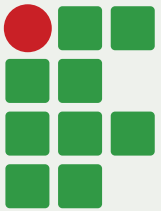
Classes Abstratas, Interface e Polimorfismo

Linguagem de Programação I

Prof. Fábio José Rodrigues Pinheiro

DEZEMBRO DE 2021



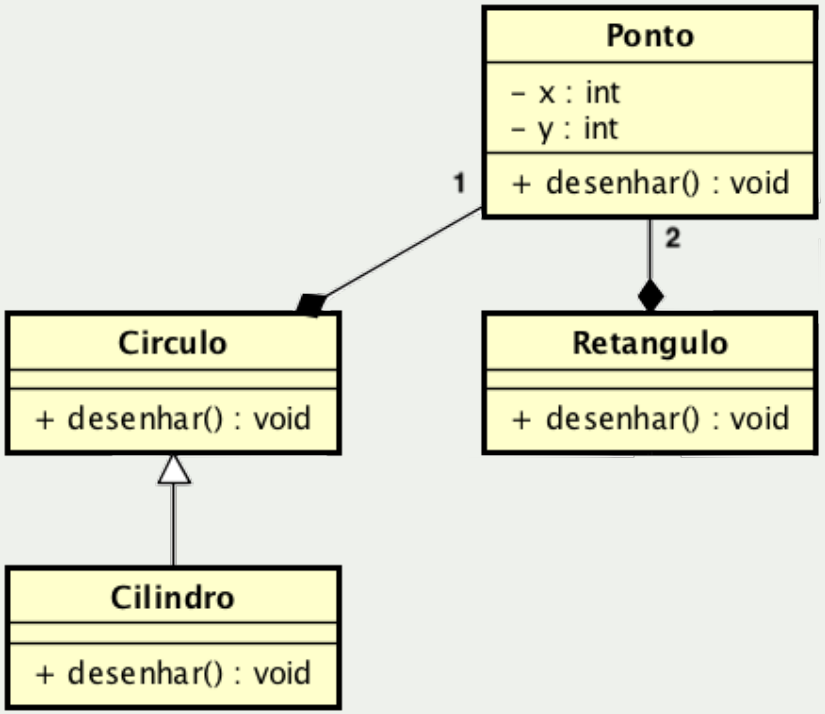
INSTITUTO FEDERAL
Catarinense
Campus Videira

Exemplo: Aplicativo para desenho vetorial

- Formas geométricas: Ponto, Círculo, Cilindro e Retângulo
- Todas as classes deverão ter obrigatoriamente um método **desenhar**, que uma vez invocado, fará com que a forma seja desenhada.

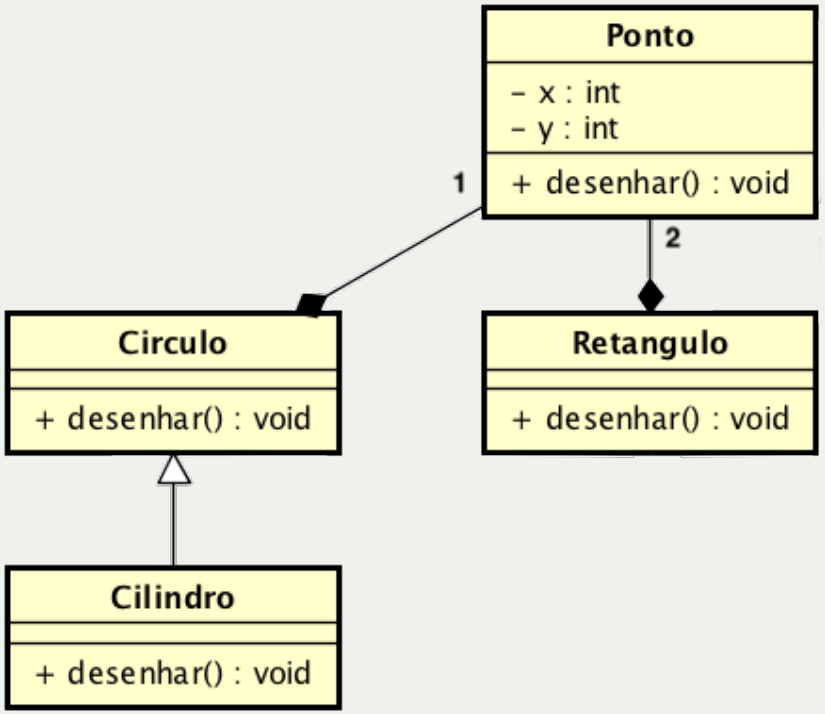
Exemplo: Aplicativo para desenho vetorial

- Formas geométricas: Ponto, Círculo, Cilindro e Retângulo
- Todas as classes deverão ter obrigatoriamente um método **desenhar**, que uma vez invocado, fará com que a forma seja desenhada.



