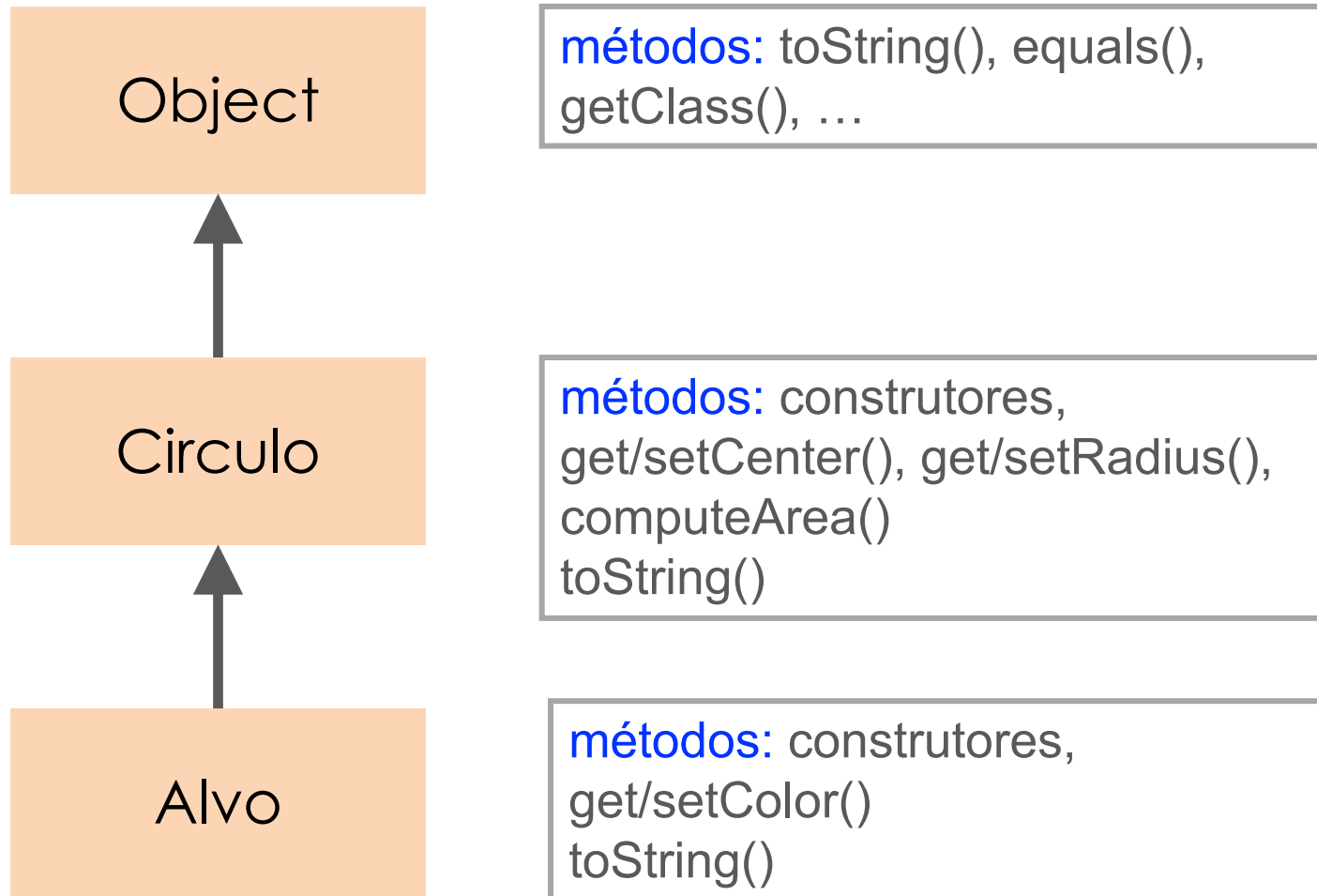


# Polimorfismo

# Exemplo de herança



# Upcasting e downcasting

```
double z = 2.75;  
int k = (int) z;  
float x = k;  
double w = 5;
```

downcast,  $k \leftarrow 2$

upcast automático  
 $x \leftarrow 2.0$ ;  $w \leftarrow 5.0$

```
Alvo fc1 = new Alvo(1.5, 10, 20, Color.red);
```

```
Circulo c1;  
c1 = fc1;
```

OK – um Alvo é um Circulo

```
Alvo fc2;  
fc2 = c1;
```

Erro! – c1 é uma referência para Circulo. Mesmo que aponte para um Alvo precisa de downcast

```
fc2 = (Alvo) c1;
```

OK

# Upcasting e downcasting

```
Circulo c2 = new Circulo(1.5f, 10, 20);
```

```
fc2 = (Alvo) c2;
```

run-time error:  
ClassCast exception

- ❖ O tipo do objeto pode ser testado com o operador instanceof

```
if (c3 instanceof Alvo)  
    fc2 = (Alvo) c3;
```

OK

# Polimorfismo

---

## ❖ Ideia base:

- o tipo declarado na referência não precisa de ser exatamente o mesmo tipo do objeto para o qual aponta – pode ser de qualquer tipo derivado

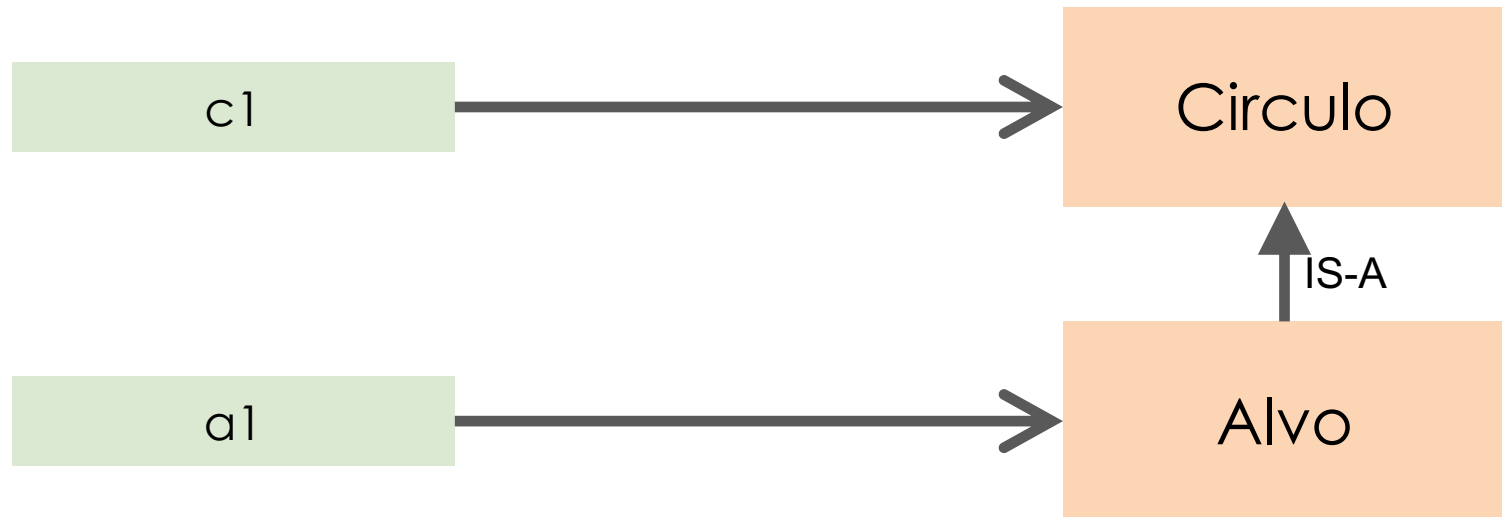
```
Circulo c1 = new Alvo(...);  
Object obj = new Circulo(...);
```

