

# Classes e Herança

UA.DETI.POO

# Relações entre Classes

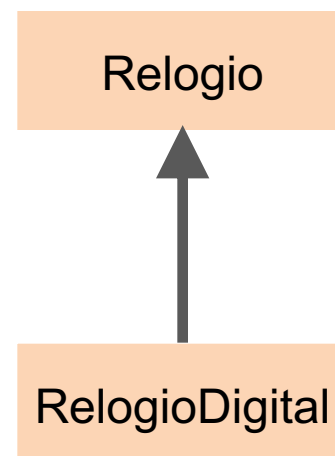
---

- ❖ Parte do processo de modelação em classes consiste em:
  - Identificar entidades candidatas a classes
  - Identificar relações entre estas entidades
- ❖ As relações entre classes identificam-se facilmente recorrendo a alguns modelos reais.
  - Por exemplo, um RelógioDigital e um RelógioAnalógico são ambos tipos de Relógio (especialização ou herança).
  - Um RelógioDigital, por seu lado, contém uma Pilha (composição).
- ❖ Relações:
  - IS-A
  - HAS-A

# Herança (IS-A)

- ❖ **IS-A** indica especialização (herança) ou seja, quando uma classe é um sub-tipo de outra classe.
- ❖ Por exemplo:
  - Pinheiro é uma (IS-A) Árvore.
  - Um RelógioDigital é um (IS-A) Relógio.

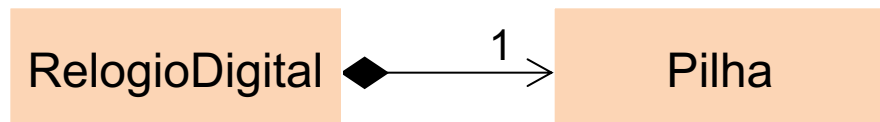
```
class Relogio {  
    /* ... */  
}  
  
class RelogioDigital extends Relogio {  
    /* ... */  
}
```



# Composição (HAS-A)

- ❖ HAS-A indica que uma classe é composta por objetos de outra classe.
- ❖ Por exemplo:
  - Floresta contém (HAS-A) Árvores.
  - Um RelógioDigital contém (HAS-A) Pilha.

```
class Pilha {  
    /* ... */  
}  
class RelogioDigital extends Relogio {  
    Pilha p;  
    /* ... */  
}
```



# Reutilização de classes

---

- ❖ Sempre que necessitamos de uma classe, podemos:
  - Recorrer a uma classe já existente que cumpre os requisitos
  - Escrever uma nova classe a partir "do zero"
  - Reutilizar uma classe existente usando composição
  - Reutilizar uma classe existente através de herança

# Identificação de Herança

---

- ❖ Sinais típicos de que duas classes têm um relacionamento de herança
  - Possuem aspetos comuns (dados, comportamento)
  - Possuem aspetos distintos
  - Uma é uma especialização da outra
- ❖ Exemplos:
  - Gato é um Mamífero
  - Circulo é uma Figura
  - Água é uma Bebida

# Questões?

---

❖ Quais as relações entre:

- Trabalhador, Motorista, Vendedor, Administrativo e Contabilista
- Triângulo, Retângulo e Losango
- Professor, Aluno e Funcionário
- Autocarro, Viatura, Roda, Motor, Pneu, Jante

# Questões?

---

- ❖ Represente os seguintes elementos (classes) bem como as suas relações (herança e composição)
  - Livro
  - Artigo
  - Jornal
  - Publicação
  - Autor
  - Periódico
  - Editora
  - LivroEditado
  - Revista



# Herança - Conceitos

---

- ❖ A herança é uma das principais características da POO
  - A classe *CDeriv* herda, ou é derivada, de *CBase* quando *CDeriv* representa um sub-conjunto de *CBase*
- ❖ A herança representa-se na forma:  

```
class CDeriv extends CBase { /* ... */ }
```
- ❖ *Cderiv* tem acesso aos dados e métodos de *CBase*
  - que não sejam privados em *CBase*
- ❖ Uma classe base pode ter múltiplas classes derivadas mas uma classe derivada não pode ter múltiplas classes base
  - Em Java não é possível a herança múltipla

# Herança - Exemplo

```
class Person {  
    private String name;  
    public Person(String n) { name = n; }  
    public String name() { return name; }  
    public String toString() { return "PERSON";}  
}
```

Base

```
class Student extends Person {  
    private int nmec;  
    public Student(String s, int n) { super(s); nmec=n; }  
    public int num() { return nmec; }  
    public String toString() { return "STUDENT"; }  
}
```

