
INSIGHT

INSIGHT

TÉCNICAS DE PROGRAMAÇÃO I

Herança, Classes Abstratas e Interfaces

DEPARTAMENTO DE COMPUTAÇÃO

UNIVERSIDADE FEDERAL DO CEARÁ

HERANÇA

HERANÇA

- Uma classe pode herdar propriedades e métodos de outra classe
- Em Java, podemos ter herança de **código** ou herança de **interface**
- Herança de **Interface**: herda todos métodos declarados que não sejam privados
- Herança de **Código**: herda métodos com implementações e campos que não são privados

HERANÇA

- Quando uma classe A **herda** de B, diz-se que A é a **sub-classe** e estende B, a **superclasse**
- Uma classe Java estende **apenas uma** outra classe- a essa restrição damos o nome de herança simples
- Para criar uma sub-classe, usamos a palavra reservada ***extends***

EXEMPLO DE HERANÇA

- Suponha a seguinte classe ponto:

```
class Ponto {
        float x, y;
        Ponto () {
         x=0; y=0;
        }
        Ponto(float x1, float y1) {
            x = x1; y = y1;
        }
        void setXY (float x1, float y1) {
                x = x1; y = y1;
        }
        void mover(float dx, float dy) {
                x = x + dx; y = y + dy;
        }
}
```

EXEMPLO DE HERANÇA

- Podemos criar uma classe **PontoColorido** a partir de **Ponto.**

```
class PontoColorido extends Ponto {
        int cor;

        void setCor(int c) {
                cor = c;
        }

}
```

EXEMPLO DE HERANÇA

- A classe PontoColorido herda a **interface** e o **código** da classe **Ponto**. Ou seja, PontoColorido passa a ter tanto os **campos** quanto os **métodos** (com suas implementações) de Point.

```
PontoColorido p1 = new PontoColorido();
p1.setXY(2,1);
p1.setCor(0);
p1.mover(1,0);
```

EXEMPLO DE HERANÇA

- Podemos também definir um construtor para **PontoColorido** similar ao da *superclasse* **Ponto**.

```
class PontoColorido extends Ponto {
        int cor;
        PontoColorido(float x1, float y1, int c)   {
                x = x1;
                y = y1;
                cor = c;
        }
}
```

EXEMPLO DE HERANÇA

- Podemos também definir um construtor para **PontoColorido** similar ao da *superclasse* **Ponto**.

```
class PontoColorido extends Ponto {
        int cor;
        PontoColorido(float x1, float y1, int c)   {
                x = x1;
                y = y1;
                cor = c;
        }
}
```

Repetição de código

SUPER

- Usada para **referenciar** o **construtor** da superclasse.
- PontoColorido precisa **iniciar** sua parte Ponto antes de iniciar sua parte estendida
- No caso de não haver chamada explícita, a linguagem Java chama o construtor padrão da super classe (super()), como foi o caso do exemplo anterior

EXEMPLO DE SUPER

- Podemos modificar o construtor de **PontoColorido** para utilizar o construtor da *superclasse* **Ponto**.

```
class PontoColorido extends Ponto {
        int cor;
        PontoColorido(float x1, float y1, int c)   {
                super(x1,y1);
                cor = c;
        }
}
```

HERANÇA DE CÓDIGO

- Com o uso de **super** utilizamos o código definido no método construtor da **superclasse**.
- Podemos criar agora um **PontoColorido** usando o novo construtor de **PontoColorido**:

```
PontoColorido p1 = new PontoColorido(1,2,0);
p1.mover(1,0);
```

MODELO DE HERANÇA

- Java adota o modelo de **árvore**
- A classe **Object** é a **raiz** da hierarquia de classes à qual todas as classes existentes pertencem
- Quando não declaramos que uma classe estende outra, ela, implicitamente, estende Object

ESPECIALIZAÇÃO X EXTENSÃO

- Uma classe pode herdar de outra para especializá-la, redefinindo métodos, sem ampliar sua interface
- Uma classe pode herdar de outra para estendê-la declarando novos métodos e, dessa forma, ampliando sua interface.
- Ou as duas coisas podem acontecer simultaneamente