Exemplo: Aplicativo para desenho vetorial

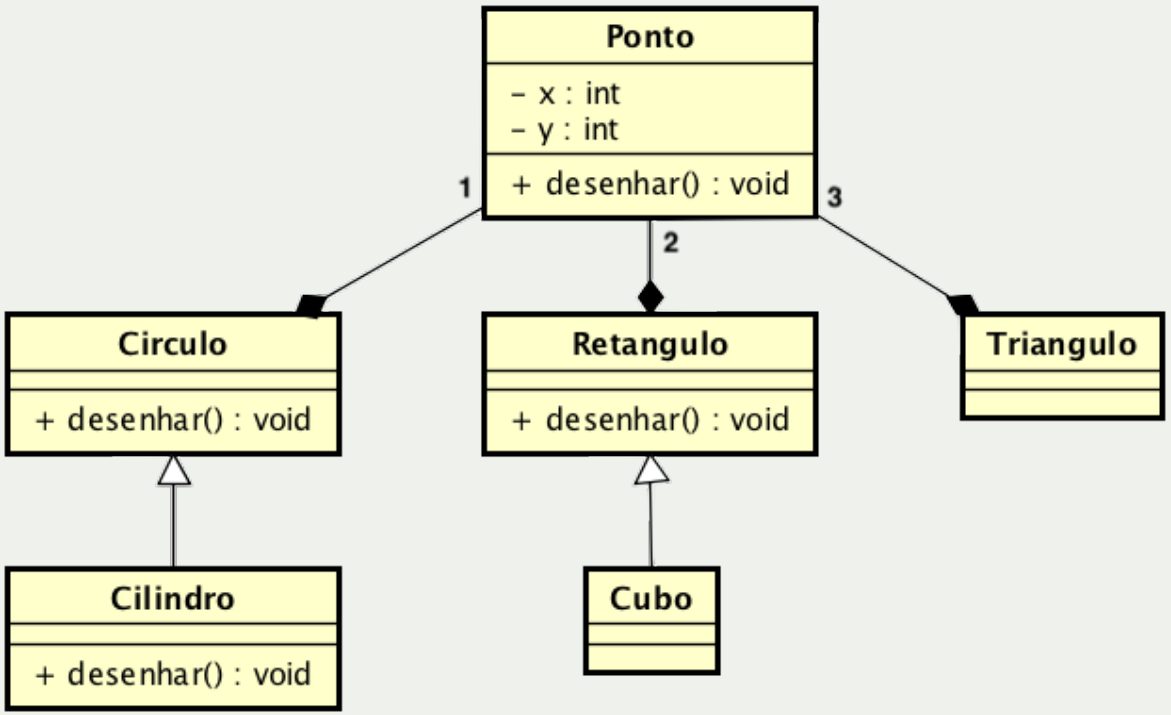
- Formas geométricas: Ponto, Círculo, Cilindro e Retângulo
- Todas as classes deverão ter obrigatoriamente um método **desenhar**, que uma vez invocado, fará com que a forma seja desenhada.



- **Nova necessidade:** Classes Triangulo e Cubo

Exemplo: Aplicativo para desenho vetorial

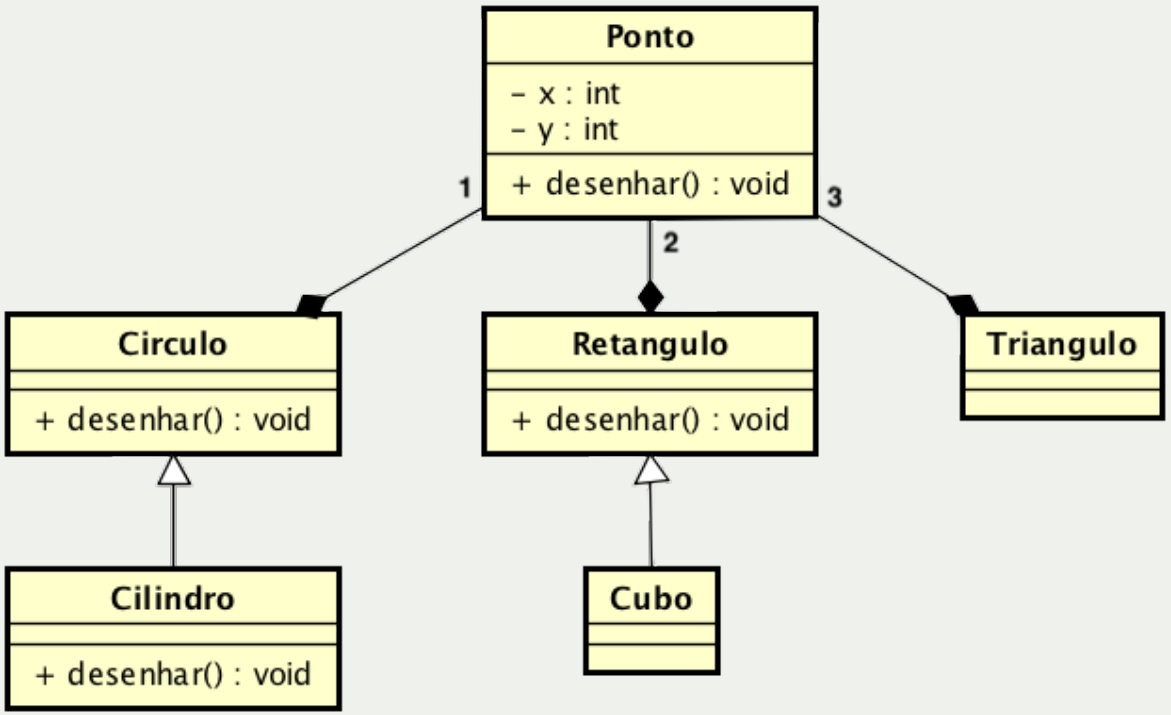
- Formas geométricas: Ponto, Círculo, Cilindro e Retângulo
- Todas as classes deverão ter obrigatoriamente um método **desenhar**, que uma vez invocado, fará com que a forma seja desenhada.



