17    @Override
18    public void limparQuarto() {
19        System.out.println("Sei como deve ficar um quarto limpo...");
20    }
21
22    @Override
23    public void atender() {
24        System.out.println("Sei mais ou menos como atender...");
25    }
26
27    @Override
28    public void falarIngles() {
29        System.out.println("Sei falar inglês muito bem...");
30    }
31 }
```


PRÁTICA

■ Classe Principal

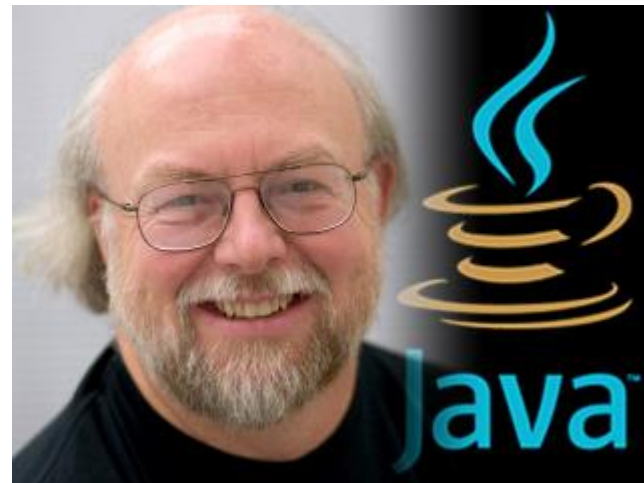
```
herculano - Principal.java
1 import model.Camareira;
2 import model.Cliente;
3 import model.Gerente;
4 import model.Recepcionista;
5
6 public class Principal {
7
8     public static void main(String[] args) {
9         System.out.println("Hotel Java\n");
10
11         Cliente cliente = new Cliente("Cliente", "(11) 2222-3333", "111.111.111-11");
12         Recepcionista recepcionista = new Recepcionista("Recepcionista", "(22) 1234-5678", "222.222.222-22");
13         Camareira camareira = new Camareira("Camareira", "(33) 4321-8765", "333.333.333-33");
14         Gerente gerente = new Gerente("Gerente", "(44) 6666-6666", "444.444.444-44");
15
16         System.out.println("Recepcionista...");
17         recepcionista.atender();
18         recepcionista.falarIngles();
19         System.out.println();
20
21         System.out.println("Camareira...");
22         camareira.arrumarACama();
23         camareira.limparQuarto();
24         System.out.println();
25
26         System.out.println("Gerente...");
27         gerente.arrumarACama();
28         gerente.falarIngles();
29         gerente.atender();
30         gerente.limparQuarto();
31     }
32
33 }
```

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

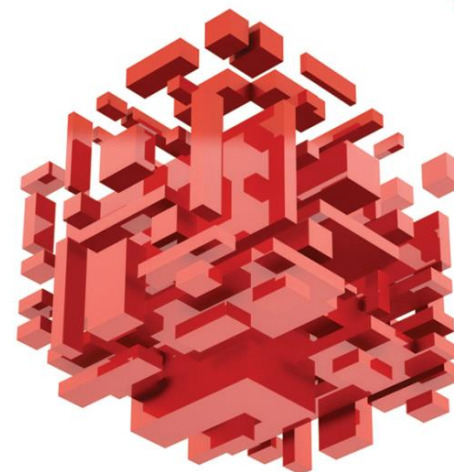
INTERFACE OU CLASSE ABSTRATA?

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.

Design Patterns com Java

Projeto orientado a objetos guiado por padrões



 Casa do
Código

EDUARDO GUERRA

BOAS PRÁTICAS DE POO

- É desejável um modelo com 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

CDI

Integre as dependências e contextos do seu código Java

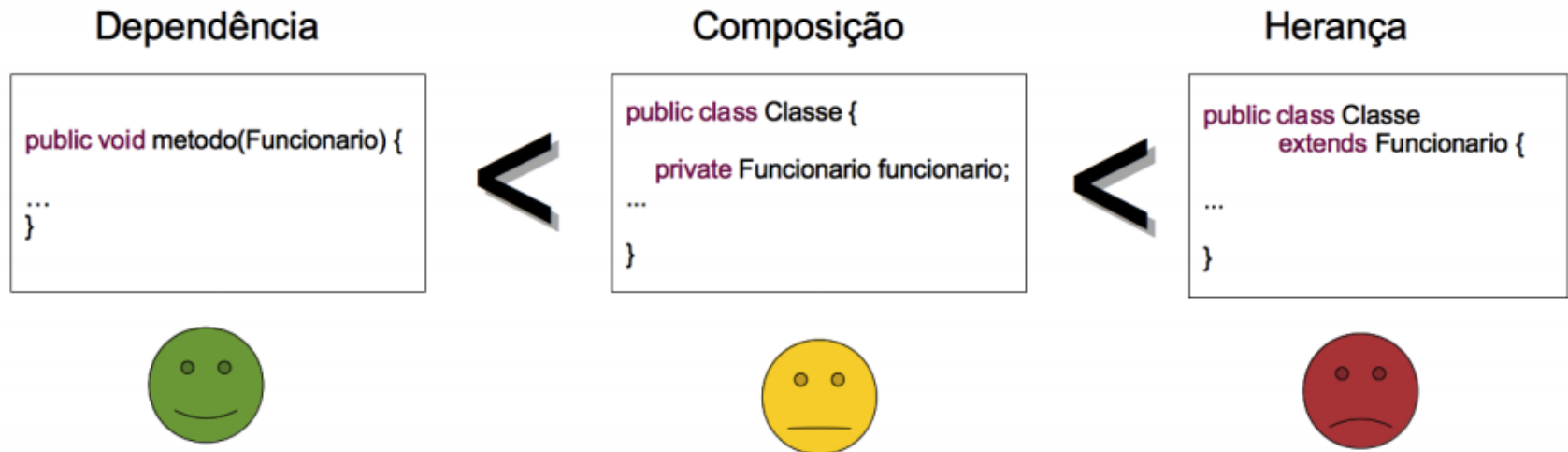


Casa do Código

GILLIARD CORDEIRO

BOAS PRÁTICAS DE POO

- É desejável um modelo com alta coesão e baixo acoplamento
- Acoplamento: Capacidade de poder se relacionar com outros objetos
 - 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*:

- ICozinha
- 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.