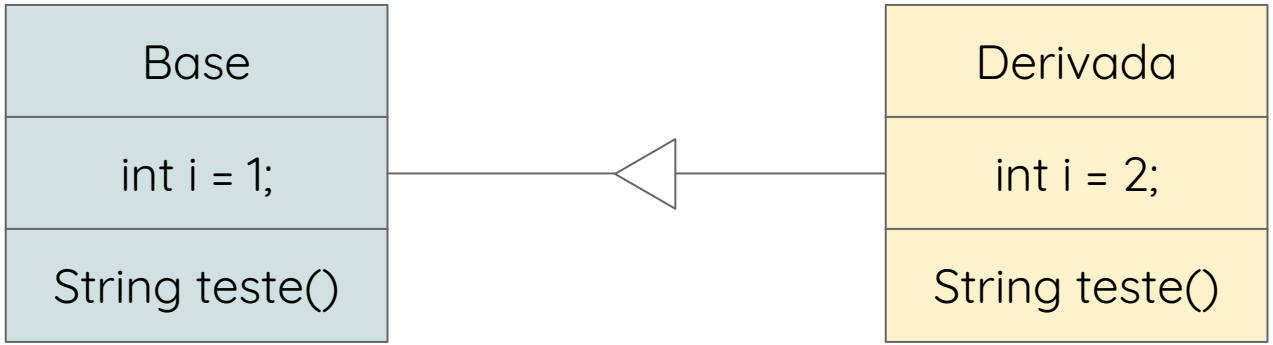
ESPECIALIZAÇÃO X EXTENSÃO

```
class Base {
        int i = 1;

        String teste() {
                return "Base";
        }
}
```

```
class Derivada extends Base {
        int i = 2;

        String teste() {
                return "Derivada";
        }
}
```



ESPECIALIZAÇÃO X EXTENSÃO

- Qual serão os valores finais de x, y, z, xstr, ystr e zstr?

```
class Heranca {
	public static void main (String arg[]) {
		Base base = new Base();
		Derivada derivada = new Derivada();
		Base baseDerivada = new Derivada();
		int x = base.i;
		int y = derivada.i;
		int z = baseDerivada.i;
		String xstr = base.teste();
		String ystr = derivada.teste();
		String zstr = baseDerivada.teste();
		System.out.println("Teste de Herança");
		System.out.println("Propriedades => "+x+" - "+y+" - "+z);
		System.out.println("Metodos      => "+xstr+" - "+ystr+" - "+zstr);
	}
}
```

ESPECIALIZAÇÃO X EXTENSÃO

- Qual serão os valores finais de x, y, z, xstr, ystr e zstr?

```
class Heranca {
	public static void main (String arg[]) {
		Base base = new Base();
		Derivada derivada = new Derivada();
		Base baseDerivada = new Derivada();
		int x = base.i;
		int y = derivada.i;
		int z = baseDerivada.i;
		String xstr = base.teste();
		String ystr = derivada.teste();
		String zstr = baseDerivada.teste();
		System.out.println("Teste de Herança");
		System.out.println("Propriedades => "+x+" - "+y+" - "+z);
		System.out.println("Metodos      => "+xString+" - "+yString+" - "+zString);
	}
}
```

```
Teste de Herança
Propriedades => 1 - 2 - 1
Metodos      => Base - Derivada - Derivada
```

SOMBREAMENTO DE MÉTODOS

```
class Super {
        Super() {
          System.out.println("Construtor    Super");
        }

        void teste() {
          System.out.println("Metodo   Superclasse");
        }
}
```

```
class Subclasse extends Super {
        Subclasse() {
           System.out.println("Construtor Subclasse");
        }

        void teste() {
          super.teste();
          System.out.println("Metodo Subclasse");
        }
}
```

SOMBREAMENTO DE MÉTODOS

- Qual será a saída do código?

```
class HerancaTeste {
        public static void main (String arg[]) {
                System.out.println("*** TESTE ***");
                Subclasse sub = new Subclasse();
                sub.teste();
        }
}
```

