UNIVERSIDADE FEDERAL DO CEARÁ

SEMANA 11

PROGRAMAÇÃO ORIENTADA A OBJETOS

Prof. Marcos Vinicius de Andrade Lima
E-mail: marcos.vinicius@ufc.br

Olá!

Sou Marcos Vinicius

No tópico passado nós aprendemos a reescrever comportamentos (métodos) em Java...

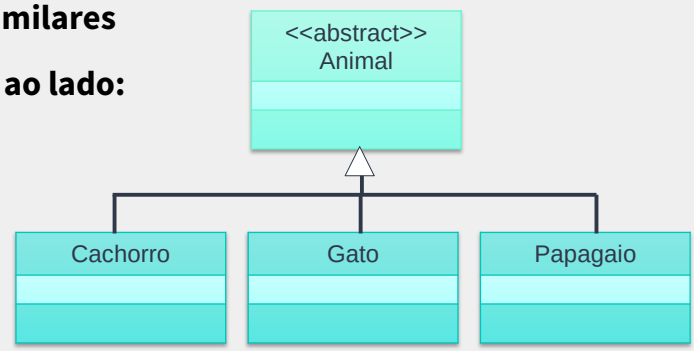
Neste tópico aprenderemos como **tudo sobre classes abstratas e interfaces**!

Os dias prósperos não vêm por acaso; nascem de muita fadiga e persistência
(Henry Ford)

Classes Abstratas

INTRODUÇÃO

- No projeto de uma hierarquia de classes, **nem todas as classes são previstas de serem instanciadas**
- **Algumas classes existem** simplesmente **para agrupar comportamentos similares**
- **Observe o exemplo ao lado:**



CLASSES ABSTRATAS

- Você pode até dizer que cria um animal em casa, mas seu bichinho de estimação com certeza nascerá de alguma espécie, que pode ser Cachorro, Gato, Papagaio, ou outro qualquer, mas nunca da classe Animal
- Essa é a ideia principal por trás do conceito de Classes Abstratas
- **Se a classe abstrata (Animal) não existisse, as demais classes (Cachorro, Gato, Papagaio, etc.) continuariam a existir, mas não haveria nenhuma relação definida entre elas**

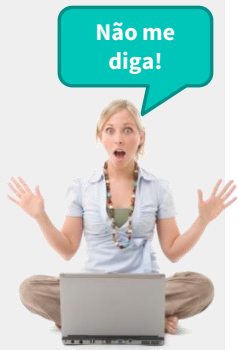
Classes Abstratas

- A estratégia de usar classes abstratas é ter uma superclasse que dita o comportamento, enquanto as subclasses concretas têm a liberdade de prover a implementação que acharem relevante.
- Onde se pode usar uma referência da superclasse Animal, mesmo sendo abstrata, objetos de quaisquer subclasses (Cachorro, Gato e Papagaio) podem ser usados.
- Classes Abstratas são úteis no reuso, incrementando o mecanismo de polimorfismo.

Classes Abstratas

- Uma classe abstrata usa o modificador **abstract**.
- **Exemplos:**

```
abstract class Animal{ ... }
abstract class Transporte { ... }
```



Métodos Abstratos

- Métodos Abstratos são aqueles que não têm implementação, somente o protótipo é fornecido.
- Uma classe que possui um método abstrato não pode ser instanciada, pois ela é parcialmente definida.
- As subclasses de uma classe que possui métodos abstratos devem implementar os métodos abstratos herdados da superclasse, caso contrário, também não poderão ser instanciadas.

Métodos Abstratos

- Um método abstrato usa o modificador **abstract** e possui apenas sua assinatura seguida de “;” .
- **Exemplos:**

```
abstract public void comer();
abstract int getNumeroDePneus();
```

Propriedades das Classes Abstratas

1. Um classe abstrata **não** pode ser instanciada
2. É especificada com a palavra-chave **abstract**
3. Uma classe que possui algum método abstrato **deve** ser declarada abstrata
4. A subclasse de uma classe abstrata **deve** prover a implementação dos métodos abstratos herdados, caso contrário ela também será uma abstrata

5. Uma classe pode ser declarada abstrata mesmo que não possua métodos abstratos. Isso evita que se instanciem objetos dessa classe

Um Pequeno Exemplo de Classe Abstrata

```
abstract class Animal {
  abstract public void comer();
  abstract public void dormir();
  abstract public void moverSe();
}

class Cachorro extends Animal {
  public void comer() { ... }
  public void dormir() { ... }
  public void moverSe() { ... }
}
```

Implementação dos Métodos

Hey, Presta Atenção!

