Programação Orientada a Objetos (PO24CP)

Aula #10 - Herança: classe abstrata, interface, e herança múltipla

Profª Luciene de Oliveira Marin
lucienemarin@utfpr.edu.br

Classe abstrata

Classes Abstratas

Por que usar classes abstratas?

- Às vezes você quer implementar parcialmente uma classe e delegar às subclasses o restante da implementação.
- Ao **subir na hierarquia de herança**, as classes se tornam mais **genéricas** a tal ponto que não representam algo tangível ou visível. Elas acabam se tornando **modelos para classes**.
  - Nesses casos, pode ser desejável **não permitir que objetos sejam instanciados** a partir dessas classes.
    - Por exemplo, **não faz sentido** que **objetos** sejam declarados a partir de classes como `Forma` ou `Pessoa`.

▷ Classes que não admitem objetos devem ser definidas como **abstratas**.

Classes Abstratas

- Em Java usa-se a palavra-chave **abstract** para indicar que uma classe é abstrata

```
public abstract class Forma {
  ...
}

public abstract class Pessoa {
...
}
```

Métodos Abstratos

- Um **método abstrato** é um método que **não apresenta implementação**, sendo responsabilidade das classes-derivadas a implementação do mesmo
  - Por exemplo, na classe **Forma** podemos declarar o método **desenhar()** como **abstrato**, indicando que as **classes-derivadas deverão implementá-lo** - não faz sentido esse método ser implementado em **Forma**
- Um método é declarado como abstrato usando-se a palavra-chave **abstract**.

Métodos Abstratos

```
public abstract class Forma {
   ...
   public abstract void desenhar();
          ...
}

public class Quadrado extends Forma{
    public void desenhar() {
          //implementação de desenhar um quadrado
          ...
    }
}
```

Exemplo de uma Classe Abstrata

```
abstract class Super {
  int x;

  int getX() { return x; }

  abstract void setX(int newX); // corpo vazio
}

class Sub extends Super {

  void setX(int newX) { x = newX; }
}
```

Classe Abstrata

Principal uso

Servir de base para concepção de subclasses

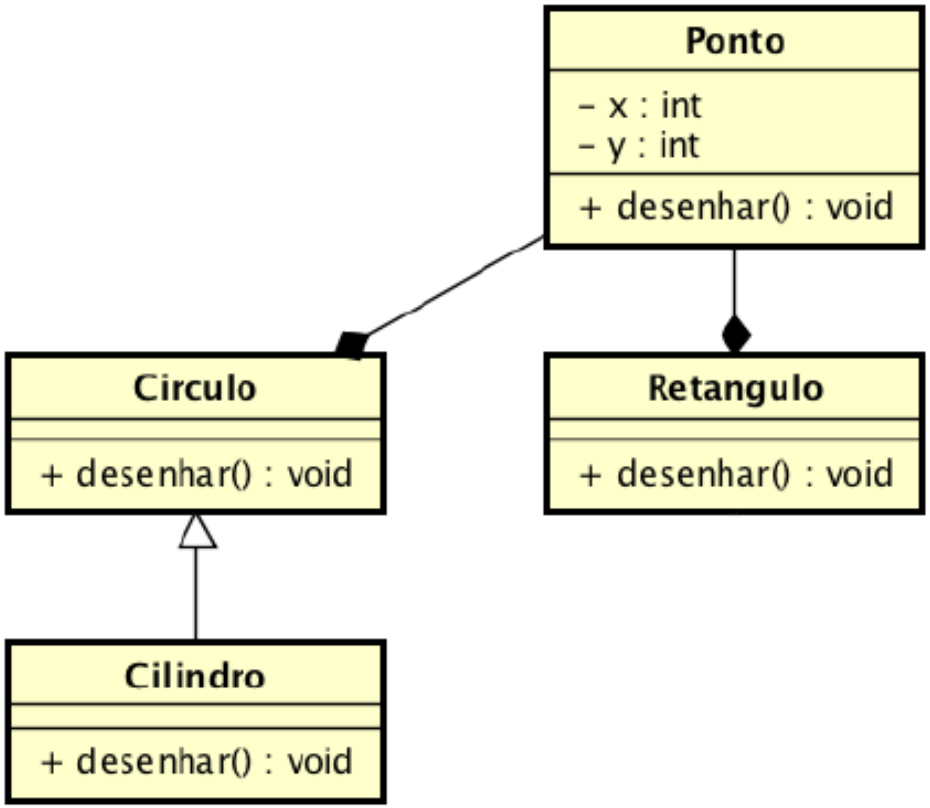
- **Não é possível instanciar objetos de uma classe abstrata**
  - As subclasses que não forem abstratas podem então possuir instâncias.
- **Pode conter métodos concretos e métodos abstratos**
- Todo método abstrato deve ser **obrigatoriamente** sobrescritos pelas subclasses, métodos concretos não precisam ser sobrescritos nas subclasses
- Uma subclasse que não prover implementações para os métodos abstratos herdados, deve **obrigatoriamente** ser abstrata