SOMBREAMENTO DE MÉTODOS

- Qual será a saída do código?

```
class HerancaTeste {
        public static void main (String arg[]) {
                System.out.println("*** TESTE ***");
                Subclasse sub = new Subclasse();
                sub.teste();
        }
}
```

```
*** TESTE ***
Contrutor Super
Contrutor Subclasse
Metodo Superclasse
Metodo Subclasse
```

MÉTODOS CONSTANTES

- Métodos cujas implementações **não** podem ser **redefinidas** nas sub-classes.
- Objetivo é evitar o processamento do Late Binding
- Devemos usar o modificador **final**

```
public class A{
  public final int f ( ) {
        ...
        }
}
```

CLASSES CONSTANTES

- Uma classe constante **não** pode ser **estendida**.

```
public final class A{
        ...
}
```

CLASSES ABSTRATAS

CONVERSÃO DE TIPO

- Podemos usar uma versão mais especializada quando precisamos de um objeto de certo tipo mas o contrário não é verdade. Por isso, se precisarmos fazer a conversão de volta ao tipo mais especializado, teremos que fazê-lo explicitamente.

TYPE CASTING

- A conversão explícita de um objeto de um tipo para outro é chamada de **type casting**

```
Ponto pt = new PontoColorido(0,0,1);
PontoColorido px = (PontoColorido)pt;
pt = new Ponto();
px = (PontoColorido)pt; //ERRO
pt = new PontoColorido(0,0,0);
px = pt; //ERRO
```

CLASSES ABSTRATAS

- Ao criarmos uma classe para ser estendida, às vezes codificamos alguns métodos para os quais **não sabemos** dar uma implementação, ou seja, um método que só **sub-classes** saberão implementar
- Uma classe deste tipo não pode ser instanciada pois sua funcionalidade está incompleta. Tal classe é dita **abstrata**.

CLASSES ABSTRATAS EM JAVA

- Java suporta o conceito de classes abstratas: podemos declarar uma classe abstrata usando o modificador abstract
- Métodos podem ser declarados abstratos para que suas implementações sejam adiadas para as sub-classes. Da mesma forma, usamos o modificador abstract e omitimos a implementação desse método.

CLASSES ABSTRATAS EM JAVA

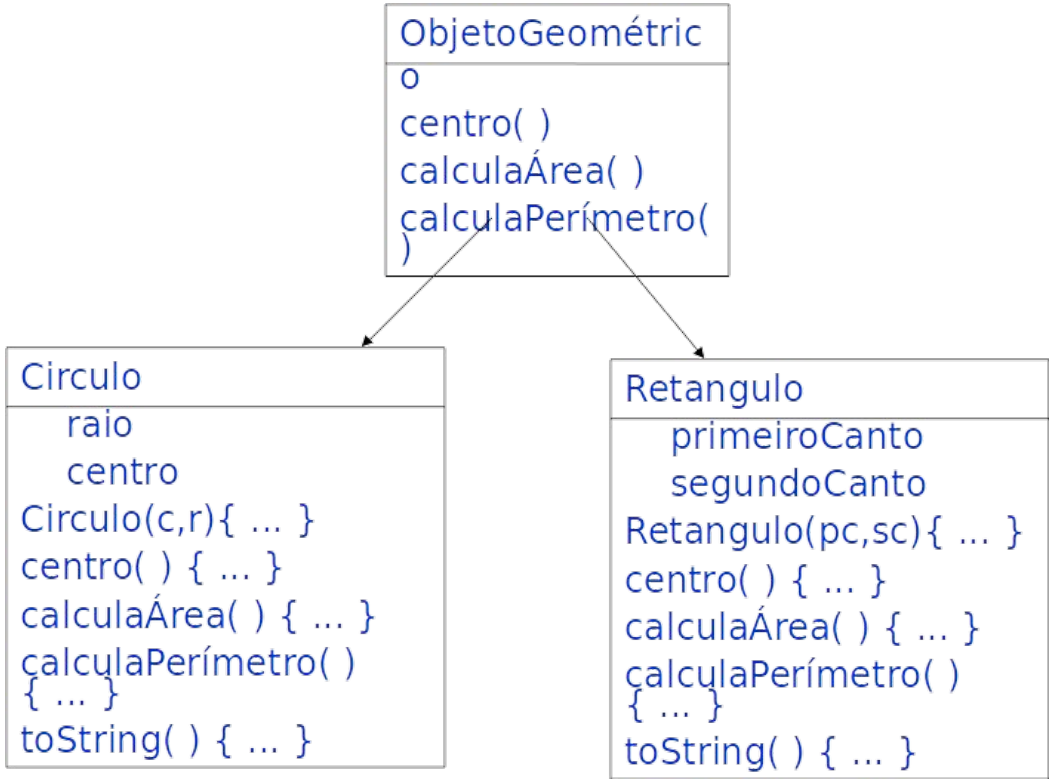
```
public abstract class Figura  {
        Ponto pOrigem;
        public abstract void desenhar();
        public abstract void apagar();
        public abstract float calcularArea();
        public void mover(Ponto p) {
                origem.mover(pOrigem.x,pOrigem.y);
        }
}
```