- **Nova necessidade:** Classes Triangulo e Cubo

Exemplo: Aplicativo para desenho vetorial

- Formas geométricas: Ponto, Círculo, Cilindro e Retângulo
- Todas as classes deverão ter obrigatoriamente um método **desenhar**, que uma vez invocado, fará com que a forma seja desenhada.

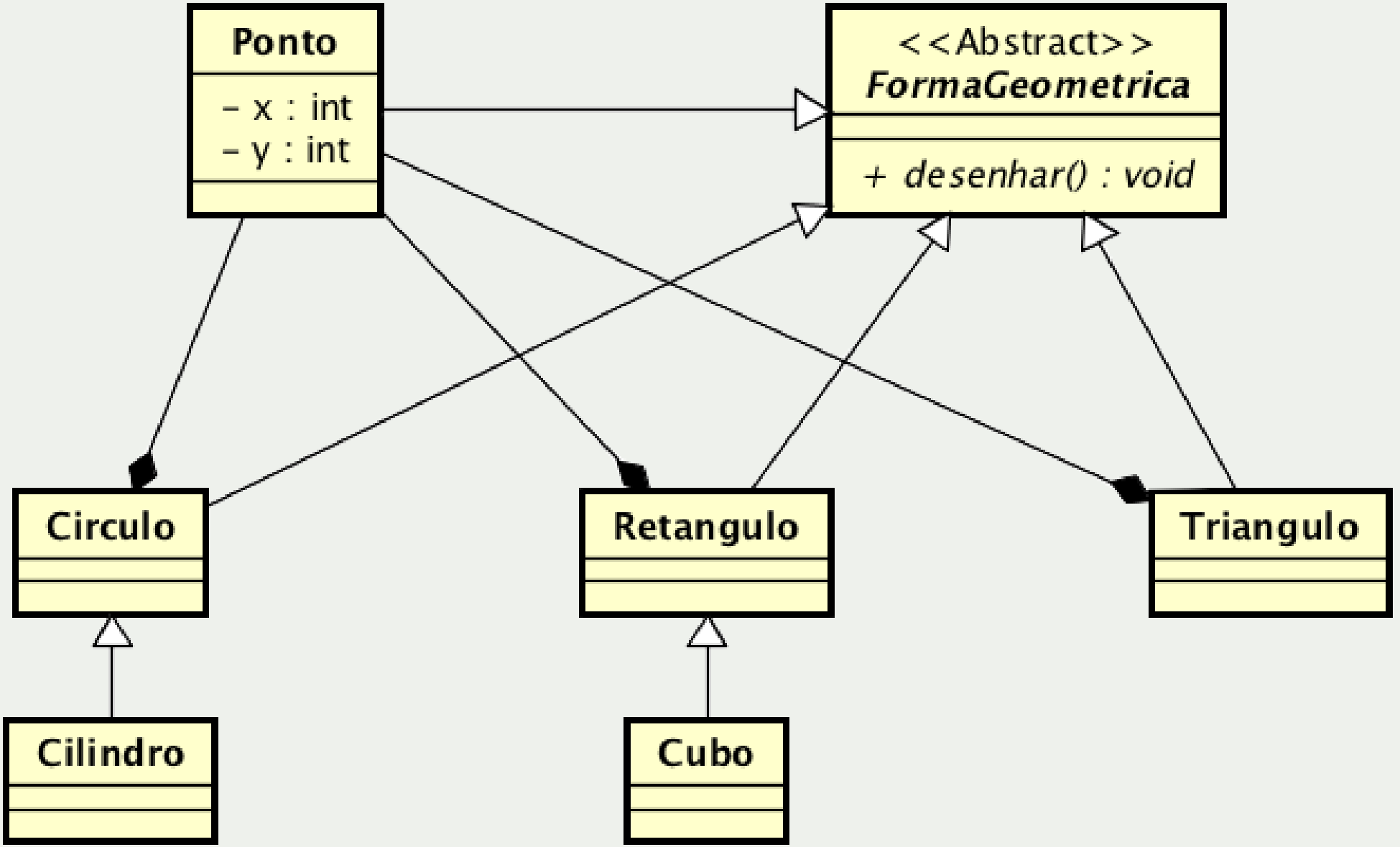


- **Nova necessidade:** Classes Triangulo e Cubo
- Como garantir que as novas classes **terão obrigatoria-mente o método desenhar**?