## ❖ Referência polimórfica

- T ref1 = new S();
- // OK desde que todo o S seja um T

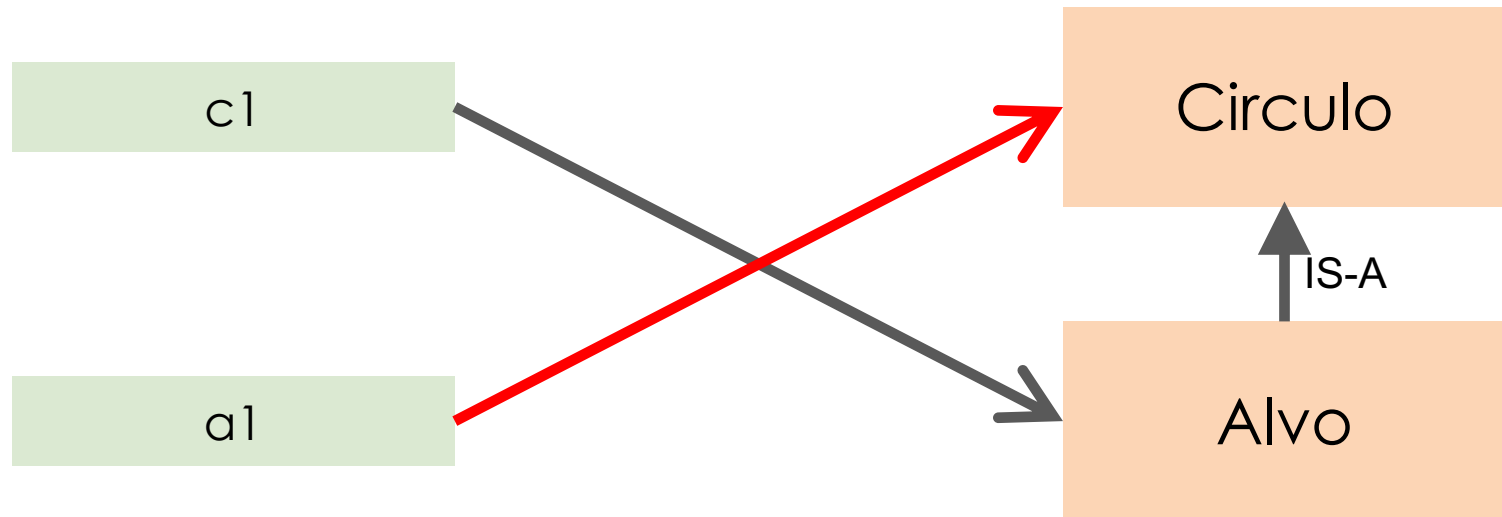
# Polimorfismo - exemplos

```
Circulo c1 = new Circulo(1,1,5);  
Alvo a1 = new Alvo(1,1,5, 10);
```



# Polimorfismo - exemplos

```
Circulo c1 = new Alvo(1,1,5, 10);  
Alvo a1 = new Circulo(1,1,5); // erro
```



# Polimorfismo

---

- ❖ Polimorfismo é, conjuntamente com a Herança e o Encapsulamento, uma das características fundamentais da POO.
  - Formas diferentes com interfaces semelhantes.
- ❖ Diretamente associado ao mecanismo de ligação dinâmica (*Dynamic binding*)
  - Também referido como *late binding* ou *run-time binding*
- ❖ Esta característica permite-nos tirar mais partido da herança.
  - Podemos, por exemplo, desenvolver um método X() com parâmetro CBase com a garantia que aceita qualquer argumento derivado de CBase.
  - O método X() só é resolvido em execução.
- ❖ Todos os métodos (à exceção dos finais) são *late binding*.
  - O atributo final associado a uma função, impede que ela seja redefinida e simultaneamente dá uma indicação ao compilador para ligação estática (*early binding*) - que é o único modo de ligação em linguagens como C.

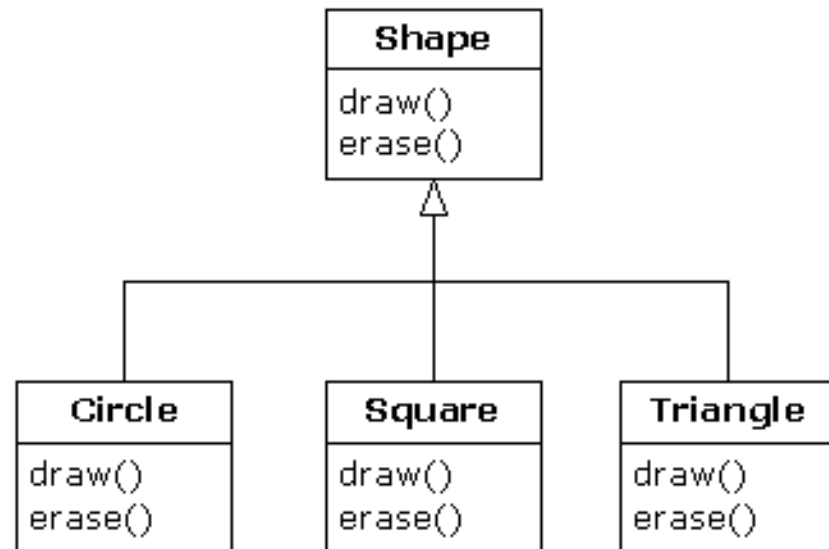


# Exemplo 1

```
Shape s = new Shape();  
s.draw();
```

```
Circulo c = new Circulo();  
c.draw();
```

```
Shape s2 = new Circulo();  
s2.draw();
```



# Exemplo 2

```
class Shape {  
    void draw() { System.out.println("I am a Shape"); }  
}  
  
class Circle extends Shape {  
    void draw() { System.out.println("I am a Circle"); }  
}  
  
class Square extends Shape {  
    void draw() { System.out.println("I am a Square"); }  
}  
  
public class ShapeSet {  
  
    private static Shape randomShape() {  
        if (Math.random() < 0.5) return new Circle();  
        return new Square();  
    }  
  
    Run | Debug  
    public static void main(String[] args) { args = String[0]@7  
        Shape[] shapes = new Shape[8]; shapes = Shape[8]@8  
        for (int i=0; i<shapes.length; i++)  
            shapes[i] = randomShape();  
  
        for (Shape s: shapes) s = Circle@17, shapes = Shape[8]@8  
        s.draw(); s = Circle@17  
    }  
}
```

I am a Circle  
I am a Square  
I am a Circle  
I am a Square  
I am a Circle  
I am a Square  
I am a Square  
I am a Square

# Generalização

---

- ❖ A generalização consiste em melhorar as classes de um problema de modo a torná-las mais gerais.