INTERFACE

INTERFACES

- Classes abstratas “puras” que possuem apenas métodos abstratos
- Na Interface:
  - Métodos são implicitamente **abstract** e **public**
  - Campos são implicitamente **static** e **final**
  - Não possuem construtores

INTERFACES


ObjetoGeométric
o
centro( )
calculaÁrea( )
calculaPerímetro(
)
Circulo
raio
centro
Circulo(c,r){ ... }
centro( ) { ... }
calculaÁrea( ) { ... }
calculaPerímetro( )
{ ... }
toString( ) { ... }
Retangulo
primeiroCanto
segundoCanto
Retangulo(pc,sc){ ... }
centro( ) { ... }
calculaÁrea( ) { ... }
calculaPerímetro( )
{ ... }
toString( ) { ... }

INTERFACES

```
interface ObjetoGeometrico { //declaração diferente de classes
        Ponto2D centro(); //métodos sem modificadores
        double calculaÁrea(); //informando apenas o retorno
        double calculaPerímetro(); //e, se necessário, os parâmetros
} // fim da interface ObjetoGeometrico
```

INTERFACES

```
class Circulo implements ObjetoGeometrico{
        private Ponto2D centro;
        private double raio;
        Circulo(Ponto2D centro,double raio){
           this.centro = centro;  this.raio = raio;
        }
        public Ponto2D centro(){
           return centro;
        }
        public double calculaÁrea(){
           return Math.PI*raio*raio;
        }
        public double calculaPerímetro(){
           return 2.0*Math.PI*raio;
        }
        public String toString(){
           return "Círculo com centro em "+centro+" e raio "+raio;
        }
} // fim da classe Circulo
```

INTERFACES

```
class Retangulo implements ObjetoGeometrico {
        private Ponto2D primeiroCanto,segundoCanto;
        Retangulo(Ponto2D pc,Ponto2D sc)   {
           primeiroCanto = pc;  segundoCanto = sc;   }
        public Ponto2D centro()   {
           double coordX = (primeiroCanto.getX()+segundoCanto.getX())/2.;
           double coordY = (primeiroCanto.getY()+segundoCanto.getY())/2.;
           return new Ponto2D(coordX,coordY);   }
        public double calculaÁrea()   {
          ... }
        public double calculaPerímetro()   {
          ....    }
        public String toString()   {
          return "Retângulo com cantos "+primeiroCanto+" e "+segundoCanto;   }
} // fim da classe Retangulo
```

INTERFACES

```
private static void imprimeTodosOsDados(ObjetoGeometrico og)    {
    System.out.println(og);
    System.out.println("Perímetro:"+og.calculaPerímetro());
    System.out.println("Área:"+og.calculaÁrea());
    System.out.println();
}
```

O que esse método faz?