Classe abstrata

- Classes abstratas são criadas para serem herdadas, servindo de "modelo"para as classes filhas
- Pode conter métodos concretos e métodos abstratos
  - **Métodos concretos** possuem implementação
  - **Métodos abstratos** não possuem implementação
- Não é possível instanciar objetos de uma classe abstrata
- Todo **método abstrato deve ser obrigatoriamente sobrescrito** pelas subclasses, métodos concretos não precisam ser sobrescritos
- Uma subclasse que não prover implementações para os métodos abstratos herdados, deve obrigatoriamente ser abstrata

Exemplo: Aplicativo para desenho vetorial



Exemplo: Aplicativo para desenho vetorial

```java
public abstract class FormaGeometrica{
  public abstract void desenhar();
}
```

```java
public class Ponto extends FormaGeometrica{
  private int x;
  private int y;

  @Override
  public void desenhar(){
    System.out.println("Desenhando ponto: " + x +"," + y);
  }
}
```


Exemplo: Classe abstrata Personagem

```java
public abstract class Personagem{
  private int id;
  private String nome;

  public Personagem(int i, String n){
    this.id = i;
    this.nome = n;
  }

  public String obterNome(){
    return this.nome;
  }

  public void imprimirDados(){
    System.out.println("Id:" + this.id + ", Nome: " + this.nome);
  }

  public abstract void atacar(float intensidade);
}
```


Exemplo: Classe concreta Arqueiro

```
1  public class Arqueiro extends Personagem{
2    private int habilidade;
3
4    public Arqueiro(int i, String n, int h){
5      super(i,n);
6      this.habilidade = h;
7    }
8
9    public void imprimirDados(){
10     super.imprimirDados();
11     System.out.println("Habilidade: " + this.habilidade);
12   }
13
14   @Override
15   public void atacar(float intensidade){
16     System.out.println("Disparando flechas com a intensidade: " +
       intensidade);
17   }
18 }
```


Interface

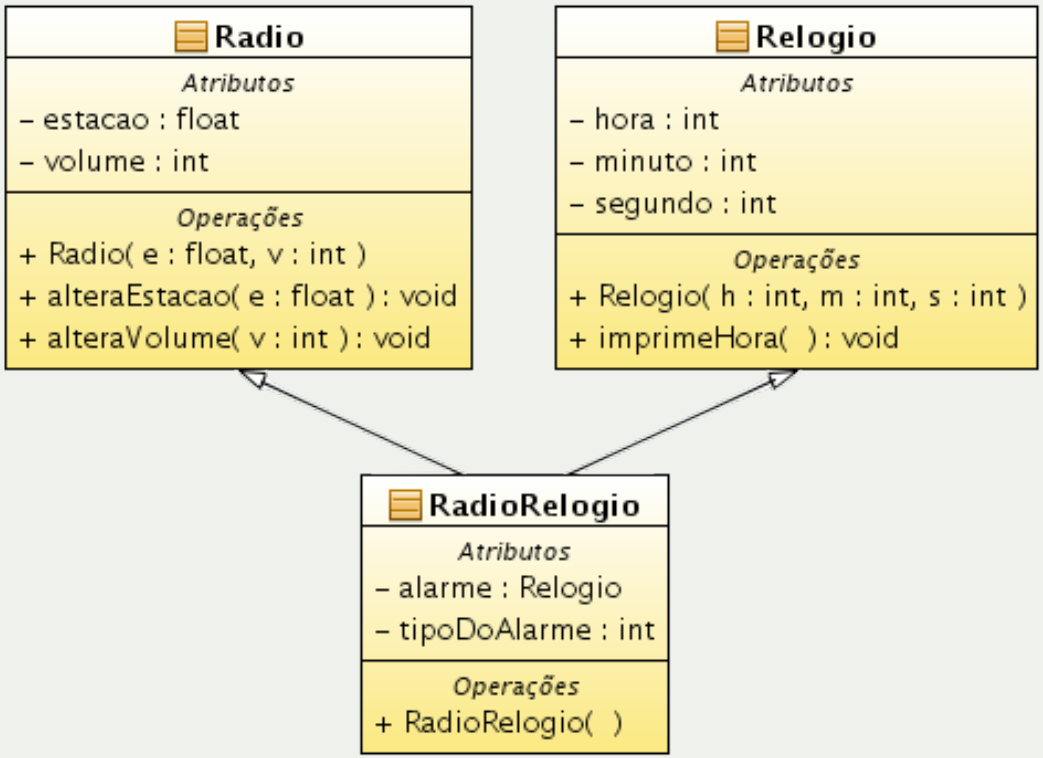
Herança múltipla

- No desenvolvimento de softwares complexos poderemos nos deparar com situações onde uma **nova classe** possui **características semelhantes** com **duas** ou **mais classes** existentes
- A linguagem C++ possui o conceito de **herança múltipla** permitindo que uma classe seja derivada de várias classes base