- Classes Abstratas podem herdar de classes abstratas e não abstratas
- Alguns membros não podem ser declarados como abstratos:
  Construtores
  Métodos estáticos
  Métodos privados

- Para uma classe que herda de uma classe abstrata:
  1. a subclasse deve implementar todos os métodos abstratos
  2. a subclasse deve continuar sendo abstrata

Vamos Pensar um Pouco...

```
abstract class Transporte {
  abstract public int getNumeroDePneus();
  public void deslocar(double distancia)
  { ... }
}

class Carro extends Transporte {
  public int getNumeroDePneus()
  { return 4; }
}

class Cliente {
  Carro c = new Carro();
  Transporte v = new Transporte(); ERRO COMP!
  Transporte c2 = new Carro();     OK!
}
```

Vamos Pensar Mais um Pouco...

```
abstract public class ClassTest {
  private int number;

  public void test2(){
    number = 2;
    System.out.println(number);
  }

  public void test1(){
    number = 1;
    System.out.println(number);
    test2();
  }
}
```

OK!

OK!

Vamos Pensar Só Mais um Pouco...

```
abstract public class ClassTest {
  private int number;

  abstract public void test2();

  public void test1(){
    number = 1;
    System.out.println(number);
    test2();
  }
}
```

OK!

OK!

Vamos Pensar Só Mais um Pouco para Finalizar...

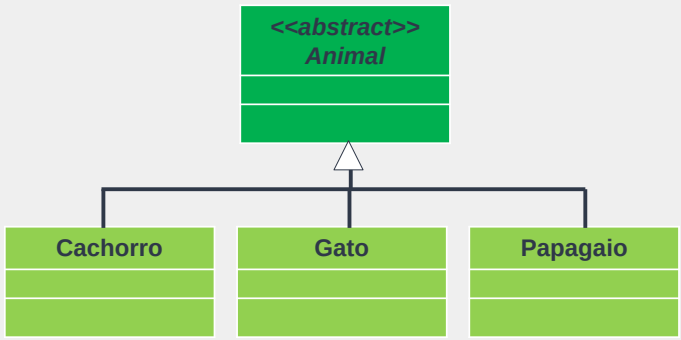
```
public class ClassTest2 extends ClassTest {
  public void test3(){
    teste1();
    teste2();
  }
}
```

ERRO COMP!

Interfaces

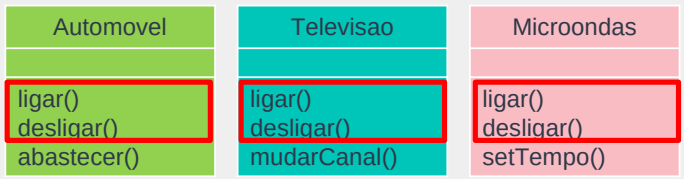
INTRODUÇÃO

- As classes **Cachorro, Gato e Papagaio** são, por essência, **animais**, por isso, podem ser agrupadas debaixo da superclasse Animal



INTRODUÇÃO

- Mas imagine que você tenha as classes **Automóvel**, **Televisão** e **Microondas**:



- **O que elas têm em comum?**

- **IMPORTANTE:** embora elas tenham os mesmos nomes, suas implementações serão completamente diferentes nas três classes!

O cenário apresentado é adequado ao uso de Interfaces!

Interface

- É um tipo especial que declara apenas assinaturas de operações que podem ser comuns a várias classes
- Uma interface define um **contrato**, especificando somente os protótipos dos métodos (assinaturas), mas não suas implementações
- Analisando sob essa ótica, uma interface é um **protótipo** de uma classe, mas não é uma classe!
- Para o nosso exemplo, podemos ter uma interface chamada Operações que declara os métodos **ligar** e **desligar**

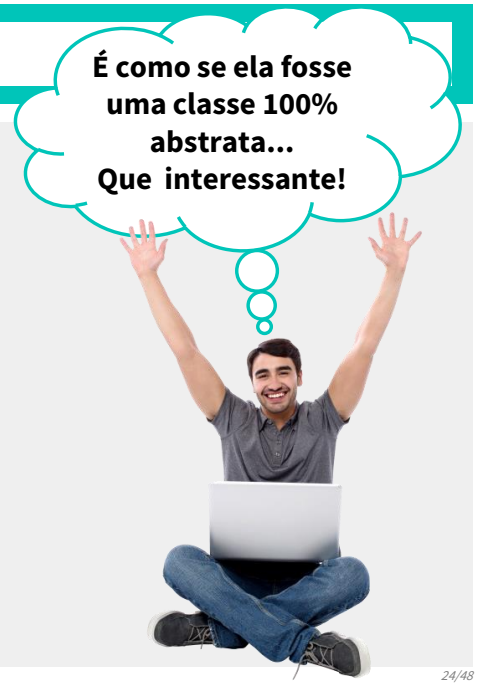
INTERFACE

- Uma interface é declarada utilizando-se a palavra-chave **interface**
- Assim como uma classe, uma interface pode ser pública (public) e não-pública, tendo, neste último caso, visibilidade restrita ao pacote onde foi definida