INTERFACES COMO PARÂMETRO

- Quando uma interface é definida como parâmetro, uma instância de uma subclasse será enviada
- O método será executado de acordo com a implementação da subclasse
- Esse fenômeno chama-se **polimorfismo**

INTERFACES COMO PARÂMETRO

```
class DemoObjetosGeometricos  {
        public static void main(String[] argumentos)    {
            Circulo c1 = new Circulo(new Ponto2D(0,0),100);
            Retangulo r1 = new Retangulo(new Ponto2D(-2,-2),
                                    new Ponto2D(2,2));
            imprimeTodosOsDados(c1);
            imprimeTodosOsDados(r1);
        }

        private static void imprimeTodosOsDados(ObjetoGeometrico og)    {
            System.out.println(og);
            System.out.println("Perímetro:"+og.calculaPerímetro());
            System.out.println("Área:"+og.calculaÁrea());
            System.out.println();
        }
}
```

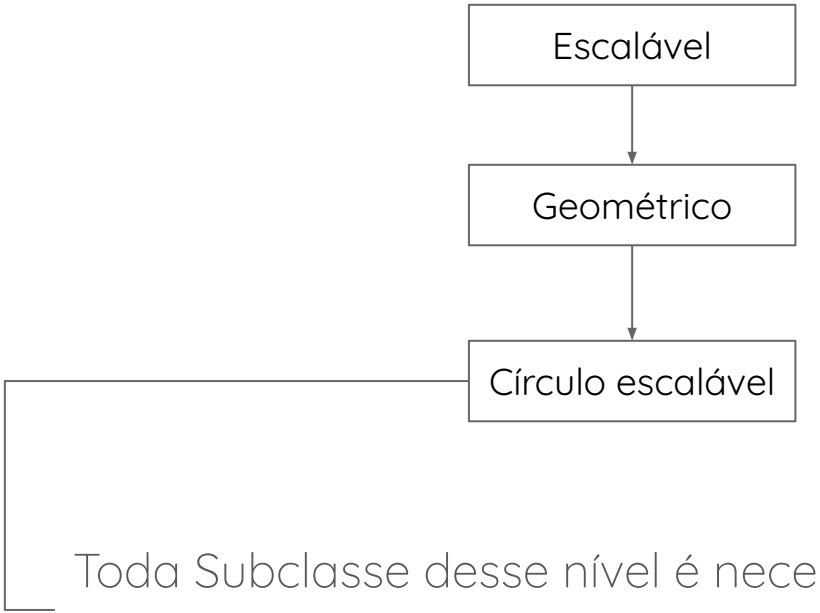
INTERFACES COMO PARÂMETRO

```
class DemoObjetosGeometricosEPolimorfismo  {
        public static void main(String[] argumentos)    {
           ObjetoGeometrico o1,o2;
           o1 = new Circulo(new Ponto2D(0,0),20);
           o2 = new Retangulo(new Ponto2D(-1,-1),
                        new Ponto2D(1,1));
           System.out.println("o1 é um Círculo ? "+
                        (o1 instanceof Circulo));
           System.out.println("o1 é um Retângulo ? "+
                        (o1 instanceof Retangulo));
           System.out.println("o1 é um ObjetoGeometrico ? "+
                        (o1 instanceof ObjetoGeometrico));
         ....
        }
} // fim da classe DemoObjetosGeometricosEPolimorfismo
```