Derivada

```
public class Test {  
    public static void main(String[] args) {  
        Person p = new Person("Joaquim");  
        Student stu = new Student("Andreia", 55678);  
        System.out.println(p + " : " + p.name());  
        System.out.println(stu + " : " + stu.name() + ", " + stu.num());  
    }  
}
```

PERSON : Joaquim

STUDENT : Andreia, 55678

# Herança - Exemplo

```
class Art {  
    Art() {  
        System.out.println("Art constructor");  
    }  
}  
class Drawing extends Art {  
    Drawing() {  
        System.out.println("Drawing constr.");  
    }  
}  
public class Cartoon extends Drawing {  
    Cartoon() {  
        System.out.println("Cartoon constr.");  
    }  
    public static void main(String[] args) {  
        Cartoon x = new Cartoon();  
    }  
}
```

Art constructor

Drawing constr.

Cartoon constr.

A construção é feita a partir da classe base

# Construtores com parâmetros

- ❖ Em construtores com parâmetros o construtor da classe base é a primeira instrução a aparecer num construtor da classe derivada.

```
class Game {  
    int num;  
    Game(int code) { ... }  
    // ...  
}  
  
class BoardGame extends Game {  
    // ...  
    BoardGame(int code, int numPlayers) {  
        super(code);  
        // ...  
    }  
}
```

# Herança de Métodos

---

- ❖ Ao herdar métodos podemos:
  - mantê-los inalterados,
  - acrescentar-lhe funcionalidades novas ou
  - redefini-los

# Herança de Métodos - herdar

---

```
class Person {
    private String name;
    public Person(String n) { name = n; }
    public String name() { return name; }
    public String toString() { return "PERSON";}
}
class Student extends Person {
    private int nmec;
    public Student(String s, int n) { super(s); nmec=n; }
    public int num() { return nmec; }
}
public class Test {
    public static void main(String[] args) {
        Student stu = new Student("Andreia", 55678);
        System.out.println(stu + " : " +
            stu.name() + ", " + stu.num());
    }
}
```

# Herança de Métodos - redefinir

---

```
class Person {  
    private String name;  
    public Person(String n) { name = n; }  
    public String name() { return name; }  
    public String toString() { return "PERSON";}  
}
```

```
class Student extends Person {  
    private int nmec;  
    public Student(String s, int n) { super(s); nmec=n; }  
    public int num() { return nmec; }  
    public String toString() { return "STUDENT"; }  
}
```

# Herança de Métodos - estender

---

```
class Person {  
    private String name;  
    public Person(String n) { name = n; }  
    public String name() { return name; }  
    public String toString() { return "PERSON";}  
}
```

```
class Student extends Person {  
    private int nmec;  
    public Student(String s, int n) { super(s); nmec=n; }  
    public int num() { return nmec; }  
    public String toString()  
        { return super.toString() + " STUDENT"; }  
}
```



# Herança e controlo de acesso

---

- ❖ Não podemos reduzir a visibilidade de métodos herdados numa classe derivada
  - Métodos declarados como public na classe base devem ser public nas subclasses
  - Métodos declarados como protected na classe base devem ser protected ou public nas subclasses. Não podem ser private
  - Métodos declarados sem controlo de acesso (default) não podem ser private em subclasses
  - Métodos declarados como private não são herdados

# Final

---

- ❖ O classificador final indica "não pode ser mudado"
- ❖ A sua utilização pode ser feita sobre:
  - Dados - constantes  
`final int i1 = 9;`
  - Métodos - não redefiníveis  
`final int swap(int a, int b) { //:  
}`
  - Classes - não herdadas  
`final class Rato { //...  
}`
- ❖ "final" fixa como constantes atributos de tipos primitivos mas não fixa objetos nem vetores
  - nestes casos o que é constante é simplesmente a referência para o objeto

```

class Value {  int i = 1;  }
public class FinalData {
    // Can be compile-time constants
    private final int i1 = 9;
    private static final int VAL_TWO = 99;
    // Typical public constant:
    public static final int VAL_THREE = 39;

    public final int i4 = (int)(Math.random()*20);
    public static final int i5 = (int)(Math.random()*20);

    private Value v1 = new Value();
    private final Value v2 = new Value();
    private final int[] a = { 1, 2, 3, 4, 5, 6 }; // Arrays

    public static void main(String[] args) {
        FinalData fd1 = new FinalData();
        //! fd1.i1++; // Error: can't change value
        fd1.v2.i++; // Object isn't constant!
        fd1.v1 = new Value(); // OK -- not final
        for(int i = 0; i < fd1.a.length; i++)
            fd1.a[i]++; // Object isn't constant!
        //! fd1.v2 = new Value(); // Can't change ref
        //! fd1.a = new int[3];
    }
}