Herança múltipla

- No desenvolvimento de softwares complexos poderemos nos deparar com situações onde uma **nova classe** possui **características semelhantes** com **duas** ou **mais classes** existentes

- A linguagem C++ possui o conceito de **herança múltipla** permitindo que uma classe seja derivada de várias classes base



Herança múltipla em Java

- Java **não permite** que uma **classe seja derivada** de mais de uma outra classe
  - Para evitar as complicações relacionadas a **herança múltipla de estados** – habilidade de herdar atributos de múltiplas classes

Herança múltipla em Java

- Java **não permite** que uma **classe seja derivada** de mais de uma outra classe
  - Para evitar as complicações relacionadas a **herança múltipla de estados** – habilidade de herdar atributos de múltiplas classes
- O conceito de herança múltipla pode ser obtido em Java fazendo uso de **Interfaces**
  - **herança múltipla de tipos** – uma classe pode implementar mais de uma interface
  - **herança múltipla de implementação** – habilidade de herdar as definições de métodos de múltiplas interfaces
  - Java 8 introduziu o conceito de métodos `default` para Interfaces, que permite que métodos em uma Interface tenham implementação

Interface – Exemplo de uso

Jogo de corrida

Um fabricante de jogo de corrida gostaria de **permitir que seu jogo fosse estendido por outras pessoas** de forma que **possam criar seus próprios carros**. Contudo, **deve-se garantir que todos os carros possuam os mesmos métodos** (p. ex. frear, acelerar, etc)

Interface – Exemplo de uso

Jogo de corrida

Um fabricante de jogo de corrida gostaria de **permitir que seu jogo fosse estendido por outras pessoas** de forma que **possam criar seus próprios carros**. Contudo, **deve-se garantir que todos os carros possuam os mesmos métodos** (p. ex. frear, acelerar, etc)

- Interfaces podem ser vistas como **contratos** que devem ser respeitados
- Possibilita que códigos desenvolvidos por um time possam interagir com os códigos desenvolvidos pelo outro time
- Ambos os times não precisam ter conhecimento sobre o código que está escrito pelo outro time

Interface

Uma Interface é Java é semelhante a uma classe abstrata, porém só pode conter constantes, métodos abstratos e métodos default

- Por padrão todos atributos são `public`, `static` e `final` e todos os métodos são `public`
- Uma **Interface não pode ser instanciada** e o principal objetivo é servir como gabarito e ser implementada por classes
- A partir do Java8 **somente métodos estáticos ou default** poderão conter implementação

Exemplo: Interface Carro

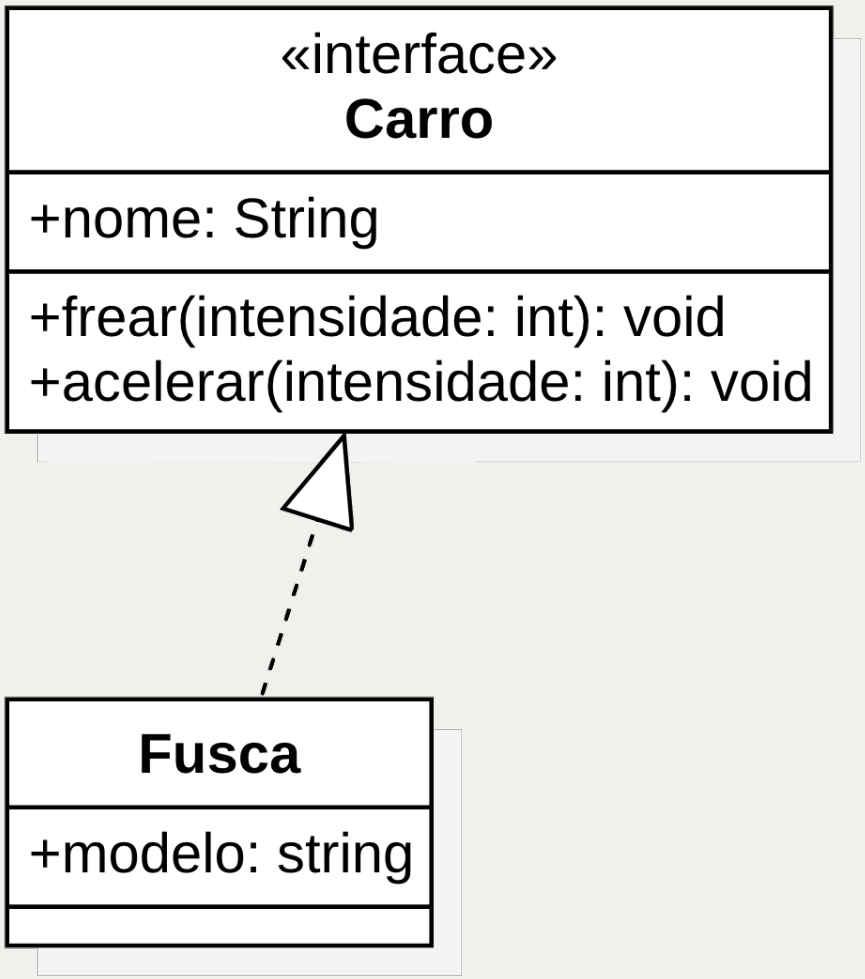
```
34 public interface Carro{
35   public static final String nome = "Carro";
36
37   void frear(int intensidade);
38
39   default void desligar() {
40     System.out.println("Desligando carro.");
41   }
42 }
```