HERANÇA MÚLTIPLA

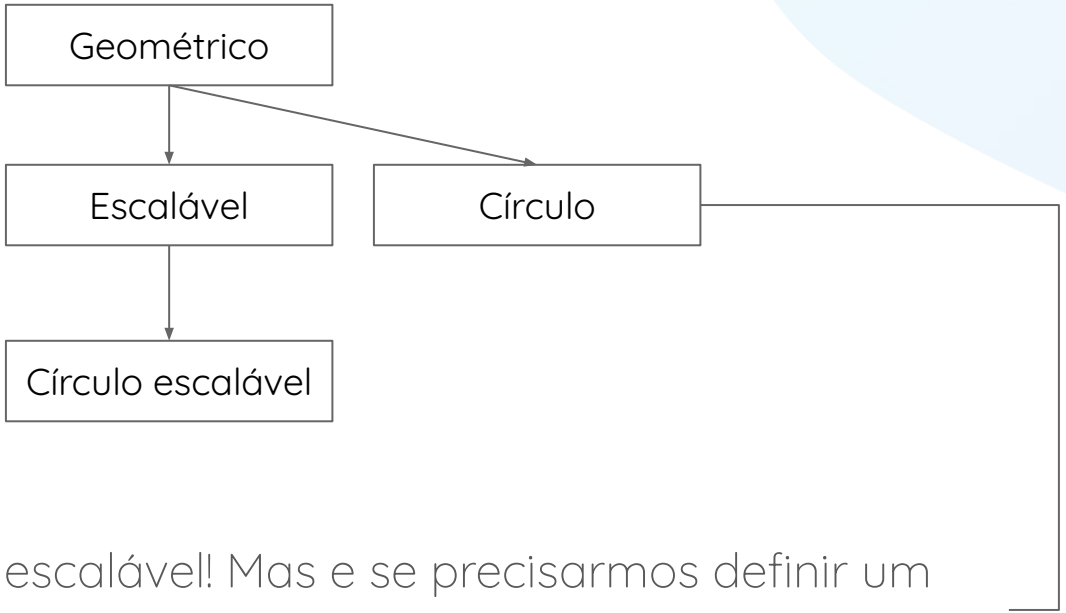
- Suponha que desejamos implementar dois tipos de objetos:
  - Objetos Geométricos
  - Objetos Escaláveis
- Precisamos levar em conta as seguintes restrições:
  - Nem todo objeto geométrico deve ser escalável
  - Nem todo objeto escalável deve ser geométrico

HERANÇA MÚLTIPLA



Toda Subclasse desse nível é necessariamente escalável e geométrico! Como não violar as restrições?

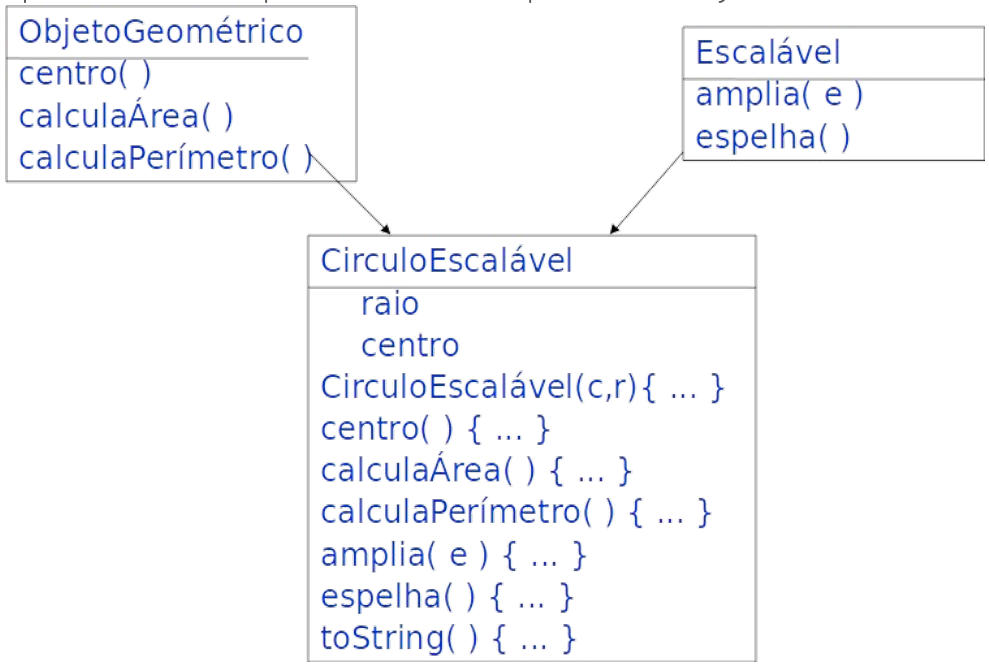
HERANÇA MÚLTIPLA



Este círculo não é escalável! Mas e se precisarmos definir um objeto escalável não geométrico?