```

# Exemplo – classe Ponto

---

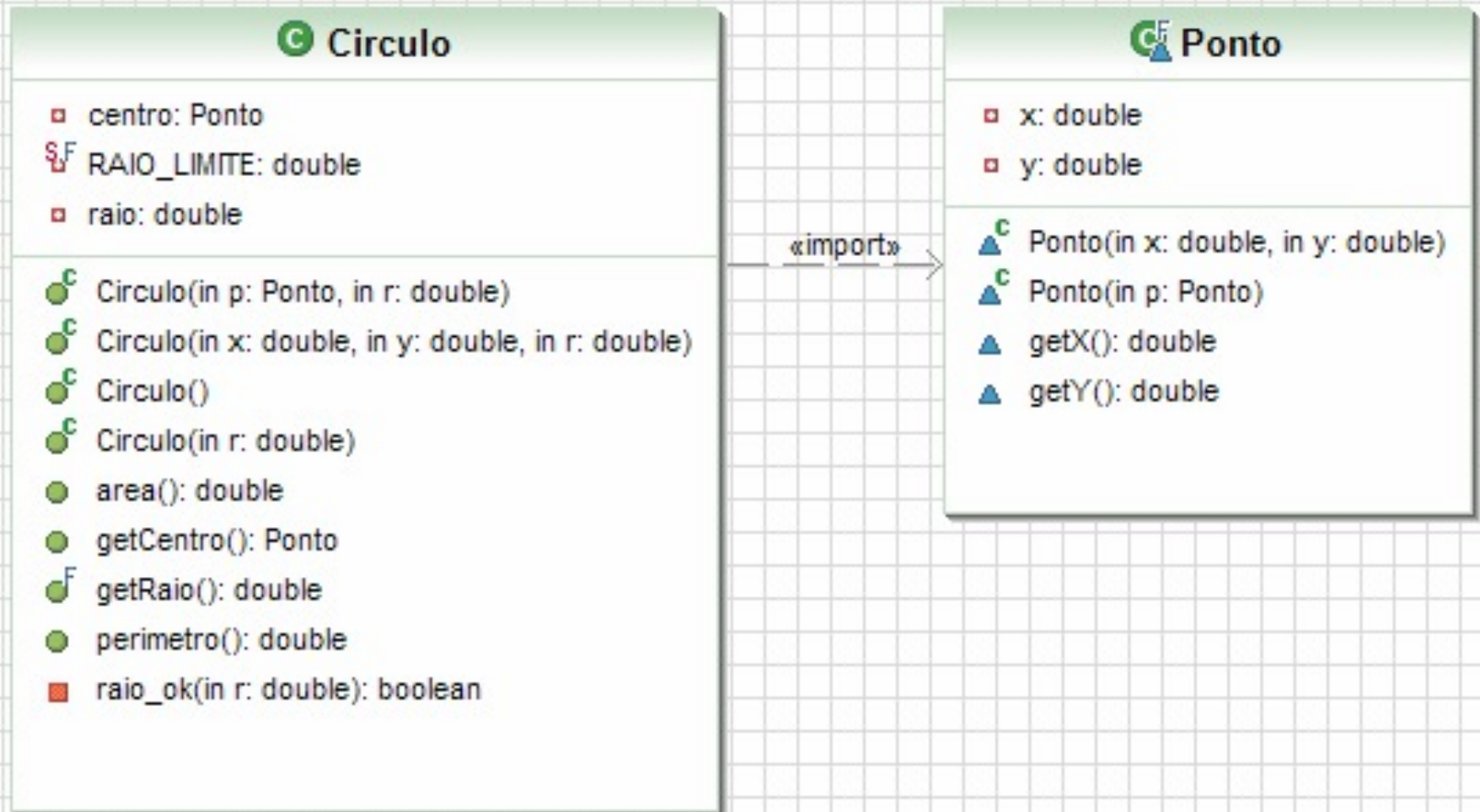
```
public final class Ponto {  
    private double x;  
    private double y;  
  
    public Ponto(double x, double y) { this.x=x; this.y=y; }  
    public final double x() { return(x); }  
    public final double y() { return(y); }  
}
```