Exemplo 1: Aplicativo para desenho vetorial

Faça um diagrama de classes UML para representar as 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.

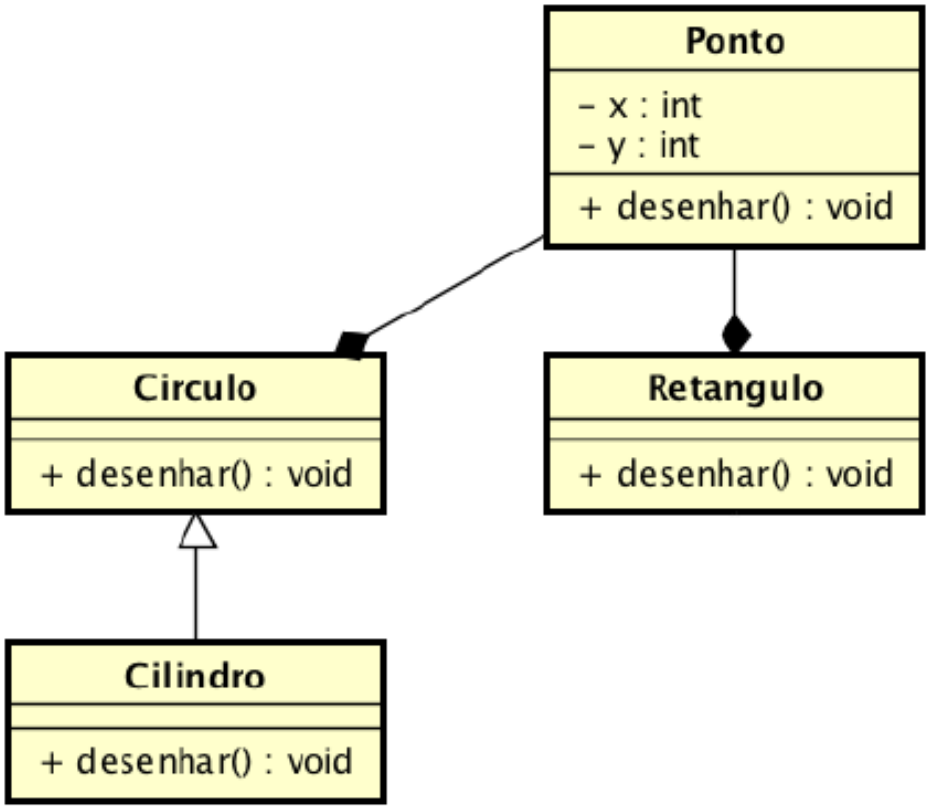
Exemplo 1: Aplicativo para desenho vetorial

Faça um diagrama de classes UML para representar as 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 1: Aplicativo para desenho vetorial