- ❖ Formas de generalização:

- ❖ Tornar a classe o mais abrangente possível de forma a cobrir o maior leque de entidades.

```
class ZooAnimal;
```

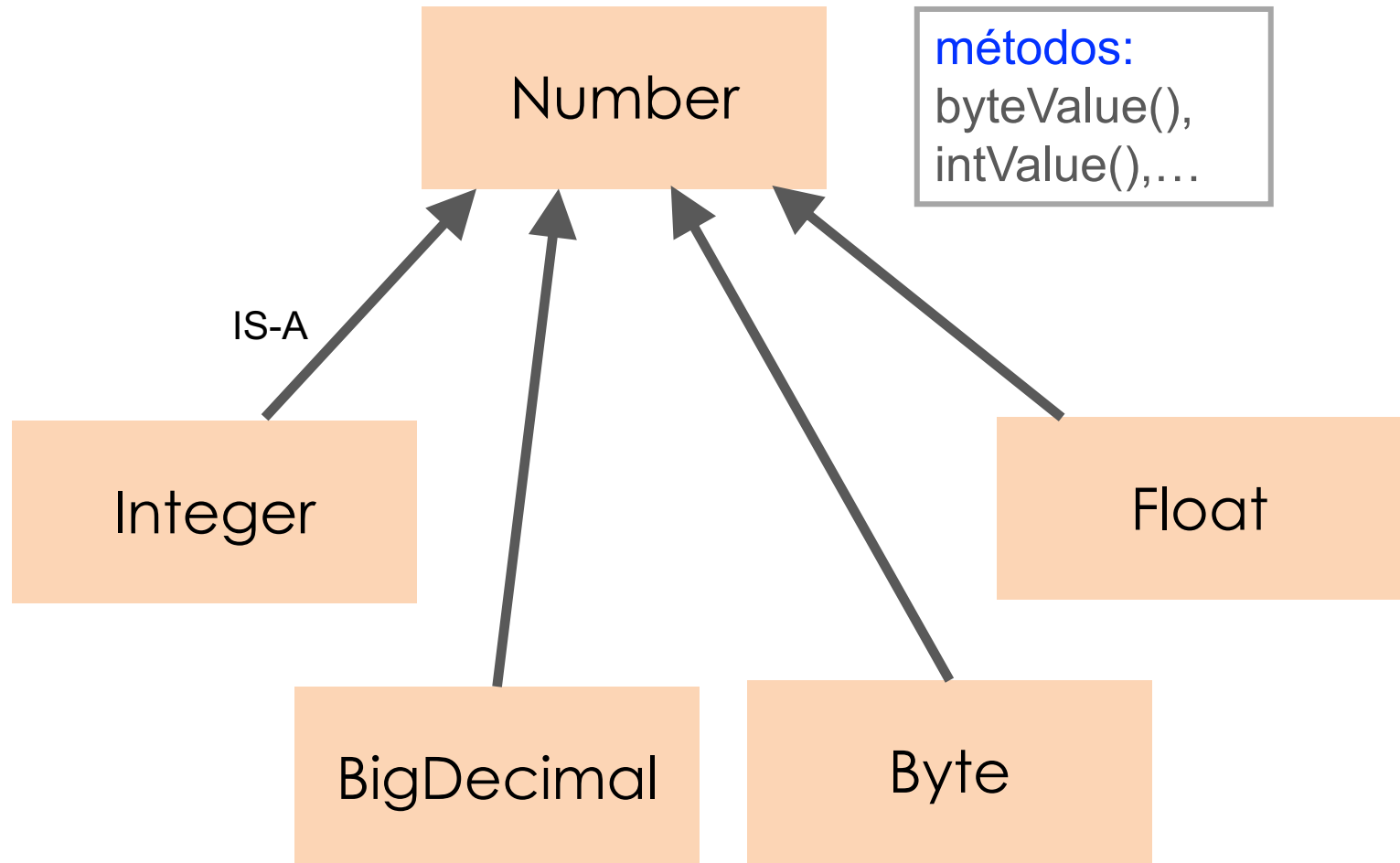
- ❖ Abstrair implementações diferentes para operações semelhantes em classes abstratas num nível superior.

```
ZooAnimal.draw();
```

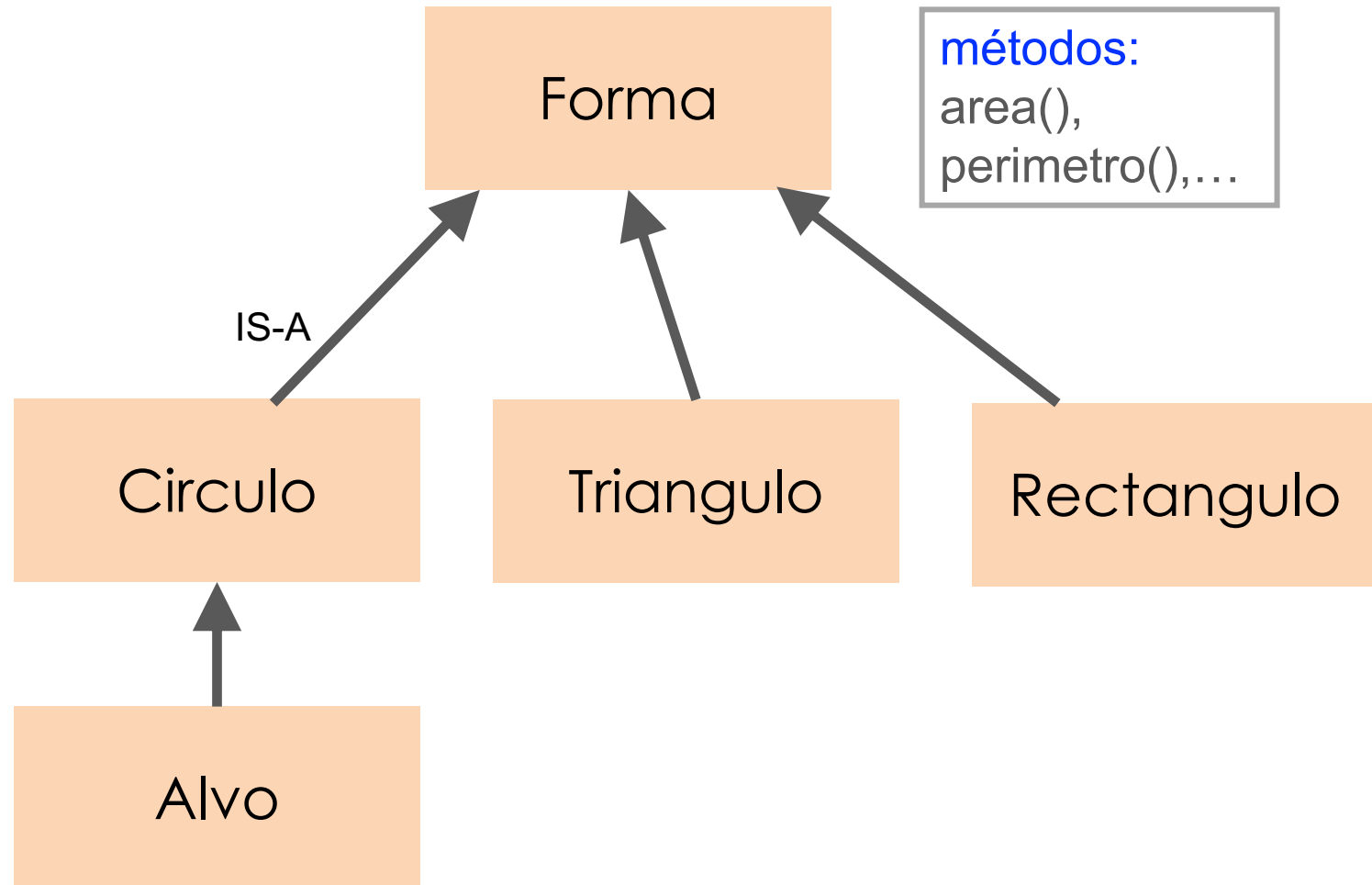
- ❖ Reunir comportamentos e características e fazê-los subir o mais possível na hierarquia de classes.

```
ZooAnimal.peso;
```