HERANÇA MÚLTIPLA

- Para resolver esse problema, precisamos criar **interfaces** independentes para cada tipo de objeto



HERANÇA MÚLTIPLA

```
interface ObjetoGeometrico {
        Ponto2D centro();
        double calculaÁrea();
        double calculaPerímetro();
}

interface Escalavel {
        void amplia(Double escala);
        void espelha();
}
```

HERANÇA MÚLTIPLA

```
class CirculoEscalavel implements ObjetoGeometrico,Escalavel {
        private Ponto2D centro;
        private double raio;
        CirculoEscalavel(Ponto2D centro,double raio)    {
           this.centro = centro;
           this.raio = raio;        }
        public Ponto2D centro()    {
           return centro;    }
        public double calculaÁrea()    {
           return Math.PI*raio*raio;     }
        public double calculaPerímetro()    {
           return 2.0*Math.PI*raio;     }
        public void amplia(double escala)    {
           raio *= escala;    }
        public void espelha()    {
           centro = new Ponto2D(-centro.getX(),centro.getY());    }
        public String toString()    {
           return "Círculo com centro em "+centro+" e raio "+raio;    }
}
```

CONFLITOS EM HERANÇA MÚLTIPLA

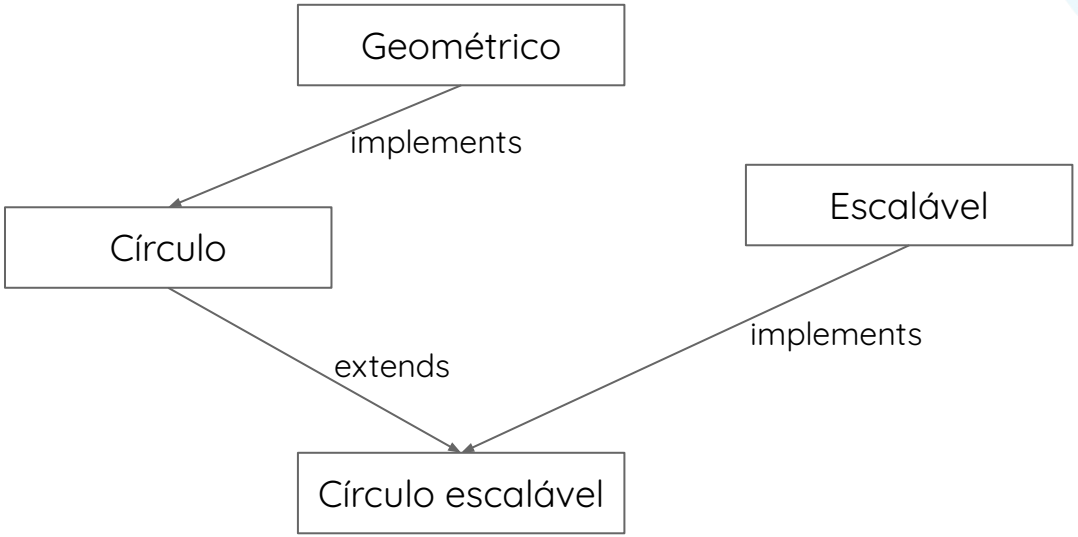
- Conflitos de **métodos**:
  - As superclasses possuem métodos com mesma assinatura. Qual deles herdar???
- Conflitos de **campos**:
  - As superclasses possuem campos com mesmo nome. Qual deles herdar ???

CONFLITOS EM HERANÇA MÚLTIPLA

- Solução C++: **herança seletiva**

- Solução Java: **interfaces**
  - Não há conflito de métodos porque a sobrescrição é obrigatória nas classes herdeiras
  - O compilador detecta conflito de campos e não compila a classe herdeira.

CONFLITOS EM HERANÇA MÚLTIPLA



O cenário acima é possível?

OBRIGADO!

Dúvidas?

Você pode me encontrar em

- jose.macedo@dc.ufc.br