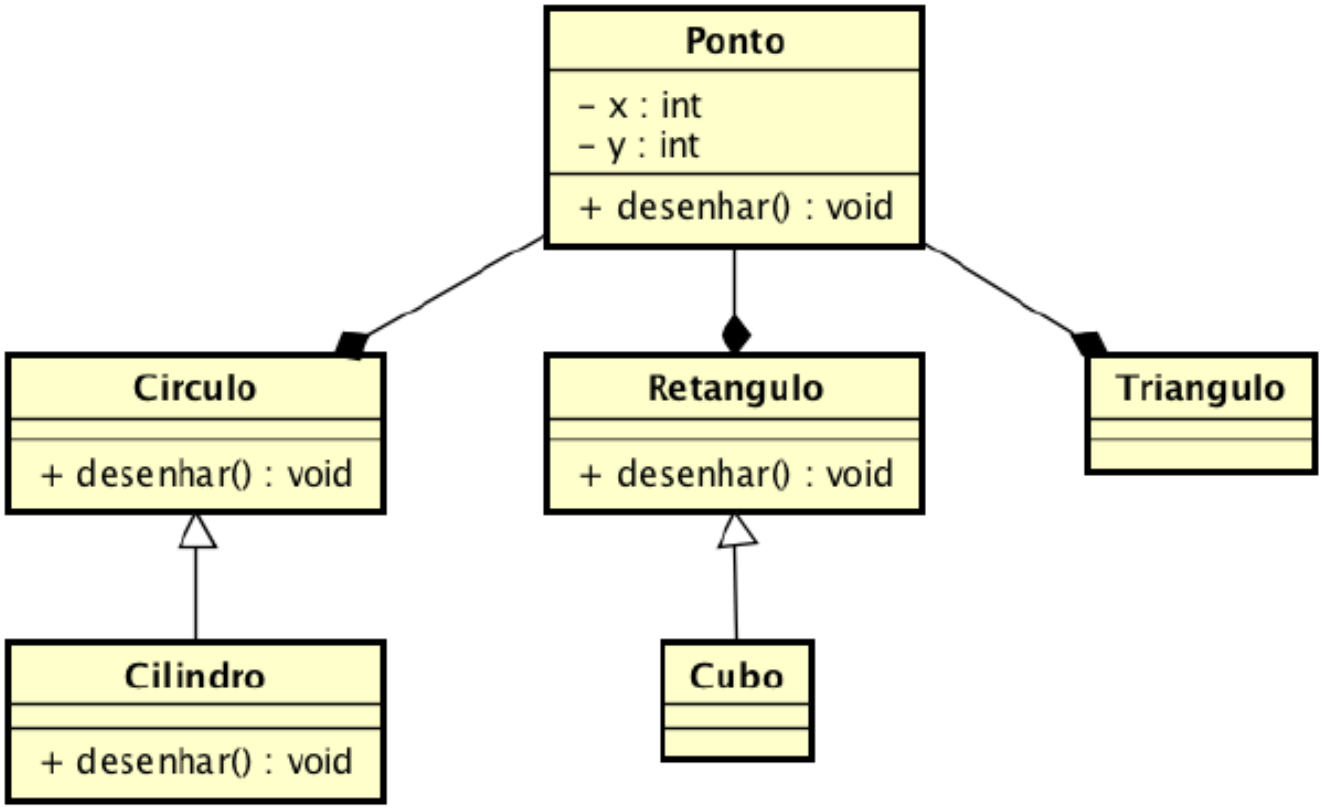
Faça um diagrama de classes UML para representar as formas geométricas: Ponto, Círculo, Cilindro e Retângulo. Todas as classes deverão ter obrigatoriamente um método **desenhar**, que uma vez invocado, fará com que a forma seja desenhada.