# Exemplo de herança

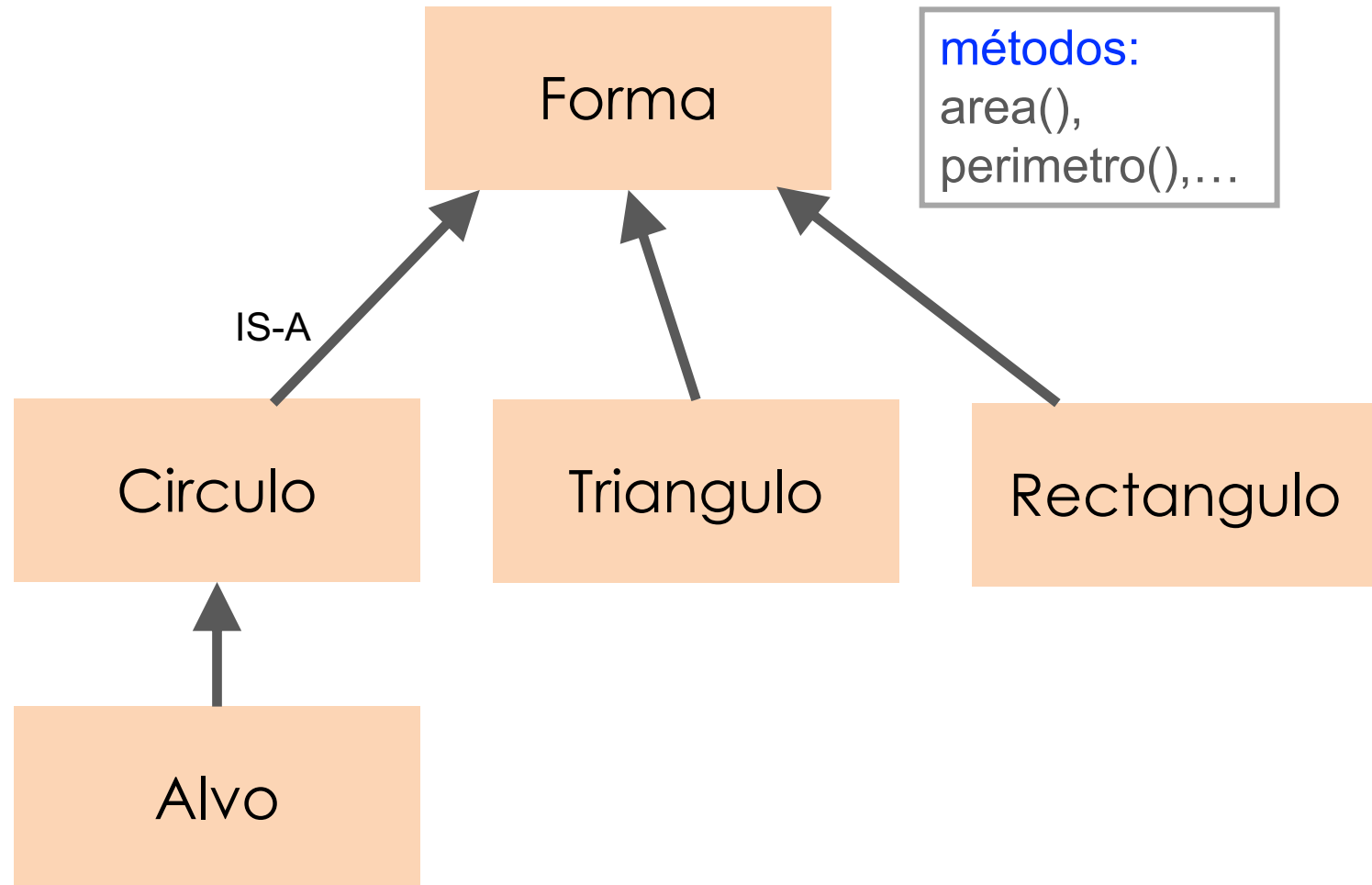


# Exemplo de herança



# Exemplo de herança

❖ Como implementamos os métodos de Forma?



# Classes abstratas

- ❖ Uma classe é abstrata se contiver pelo menos um método abstrato.

- Um método abstrato é um método cujo corpo não é definido.

```
public abstract class Forma {  
    // pode definir constantes  
    public static final double DOUBLE_PI = 2*Math.PI;  
  
    // pode declarar métodos abstractos  
    public abstract double area();  
    public abstract double perimetro();  
  
    // pode incluir métodos não abstractos  
    public String aka() { return "euclidean"; }  
}
```

- ❖ Uma classe abstrata não é instanciável.

```
Forma f;           // OK. Podemos criar uma variável do tipo Forma  
f = new Forma();   // Erro! Não podemos criar Formas
```

# Classes abstratas

---

- ❖ Num processo de herança a classe só deixa de ser abstrata quando implementar todos os métodos abstratos.

```
public class Circulo extends Forma {  
  
    protected double r;  
  
    public double area() {  
        return Math.PI*r*r;  
    }  
  
    public double perimetro() {  
        return DOUBLE_PI*r;  
    }  
}  
  
Forma f;  
f = new Circulo();    // OK! Podemos criar Circulos
```