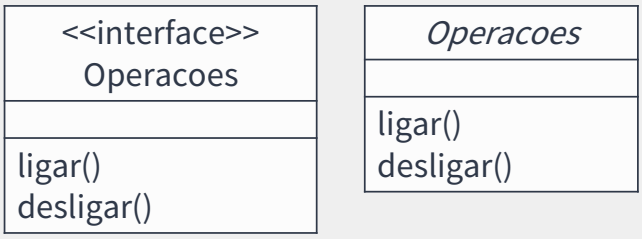
```
public interface Operacoes {
   void ligar();
   void desligar();
}
```


PAY ATTENTION!

- As operações de uma interface são por default:
  - ✓ **abstract**
  - ✓ **public**
- Embora não seja necessário declará-los como tal!

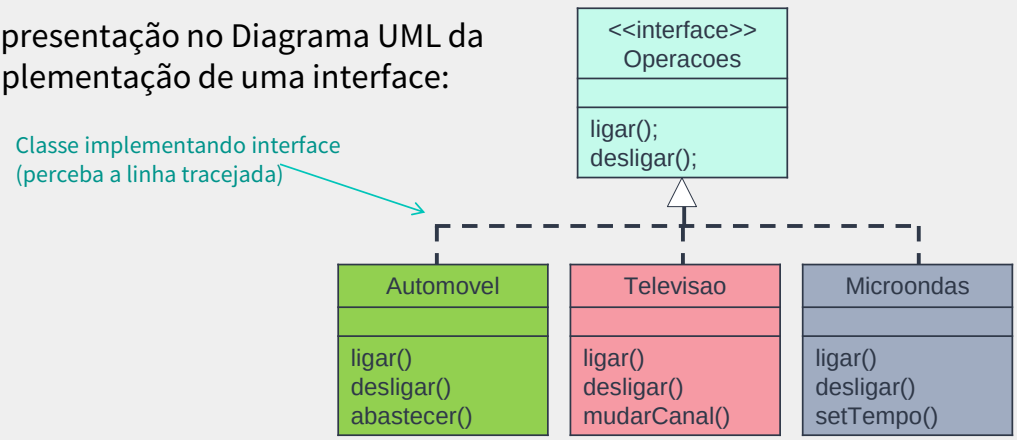


Representação na UML

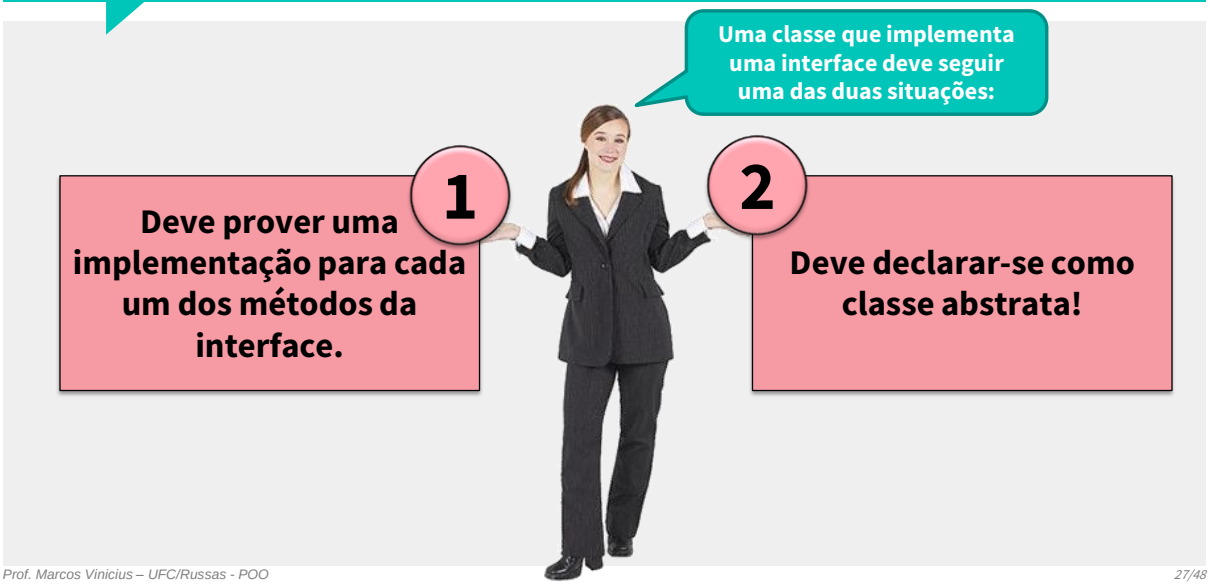


Olhando Mais de Perto...