- Nova necessidade: criar classes para representar um Triângulo e um Cubo

Exemplo 1: Aplicativo para desenho vetorial

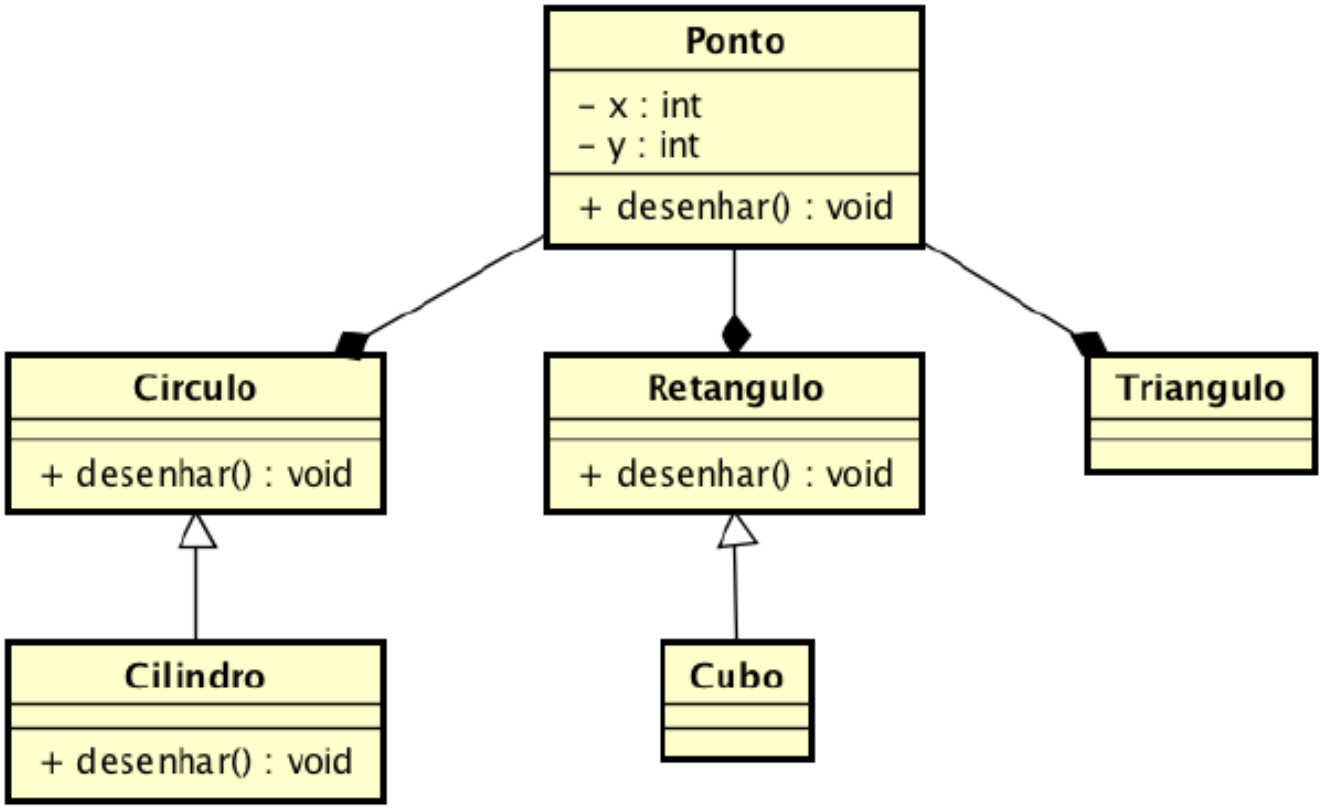
Faça um diagrama de classes UML para representar as formas geométricas: Ponto, Círculo, Cilindro e Retângulo. Todas as classes deverão ter obrigatoriamente um método **desenhar**, que uma vez invocado, fará com que a forma seja desenhada.