# Classes abstratas e Polimorfismo

```
abstract class Figura {
    abstract void doWork();
    protected int cNum;
}

class Circulo extends Figura {
    Circulo(int i) { cNum = i; }
    void doWork() { System.out.println("Circulo"); }
}

class Alvo extends Circulo {
    Alvo(int i) { super(i); }
    void doWork() { System.out.println("Alvo"); }
}

class Quadrado extends Figura {
    void doWork() { System.out.println("Quadrado"); }
}

public class ArrayOfObjects {
    public static void main(String[] args) {

        Figura[] anArray = new Figura[10];
        for (int i = 0; i < anArray.length; i++) {
            switch ((int) (Math.random() * 3)) {
                case 0 : anArray[i] = new Circulo(i); break;
                case 1 : anArray[i] = new Alvo(i); break;
                case 2 : anArray[i] = new Quadrado(); break;
            }
        }
        // invoca o método doWork sobre todas as Figura da tabela
        // -- Polimorfismo
        for (int i = 0; i < anArray.length; i++) {
            System.out.print("Figura(" + i + ") --> ");
            anArray[i].doWork();
        }
    }
}
```

Figura(0) --> Quadrado

Figura(1) --> Circulo

Figura(2) --> Quadrado

Figura(3) --> Circulo

Figura(4) --> Quadrado

Figura(5) --> Alvo

Figura(6) --> Circulo

Figura(7) --> Circulo

Figura(0) --> Circulo

Figura(1) --> Quadrado

Figura(2) --> Alvo

Figura(3) --> Quadrado

Figura(4) --> Alvo

Figura(5) --> Quadrado

Figura(6) --> Quadrado

Figura(7) --> Quadrado

Figura(8) --> Circulo

Figura(9) --> Quadrado

# Java Interfaces

# Interfaces

---

- ❖ Uma interface funciona como uma classe que só contém assinaturas
  - A partir do Java 8 passou a incluir métodos *default* e *static*.

```
public interface Desenhavel {  
    //...  
}
```

- ❖ Atua como um protocolo perante as classes que as implementam.

```
public class Grafico implements Desenhavel {  
    // ...  
}
```

- ❖ Uma classe pode herdar de uma só classe base e implementar uma ou mais interfaces.

# Interfaces - Exemplo

---

```
interface Desenhavel {  
    public void cor(Color c);  
    public void corDeFundo(Color cf);  
    public void posicao(double x, double y);  
    public void desenha(DrawWindow dw);  
}
```

```
class CirculoGrafico extends Circulo implements Desenhavel {  
    public void cor(Color c) {...}  
    public void corDeFundo(Color cf) {...}  
    public void posicao(double x, double y) {...}  
    public void desenha(DrawWindow dw) {...}  
}
```

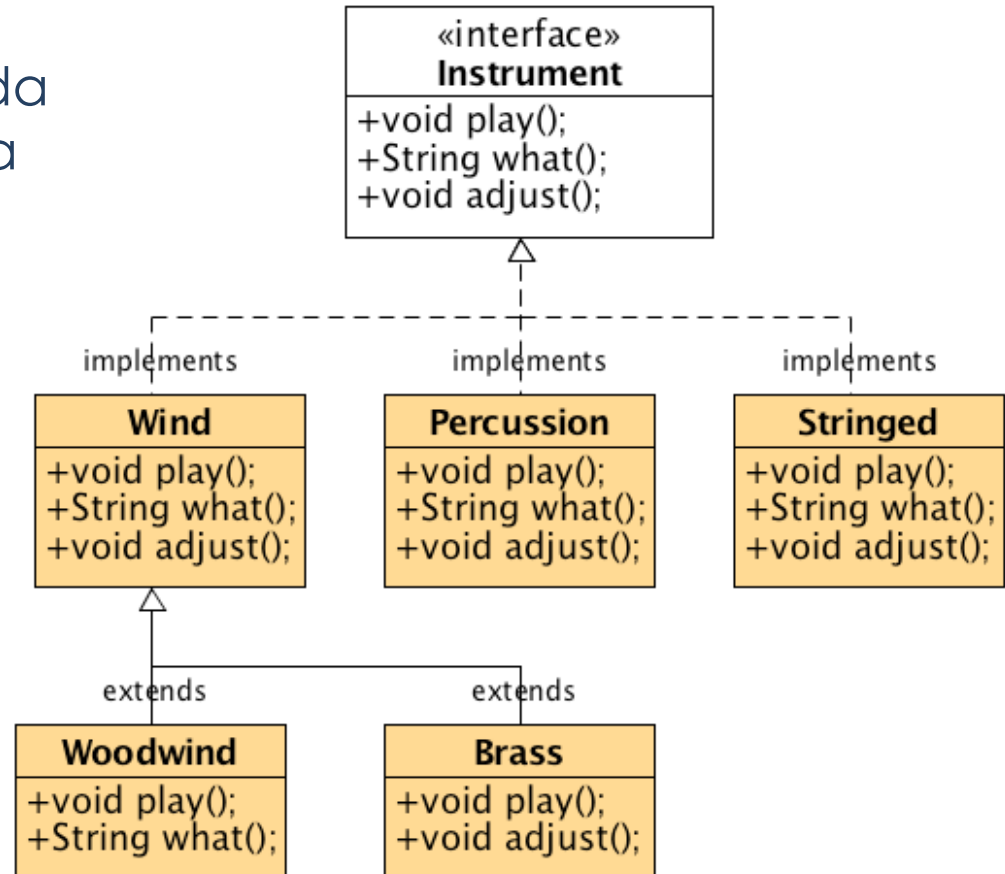
# Características principais

---

- ❖ Todos os seus métodos são, implicitamente, abstratos.
  - Os únicos modificadores permitidos são *public* e *abstract*.
- ❖ Uma interface pode herdar (*extends*) mais do que uma interface.
- ❖ Não são permitidos construtores.
- ❖ As variáveis são implicitamente estáticas e constantes
  - `static final` ..
- ❖ Uma classe (não abstrata) que implemente uma interface deve implementar todos os seus métodos.
- ❖ Uma interface pode ser vazia
  - `Cloneable`, `Serializable`
- ❖ Não se pode criar uma instância da interface
- ❖ Pode criar-se uma referência para uma interface

# Interfaces - Exemplos

- ❖ Depois de implementada uma interface passam a atuar as regras sobre classes



# Interfaces - Exemplos

---

```
interface Instrument {  
    // Compile-time constant:  
    int i = 5; // static & final  
    // Cannot have method definitions:  
    void play(); // Automatically public  
    String what();  
    void adjust();  
}  
  
class Wind implements Instrument {  
    public void play() {  
        System.out.println("Wind.play()");  
    }  
    public String what() { return "Wind"; }  
    public void adjust() { /* .. */ }  
}
```