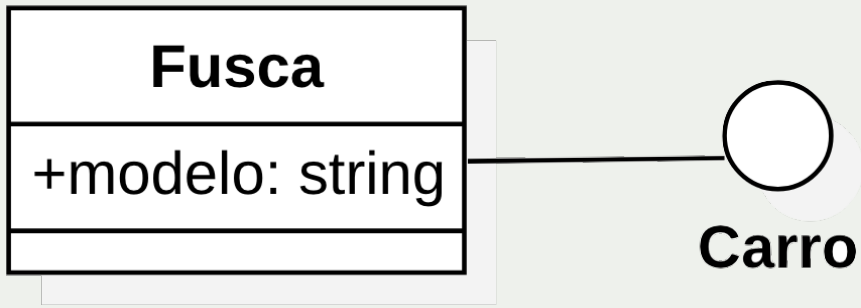

Exemplo: Interface Carro

```
34 public interface Carro{
35   public static final String nome = "Carro";
36
37   void frear(int intensidade);
38
39   default void desligar() {
40     System.out.println("Desligando carro.");
41   }
42 }
```

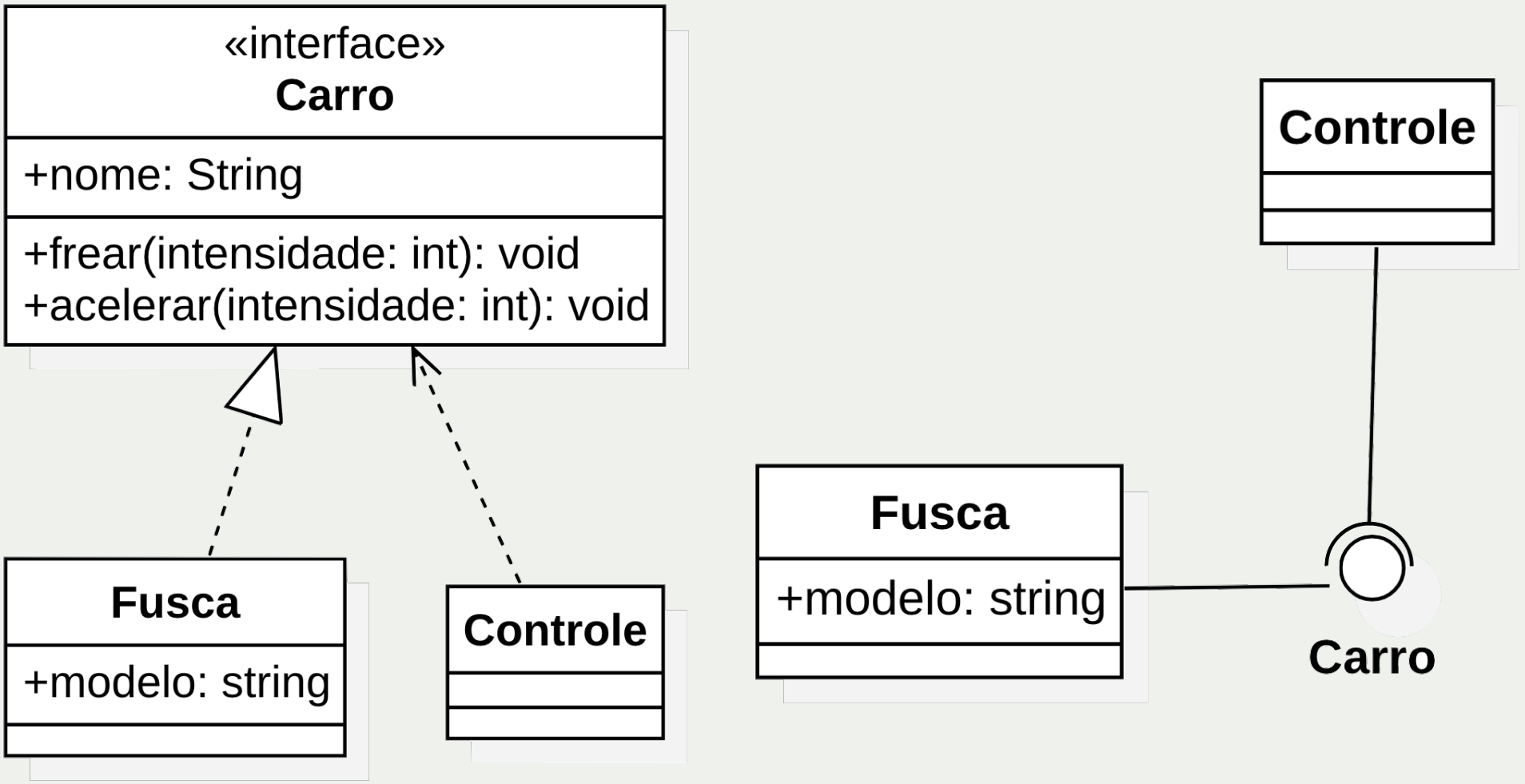
```
40 public class Fusca implements Carro{
41
42   public void frear(int intensidade){
43     System.out.println("Encostando a lona no tambor de freio");
44   }
45 }
```