- Nova necessidade: criar classes para representar um Triângulo e um Cubo

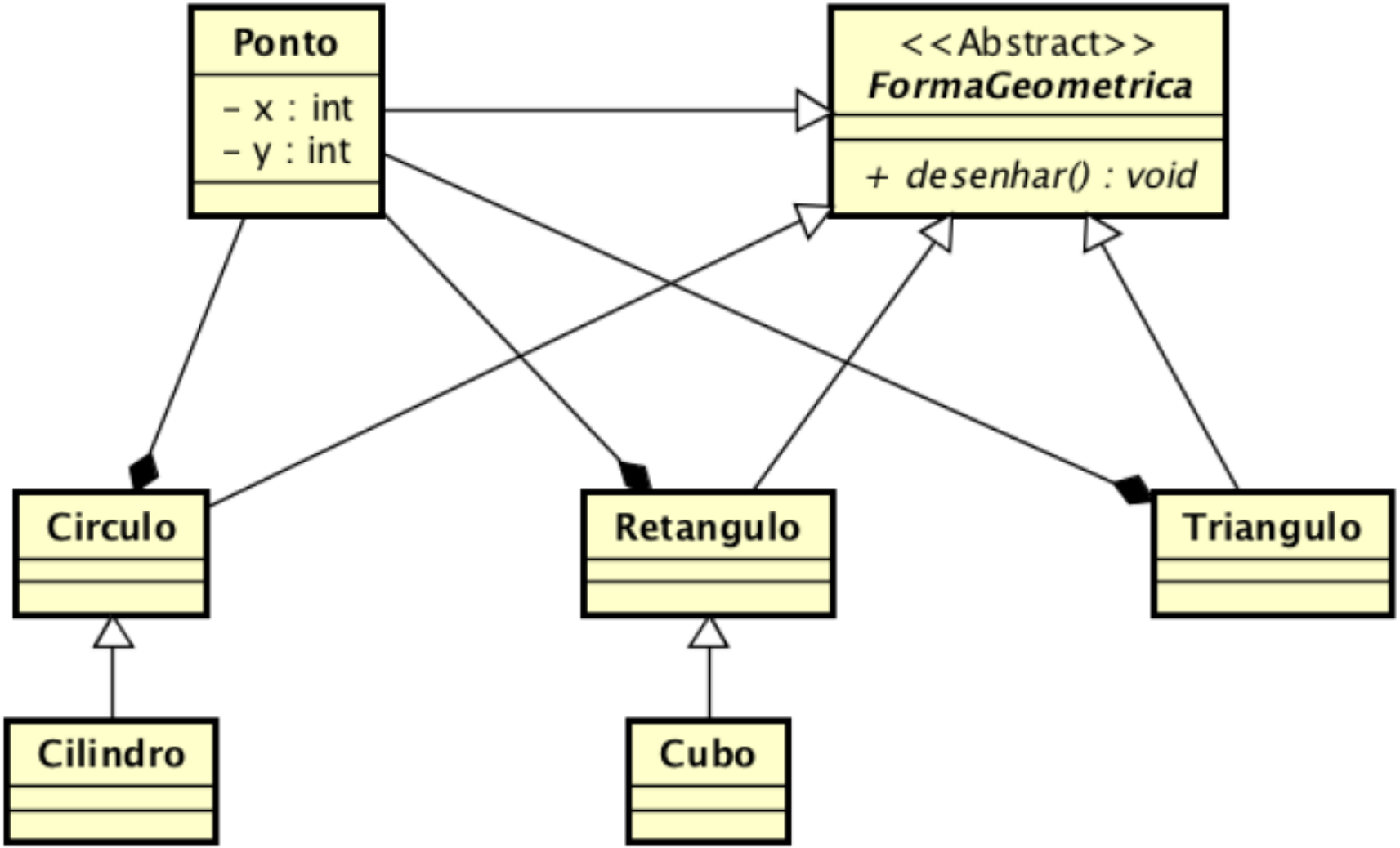
Exemplo 1: Aplicativo para desenho vetorial

Faça um diagrama de classes UML para representar as 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: criar classes para representar um Triângulo e um Cubo
- É possível garantir que as novas classes terão obrigatoriamente o método desenhar?

Exemplo 1: Aplicativo para desenho vetorial



Exemplo 1: Aplicativo para desenho vetorial

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

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

 //sobreescrita do método
 public void desenhar(){
 System.out.println("Desenhando ponto: " + x +"," + y);
 }
}
```

Exemplo 2: Classe abstrata Personagem

```
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);
 }

 // metodo abstrato que deve ser implementado na subclasse
 public abstract void atacar(float intensidade);
}
```

Exemplo 2: Classe concreta Arqueiro

```
public class Arqueiro extends Personagem{

 private int habilidade;

 public Arqueiro(int i, String n, int h){
    super(i,n);
    this.habilidade = h;
 }

 public void imprimirDados(){
    super.imprimirDados();
    System.out.println("Habilidade: " + this.habilidade);
 }