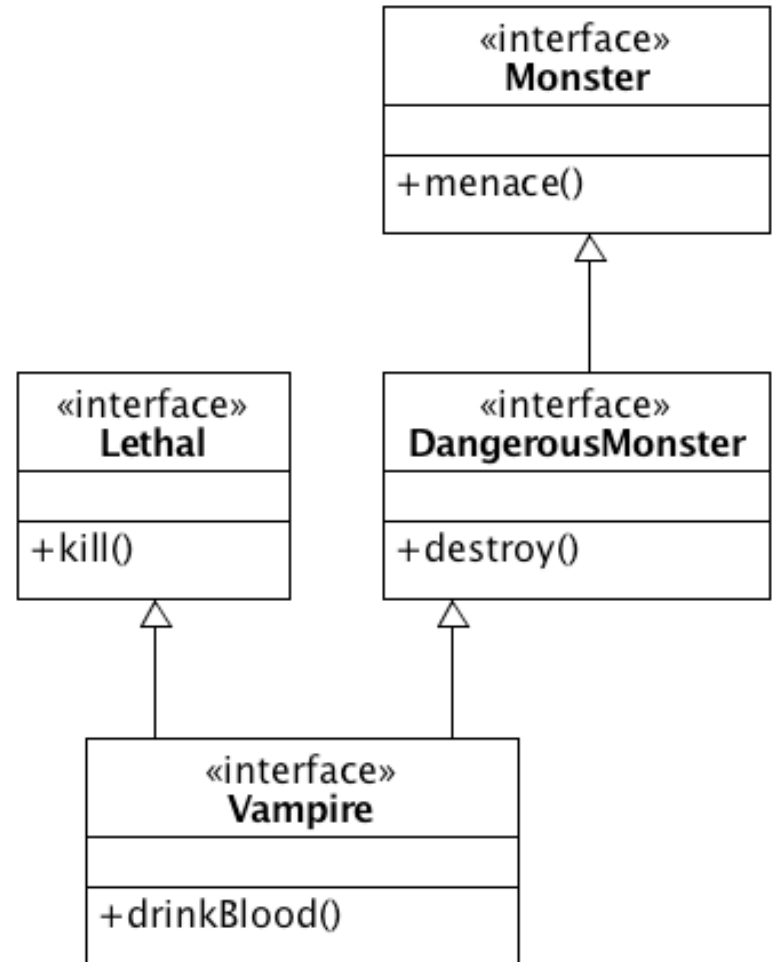
# Herança em Interfaces

```
interface Monster {  
    void menace();  
}
```

```
interface DangerousMonster  
    extends Monster {  
    void destroy();  
}
```

```
interface Lethal {  
    void kill();  
}
```

```
interface Vampire  
    extends DangerousMonster,  
        Lethal {  
    void drinkBlood();  
}
```





# Interfaces a partir de Java 8

---

## ❖ Default methods

- Podemos definir o corpo dos métodos na interface

## ❖ Static methods

- Podemos definir o corpo de métodos estáticos na interface. Devem ser invocados sobre a interface (Métodos de Interface)

## ❖ Functional interfaces

- *(vamos falar nisto mais tarde...)*

## ❖ **porquê (complicar com) estas novas funcionalidades?**

# Interfaces a partir de Java 8

---

## ❖ Default Methods

- Oferecem uma implementação por omissão
- Podem ser reescritos nas classes que implementam a interface

```
public interface InterfaceOne {  
    default void defMeth() { //... do something  
    }  
}
```

```
public class MyClass implements InterfaceOne {  
    @Override  
    public void defMeth() { // ... do something  
    }  
}
```

# Default methods

```
interface X {  
    default void foo() {  
        System.out.println("foo");  
    }  
}  
  
class Y implements X {  
    // ...  
}  
  
public class Testes {  
    public static void main(String[] args) {  
        Y myY = new Y();  
        myY.foo();  
        // ...  
    }  
}
```

# Interfaces a partir de Java 8

## ❖ Static Methods

- Similares aos default methods
- Não podem ser reescritos nas classes que implementam a interface
- Só podem ser invocados através da interface onde estão definidos  
(não através das classes que implementam a interface)

```
public interface Interface2 {  
    static void stMeth() { //... do something  
    }  
}
```

```
public class MyClass implements Interface2 {  
    @Override  
    public void stMeth() { // ... do something  
    }  
}
```



# Static methods

```
interface X {  
    static void foo() {  
        System.out.println("foo");  
    }  
}  
  
class Y implements X {  
    // ...  
}  
  
public class Testes {  
    public static void main(String[] args) {  
        X.foo();  
        // Y.foo(); // won't compile  
    }  
}
```

# Classes Abstratas versus Interfaces

---

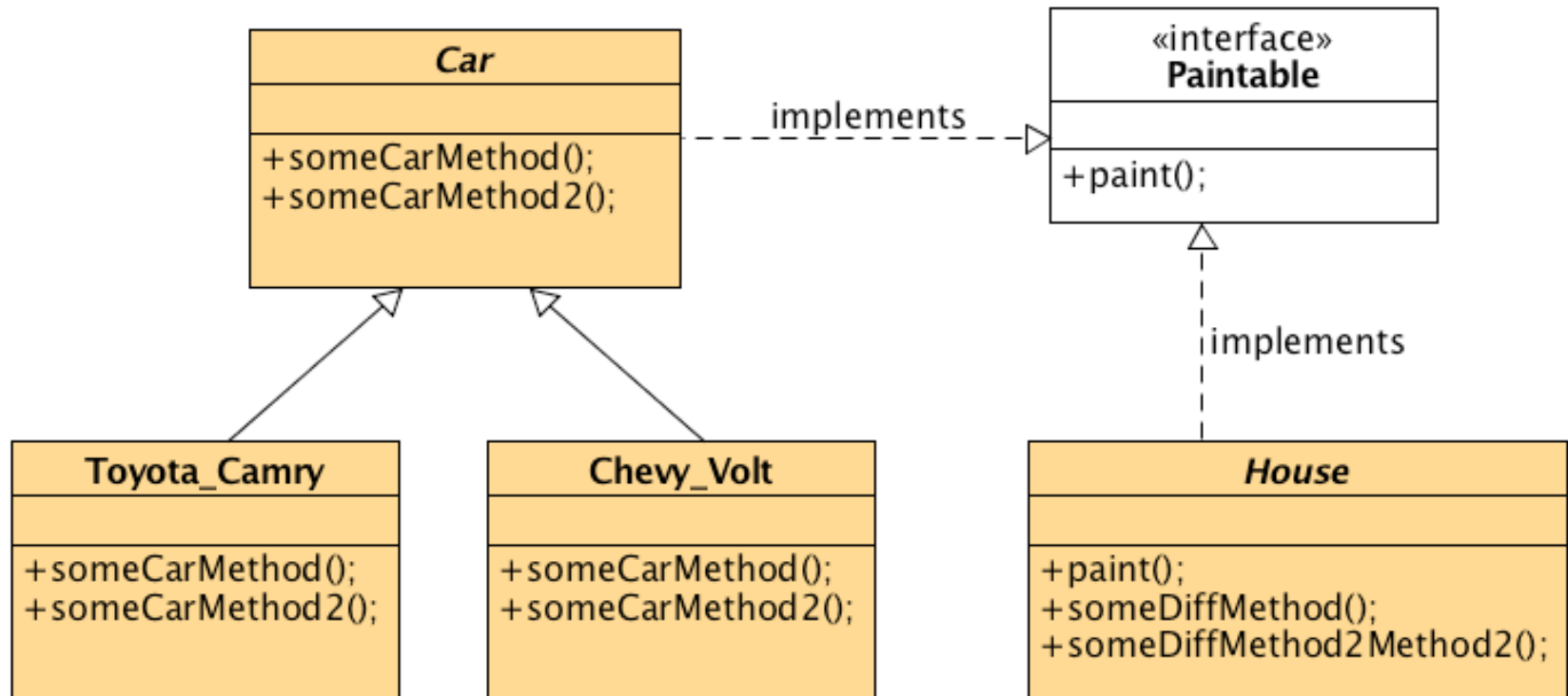
## Classes Abstratas

- ❖ Objetivo: descrever entidades e propriedades
- ❖ Podem implementar interfaces
- ❖ Permitem herança simples
- ❖ Relacionamento na hierarquia simples de classes