Representação de interface em UML: diferentes formas





Representação de interface em UML: diferentes formas



Exemplo: Herança múltipla para obtermos um Triatleta

- Corredor pode correr
- Ciclista pode pedalar
- Nadador pode nadar

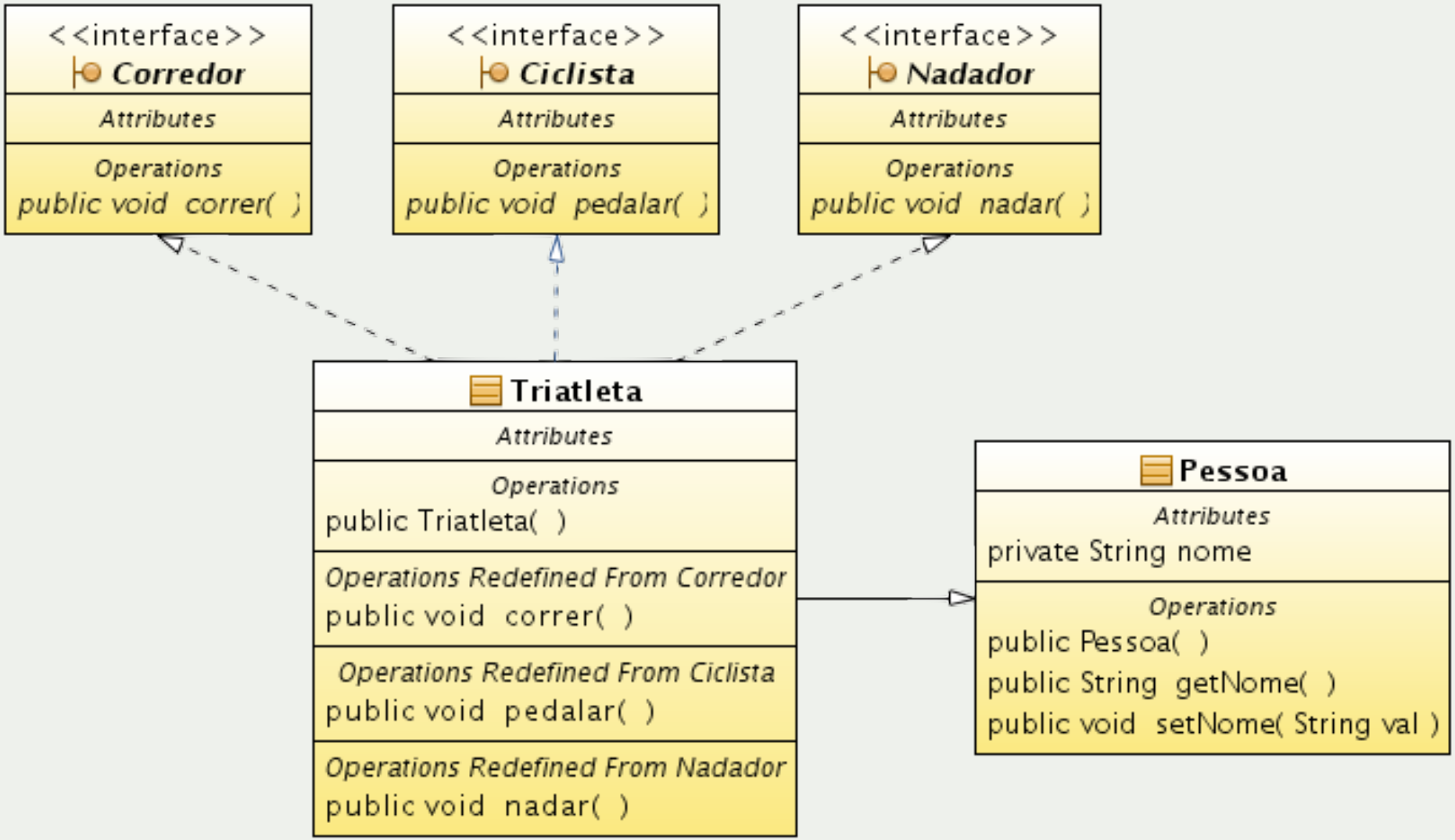
Exemplo: Herança múltipla para obtermos um Triatleta