 // implementacao do metodo abstrato herdado
 public void atacar(float intensidade){
    System.out.println("Disparando flechas com a intensidade: " +
                intensidade);
 }
}
```

Exercício

1. Crie uma classe abstrata `Pessoa` e duas classes concretas, `Aluno` e `Professor`, que herdam da classe `Pessoa`
   - Cada classe deve ter no mínimo 2 atributos específicos e um método construtor que inicia todos os atributos
   - Classe `Pessoa` deverá possuir 1 método concreto e 1 método abstrato
2. Crie uma classe `Principal` e nesta crie uma instância para cada classe e invoque alguns de seus métodos

A palavra-chave **final**

A palavra-chave **final**

- Se você não quer que uma **classe** seja derivada, então preceda a declaração da classe com a palavra reservada **final**.
- Se você não quer que um **método** seja sobrescrito por uma subclasse, preceda a declaração do método com a palavra reservada **final**.
- Se você quer que uma **variável** seja apenas *read-only* (ou seja, uma constante), preceda-a com a palavra reservada **final**.

Exemplo de uma Classe **final**

```
final class MyClass {
  final int x = 3;
  public static final double PI = 3.14159;

  final double getPI() { return PI; }
}
```

Interface

Interface

- Para alguns projetos de software, desenvolvido por diferentes pessoas em diferentes instantes, é necessário criar **contratos** para que o códigos desenvolvidos por um time possam interagir com os códigos desenvolvidos pelo outro time, sem que ambos os times precisem ter conhecimento sobre o código que está escrito pelo outro.

Interface

- Para alguns projetos de software, desenvolvido por diferentes pessoas em diferentes instantes, é necessário criar **contratos** para que o códigos desenvolvidos por um time possam interagir com os códigos desenvolvidos pelo outro time, sem que ambos os times precisem ter conhecimento sobre o código que está escrito pelo outro.

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 (i.e. frear, acelerar, etc)

Interface

Uma Interface em Java é semelhante a uma classe abstrata, porém **só pode conter**:

- **constantes, métodos abstratos, métodos default e métodos estáticos**
- Por padrão todos os atributos são `public`, `static` e `final`
- Uma Interface não pode ser instanciada e o **principal objetivo é servir como referência para ser implementada por classes**
- Somente métodos estáticos ou default poderão conter implementação

Exemplo: Interface Carro

- A classe que implementa uma interface indica isso usando a palavra-chave **implements** (diferente de **extends**)

```
public interface Carro{
 /* somente para ilustrar o uso de constantes em uma interface. Nada
 util neste exemplo */
 public static final String nome = "Carro";
 // metodo abstrato por padrao
 public void frear(int intensidade);
}
```

Exemplo: Interface Carro

- A classe que implementa uma interface indica isso usando a palavra-chave **implements** (diferente de **extends**)

```
public interface Carro{
 /* somente para ilustrar o uso de constantes em uma interface. Nada
 util neste exemplo */
 public static final String nome = "Carro";
 // metodo abstrato por padrao
 public void frear(int intensidade);
}
```

```
public class Fusca implements Carro{
 private String modelo;

 public Fusca(String m){
    this.modelo = m;
 }
 public void frear(int intensidade){
    System.out.println("Encostando a lona no tambor de freio");
 }
}
```

Exemplo: Interface Carro

Classe BMW:

```
public class BMW implements Carro{

 private String modelo;