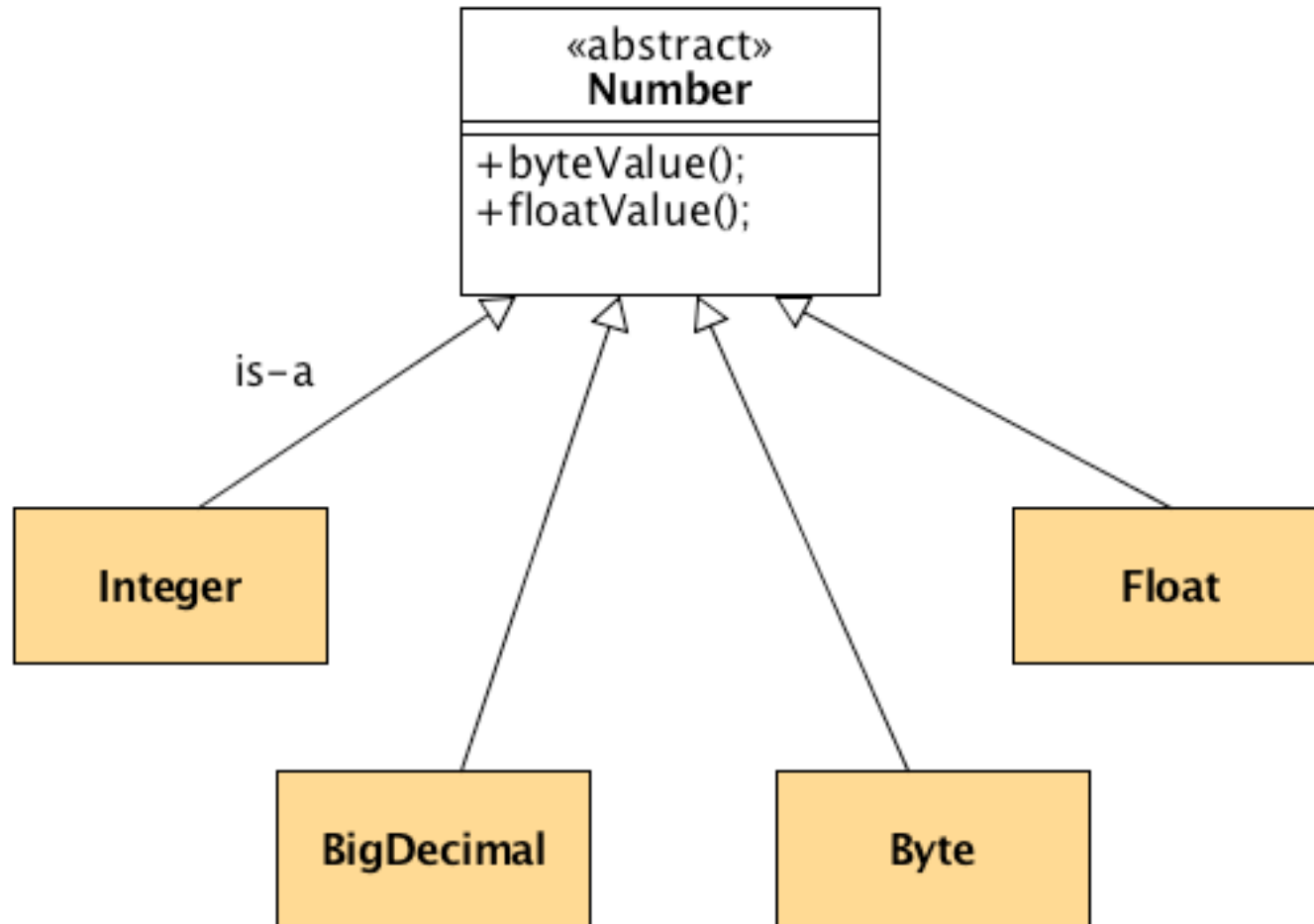
## Interfaces

- ❖ Objetivo: descrever comportamentos funcionais
- ❖ Não podem implementar classes
- ❖ Permitem herança múltipla
- ❖ Implementação horizontal na hierarquia