- Corredor pode correr
- Ciclista pode pedalar
- Nadador pode nadar

Desenhe um diagrama de classes UML

1. Uma classe para representar cada modalidade de atleta. Todos devem possuir um nome e um CPF
2. Uma classe para representar um Triatleta, que pode correr, pedalar e nadar. Este também possui um nome e CPF

Exemplo: Herança múltipla para obtermos um Triatleta



Polimorfismo

Polimorfismo: significado “muitas formas”

Permite desenvolver sistemas que sejam facilmente extensíveis de forma que novas classes possam ser adicionadas ao sistema exigindo pouca ou nenhuma modificação nas partes gerais do sistema

- **Novas classes devem obrigatoriamente fazer parte de uma hierarquia** de classes já existente no sistema

- Deve-se **programar pensando** somente nas **classes mais genéricas**, **não se preocupando** com as **classes mais específicas**
  - métodos existentes na superclasse também estarão presentes nas subclasses

Exemplo:Java of Empires

Existem três personagens: Aldeão, Arqueiro e Cavaleiro

Todos compartilham algum tipo de informação e comportamento, logo, todos herdam da classe Personagem

- Todo personagem possui um **iD** único no jogo e todo personagem poderá se **mover** pelo cenário
  - Aldeão por 1 unidade
  - Arqueiro por 2 unidades
  - Cavaleiro por 10 unidades

```
1 Aldeao    a = new Aldeao();
2 Arqueiro  b = new Arqueiro();
3 Cavaleiro c = new Cavaleiro();
4
5 // invocando o método mover de cada objeto
6 a.mover();
7 b.mover();
8 c.mover();
```


Exemplo: Jogo Java of Empires

- O exército pode ter até 300 personagens

```
55 Aldeao    vetA[] = new Aldeao[100];
56 Arqueiro  vetB[] = new Arqueiro[100];
57 Cavaleiro vetC[] = new Cavaleiro[100];
58
59 //omitindo a criação dos objetos
60
61 // invocando o método mover de cada objeto
62 for(int i = 0; i < 100; i++){
63   vetA[i].mover();
64   vetB[i].mover();
65   vetC[i].mover();
66 }
```


Exemplo: Jogo Java of Empires

- O exército pode ter até 300 personagens

```
55 Aldeao    vetA[] = new Aldeao[100];
56 Arqueiro  vetB[] = new Arqueiro[100];
57 Cavaleiro vetC[] = new Cavaleiro[100];
58
59 //omitindo a criação dos objetos
60
61 // invocando o método mover de cada objeto
62 for(int i = 0; i < 100; i++){
63   vetA[i].mover();
64   vetB[i].mover();
65   vetC[i].mover();
66 }
```

E se criarmos um novo personagem, Guerreiro?

Será necessário modificar o código dentro do laço de repetição

Exemplo: Jogo Java of Empires

- Com o **polimorfismo** é possível incluir novos personagens no jogo sem que seja preciso modificar boa parte do código
- Sempre programar para o "geral" e nunca para o específico.

```
1  // A lista da superclasse pode armazenar objetos das suas subclasses
2  private List<Personagem> personagens = new ArrayList<>();
3  Aldeao a = new Aldeao(1);
4  Arqueiro aq = new Arqueiro(2);
5  Cavaleiro c = new Cavaleiro(3);
6
7  personagens.add(a);
8  personagens.add(aq);
9  personagens.add(c);
10
11 // o método mover existe na superclasse. No tempo de execução são
       invocados os métodos de cada subclasse
12 for (Personagem p : personagens)
13   p.mover();
```


Exercícios

Gerar folha de pagamento de uma empresa

- Existem 4 carreiras na empresa
  - **mensal fixo** – valor fixo por mês independente do número de horas que trabalhou em um mês
  - **horista** – valor adicional pago por hora extra trabalhada além das 40 horas semanais. O valor da hora extra é acordado com cada funcionário
  - **comissionado** – salário calculado somente sobre o percentual das vendas que efetivou. O percentual das vendas é um valor acordado com cada funcionário
  - **comissionado efetivo** – valor fixo por mês mais adicional do percentual das vendas que efetivou

Gerar folha de pagamento de uma empresa

- Faça uma modelagem para representar os funcionários dessa empresa
- Faça uma rotina que permita gerar a folha de pagamento da empresa
- Faça uma rotina que permita aumentar em 10% o salário base de todos os funcionários da carreira **comissionado efetivo**
- **É necessário fazer uso de polimorfismo**