- As implementações dos métodos ficam a cargo das classes que irão implementar essa interface.
- Representação no Diagrama UML da implementação de uma interface:



Só Sei Que Foi Assim...



Exemplo na Situação 1

```
public class Automovel implements Operacoes {

  public void ligar() {
    girarChave();
    acionarCombustao();
    iniciarFuncionamentoDoMotor();
  }

  public void desligar() {
    girarChaveAoContrario();
    desativarFuncionamentoDoMotor();
  }
}
```

Exemplo na Situação 2

```
public abstract class Televisao implements Operacoes{

  public void desligar() {
    pressionarButao();
    apagarImagem();
  }
}

public class TelevisaoPlana extends Televisao {
  public void ligar() {
    pressionarButao();
    capturarSinal();
    jogarImagemNaTela();
 }
}
```


Hierarquia de Classes e Interfaces

- Uma classe pode implementar zero ou mais interfaces, listadas após a cláusula implements

  public class A implements Inter1, Inter2, Inter3, Inter4 { ... }

- Uma classe pode ao mesmo tempo estender de outra classe e implementar zero ou mais interfaces

- A cláusula extends **sempre** vem antes da implements!

  public class A extends B implements Inter1, Inter2, Inter3 { ... }

Hierarquia de Classes e Interfaces

- Uma interface pode estender de outras interfaces (**isso mesmo, no plural**) usando a cláusula **extends**.

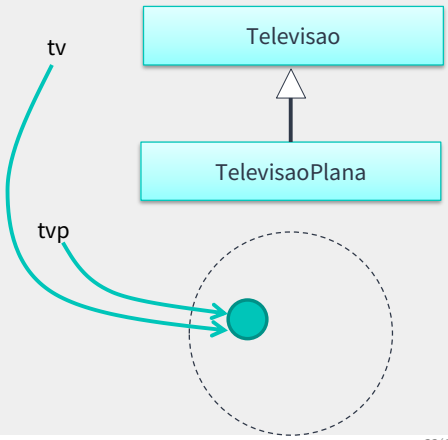
```
public interface Inter1 extends Inter2, Inter3,
                 Inter4 { ... }
```

Se Liga!

- Uma interface **NÃO** pode ser instanciada
  Eu já falei que ela **NÃO** é uma classe?
- Os **métodos** de uma classe que implementam as operações de uma interface são sempre **públicos**
- Interfaces **NÃO** possuem construtores
  Lembra que elas **NÃO** são classes?

INTERFACES E O POLIMORFISMO

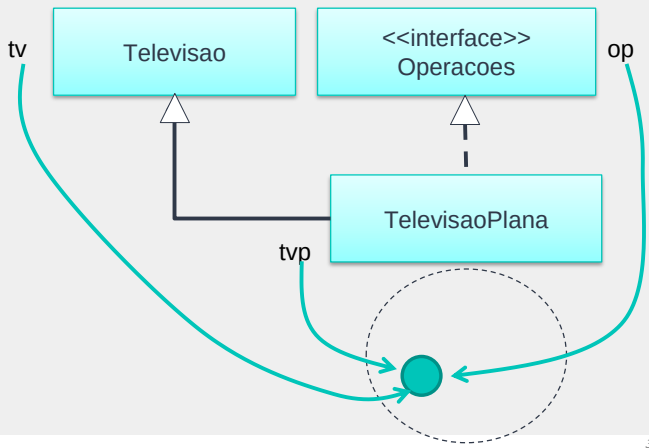
- Uma referência de objeto de uma superclasse pode **polimorficamente** referenciar objetos do tipo da própria superclasse ou de qualquer subclasse
- Como vimos, esta relação de herança em Java é linear, não permitindo herança múltipla de implementação (classes)




```
TelevisaoPlana tvp =
new TelevisaoPlana();
Televisao tv = tvp;
```


INTERFACES E O POLIMORFISMO

- As interfaces que uma classe implementa e as classes que ela estende, direta ou indiretamente, são chamadas supertipos daquela classe
- Referências de supertipos podem referenciar objetos de subtipos




```
TelevisaoPlana tvp =
new TelevisaoPlana();
Televisao tv = tvp;
Operacoes op = tvp;
```

Relações de Herança

- Em Java, existem três diferentes relações de herança, a saber:
  - ✓ **Hierarquia de herança linear de implementação entre classes:** uma classe só pode estender uma outra classe
  - ✓ **Hierarquia de herança múltipla entre interfaces:** uma interface pode estender várias outras interfaces
  - ✓ **Hierarquia de herança múltipla entre interfaces e classes:** uma classe pode implementar várias interfaces

Não se Esqueça!