# Classes Abstratas versus Interfaces

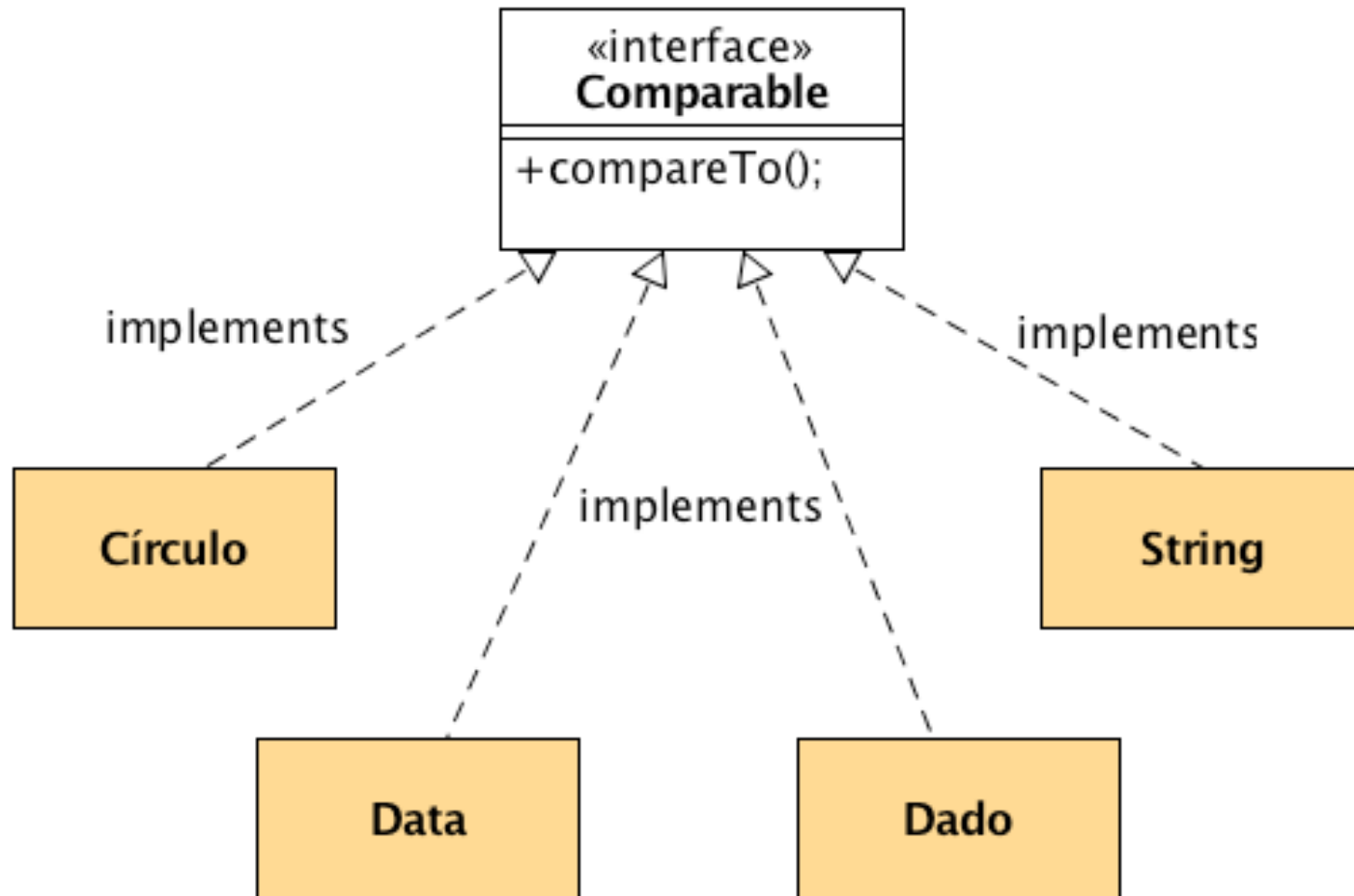


# Classes Abstratas versus Interfaces



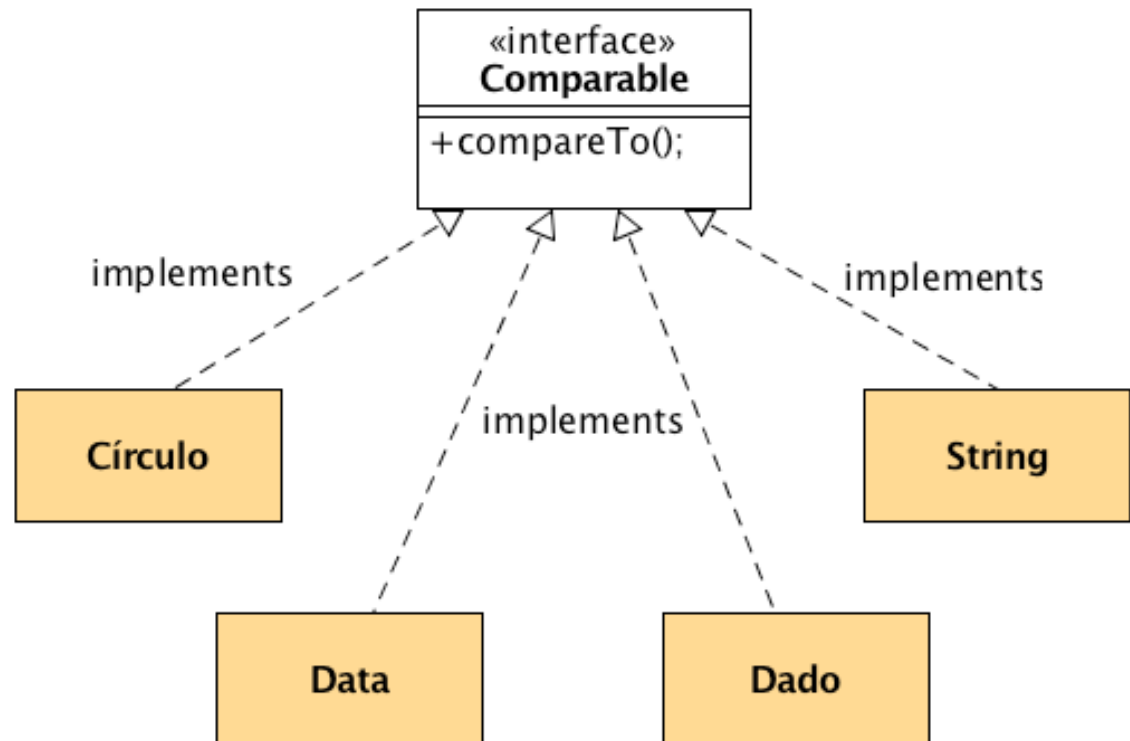


# Classes Abstratas versus Interfaces



# Questões?

- ❖ Qual o interesse de usar uma interface neste caso?
- ❖ Note que o método `int compareTo(T c)` retorna:
  - `<0` se `this < c`
  - `0` se `this == c`
  - `>0` se `this > c`



# Interface Comparable

```
public interface Comparable<T> { // package java.lang;
    int compareTo(T other);
}

public abstract class Shape implements Comparable<Shape> {
    public abstract double area( );
    public abstract double perimeter( );

    public int compareTo( Shape irhs ) {
        double res = area() - irhs.area();
        if (res > 0) return 1;
        else if (res < 0) return -1;
        else return 0;
    }
}
```

# Interface Comparable

```
public class UtilCompare {  
  
    // vamos discutir "<T extends Comparable<T>>" mais tarde  
    public static <T extends Comparable<T>> findMax(T[] a) {  
        int maxIndex = 0;  
        for (int i = 1; i < a.length; i++)  
            if (a[i] != null && a[i].compareTo(a[maxIndex]) > 0)  
                maxIndex = i;  
        return a[maxIndex];  
    }  
  
    public static <T extends Comparable<T>> void sortArray(T[] a)  
    {  
        // ...  
    }  
}
```

# Interface Comparable

```
class FindMaxDemo {  
    public static void main( String [ ] args ) {  
        Figura[] sh1 = {  
            new Circulo(1, 3, 1),           // x, y, raio  
            new Quadrado(3, 4, 2),          // x, y, lado  
            new Rectangulo(1, 1, 5, 6)      // x, y, lado1, lado2  
        };  
        String[] st1 = { "Joe", "Bob", "Bill", "Zeke" };  
  
        System.out.println(UtilCompare.findMax(sh1));  
        System.out.println(UtilCompare.findMax(st1));  
    }  
}
```

Rectangulo de Centro (1.0,1.0), altura 6.0, comprimento 5.0  
Zeke

# instanceof

---

- ❖ Operador que indica se uma referência é membro de uma classe ou interface
- ❖ Exemplo, considerando

```
class Dog extends Animal implements Pet {...}  
Animal fido = new Dog();
```

- ❖ as instruções seguintes são true:

```
if (fido instanceof Dog) ..  
if (fido instanceof Animal) ..  
if (fido instanceof Pet) ..
```

# Sumário

---

❖ Herança

❖ Polimorfismo

❖ Generalização

❖ Classes abstratas

❖ Interfaces