 public BMW(String m){
    this.modelo = m;
 }

 public void frear(int intensidade){
    System.out.println("Acionando ABS nas quatro rodas");
 }
}
```

Herança múltipla

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 o 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.

Resumo

Uma classe pode herdar de uma única classe e ainda assim implementar uma ou mais interfaces

Exemplo: Herança múltipla para obtermos um Triatleta

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

Exemplo: Herança múltipla para obtermos um Triatleta

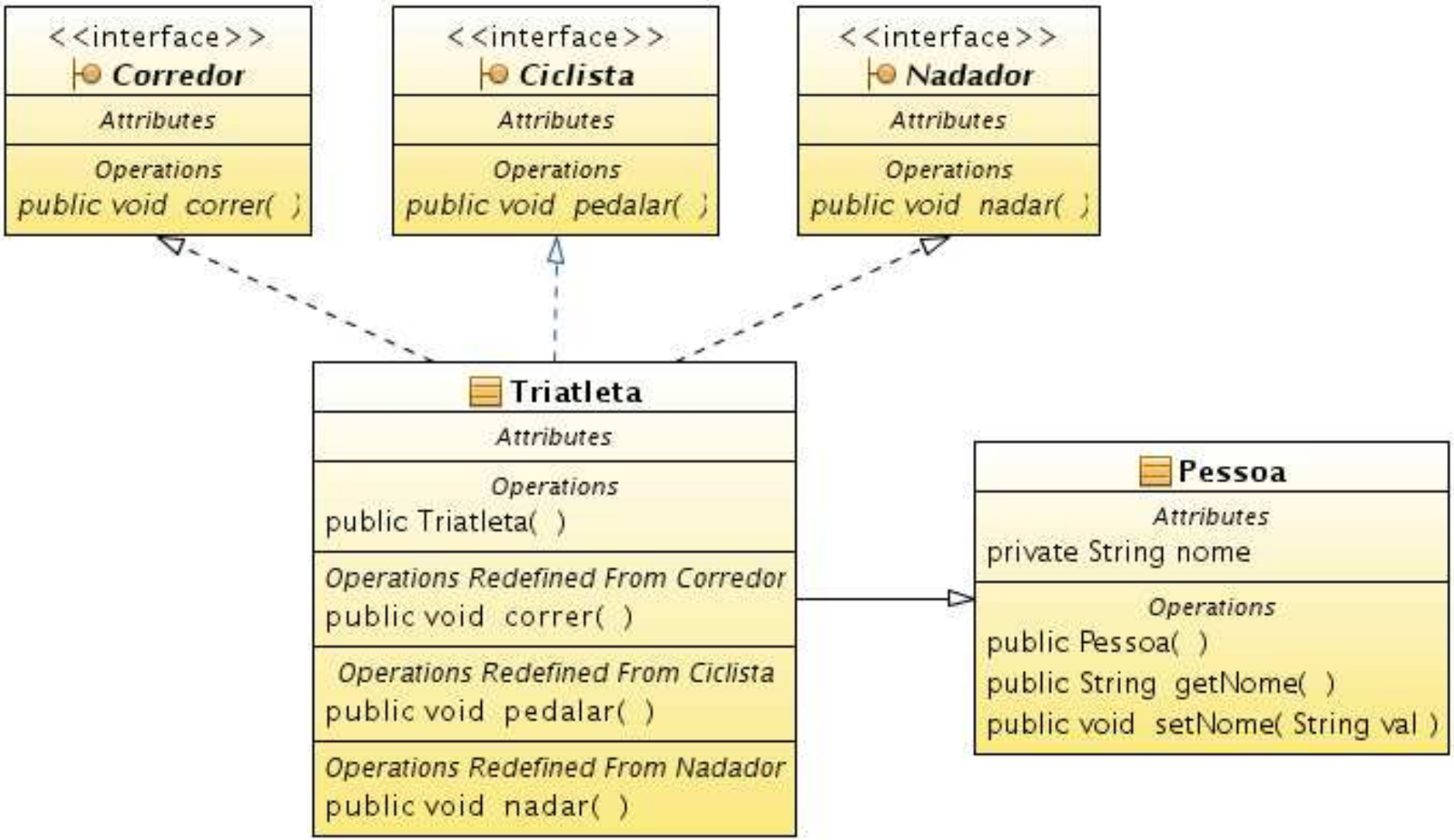
- Corredor pode correr
- Ciclista pode pedalar
- Nadador pode nadar



Desenhe um diagrama de classes UML

1. Uma classe para representar cada atleta, sabendo que todos devem possuir um nome e um CPF
   - Robson Caetano, Miguel Indurain e Cesar Cielo
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



Exercício: Herança múltipla com Interfaces

1. Faça um diagrama UML com classes e interfaces que permitam representar animais e suas habilidades (aquilo que são capazes de fazer)
   - Arara, Avestruz, Baleia, Cachorro, Gato, Leão, Lobo, Macaco, Morcego, Pinguim, Ornitorrinco
2. Implemente as classes em java e faça um aplicativo Java para instanciar alguns objetos dessas classes