Classes Abstratas	Interfaces
– Agrupa objetos com implementações semelhantes; – Define novas classes através de herança; – Só podem ter como supertipo uma outra classe. – Mais rígido.	– Agrupa objetos com implementações diferentes; – Define novas interfaces através de múltiplas implementações; – Várias podem ser supertipo do mesmo tipo. – Mais flexível.

Um Pequeno Exemplo...

```
import java.time.*;

public interface HoraData {

 void setHora(int hora, int minuto, int segundo);
 void setData(int dia, int mes, int ano);
 void setDataHora(int dia, int mes, int ano,
                  int hora, int minuto, int
segundo);
 LocalDateTime getLocalDataHora();

}
```


O que uma **classe** deve fazer para ser **HoraData**?

Implementando HoraData...

```
import java.time.*;
import java.lang.*;
import java.util.*;

public class HoraDataCliente implements HoraData {

    private LocalDateTime dataHora;

    public HoraDataCliente() {
        dataHora = LocalDateTime.now();
    }

    public void setHora(int hora, int minuto, int segundo) {
        LocalDate dataAtual = LocalDate.from(dataHora);
        LocalTime configuraHora = LocalTime.of(hora, minuto, segundo);
        dataHora = LocalDateTime.of(dataAtual, configuraHora);
    }
```


Implementando HoraData...

```
public void setData(int dia, int mes, int ano) {
        LocalDate configuraData = LocalDate.of(dia, mes, ano);
        LocalTime horaAtual = LocalTime.from(dataHora);
        dataHora = LocalDateTime.of(confiruaData, horaAtual);
    }

    public void setDataHora(int dia, int mes, int ano,
                            int hora, int minuto, int segundo) {
        LocalDate configuraData = LocalDate.of(dia, mes, ano);
        LocalTime configuraHora = LocalTime.of(hora, minuto, segundo);
        dataHora = LocalDateTime.of(configuraData, configuraHora);
    }

    public LocalDateTime getHoraDataLocal() {
        return dataHora;
    }
```

Implementando HoraData...

```
public String toString() {
        return dataHora.toString();
    }

    public static void main(String[] args) {
        HoraData meuHoraDataCliente = new HoraDataCliente();
        System.out.println(meuHoraDataCliente.toString());
    }
}
```


Agora Imagine o Seguinte Cenário...

- Você teve a ideia de adicionar à interface uma funcionalidade para adicionar uma zona de tempo

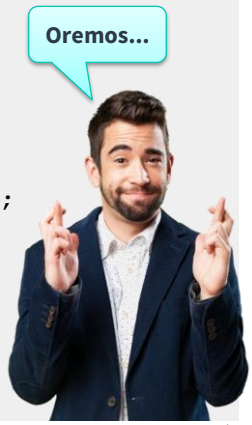
```
import java.time.*;

public interface HoraData {

 void setHora(int hora, int minuto, int segundo);
 void setData(int dia, int mes, int ano);
 void setDataHora(int dia, int mes, int ano,
                  int hora, int minuto, int segundo);
 LocalDateTime getLocalDataHora();

 ZonedDateTime getZonaDataHora(String zona);

}
```



O Código Vai Quebrar na Classe Cliente!

E Agora? Quem Poderá nos Defender?

- No Java 8 foi introduzida a possibilidade das interfaces trazerem um código **padrão**
- Esse código padrão transforma uma **operação** em **método padrão** para uma dada assinatura
- E também agora podemos definir **métodos estáticos** na interface. Loucura, loucura! Mas loucura boa! rsrsrs

Alterando Nossa Interface

```
...

  static ZoneId getZonaId (String zone) {
    return ZoneId.of(zone);
  }

  default ZonedDateTime getZonaDataHora(String zona) {
    return ZonedDateTime.of(getHoraDataLocal(), getZonaId(zona));
  }
}
```

Obrigado!

Mais alguma dúvida?

Acesse o **AME** para mais informações e treinamento do **NERDS**!

http://ame2.russas.ufc.br