# Exemplo – classe Circulo

---

```
public class Circulo {  
    private Ponto centro;  
    private double raio;  
  
    public static final double RAIIO_LIMITE = 100.0;  
  
    public Circulo(Ponto p, double r) {  
        centro = p;  
        if (raio_ok(r)) raio = r; else raio = RAIIO_LIMITE;  
    }  
  
    public double area() { return Math.PI*raio*raio; }  
    public double perimetro() { return 2*Math.PI*raio; }  
    public final double raio() { return raio; }  
    public final Ponto centro() { return centro; }  
  
    private boolean raio_ok(double r) { return(r<=RAIO_LIMITE); }  
  
}
```

# Representação UML



# Herança - Boas Práticas

---

- ❖ Programar para a interface e não para a implementação
- ❖ Procurar aspectos comuns a várias classes e promovê-los a uma classe base
- ❖ Minimizar os relacionamentos entre objetos e organizar as classes relacionadas dentro de um mesmo package
- ❖ Usar herança criteriosamente – sempre que possível favorecer a composição

# Métodos comuns a todos os objetos

---

- ❖ Em Java, todas as classes derivam da super classe `java.lang.Object`
- ❖ Métodos desta classe:
  - `toString()`
  - `equals()`
  - `hashCode()`
  - `finalize()`
  - `clone()`
  - `getClass()`
  - `wait()`
  - `notify()`
  - `notifyAll()`



# toString()

```
Circulo c1 = new Circulo(1.5, 0, 0);  
System.out.println( c1 );
```

c1.toString() é invocado automaticamente

Circulo@1afa3

- O método toString() deve ser sempre redefinido para ter um comportamento de acordo com o objeto

```
public class Circulo {  
    // ....  
    @Override  
    public String toString() {  
        return "Centro : (" + centro.x() + ", "  
            + centro.y() + ") " + " Raio : " + raio;  
    }  
}
```

Centro : (1.5, 0) Raio : 0

# equals()

---

- ❖ A expressão `c1 == c2` verifica se as referências `c1` e `c2` apontam para o mesmo objeto
  - Caso `c1` e `c2` sejam variáveis automáticas a expressão anterior compara valores
- ❖ O método `equals()` testa se dois objetos são iguais

```
Circulo p1 = new Circulo(0, 0, 1);
Circulo p2 = new Circulo(0, 0, 1);
System.out.println(p1 == p2);           // false
System.out.println(p1.equals(p2));      // false (porquê?)
```
- ❖ `equals()` deve ser redefinido sempre que os objetos dessa classe puderem ser comparados
  - Circulo, Ponto, Complexo ...

# Problemas com equals()

---

## ❖ Propriedades da igualdade

- reflexiva:  $x.equals(x) \rightarrow true$
- simétrica:  $x.equals(y) \leftrightarrow y.equals(x)$
- transitiva:  $x.equals(y) \text{ AND } y.equals(z) \rightarrow x.equals(z)$

## ❖ Devemos respeitar a assinatura `Object.equals(Object o)`

```
public class Circulo {  
    //...  
    @Override  
    public boolean equals(Object obj) { //...  
    }  
}
```

## ❖ Problemas

- E se 'obj' for null?
- E se referenciar um objeto diferente de Circulo?

# Circulo.equals()

---

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Circulo other = (Circulo) obj;
    // verify if the object's attributes are equals
    if (centro == null) {
        if (other.centro != null)
            return false;
    } else if (!centro.equals(other.centro))
        return false;
    if (raio != other.raio)
        return false;
    return true;
}
```

# equals() em Herança

---

```
class Base {
    private int x;
    public Base ( int i ) { x = i; }
    public boolean equals( Object rhs ) {
        if ( rhs == null ) return false;
        if ( getClass() != rhs.getClass() ) return false;
        if ( rhs == this ) return true;
        return x == ( (Base) rhs ).x;
    }
}

class Derived extends Base {
    public Derived ( int i, int j ) { super( i ); y = j; }
    public boolean equals( Object rhs ) {
        // Não é necessário testar a classe. Feito em Base
        return super.equals(rhs) && y== ( (Derived) rhs ).y;
    }
    private int y;
}
```

# hashCode()

- ❖ Sempre que o método `equal()` for reescrito, `hashCode()` também deve ser
  - Objetos iguais devem retornar códigos de hash iguais
- ❖ O objetivo do hash é ajudar a identificar qualquer objeto através de um número inteiro

// Circulo.hashCode() – Exemplo muito simples !!!

```
public int hashCode() {  
    return raio * centro.x() * centro.y();  
}
```

//..

```
Circulo c1 = new Circulo(10,15,27);
```

```
Circulo c2 = new Circulo(10,15,27);
```

```
Circulo c3 = new Circulo(10,15,28);
```

4050

4050

4200

- A construção de uma boa função de hash não é trivial.  
Para a sua construção recomendam-se outras fontes

# Circulo.hashCode()

---

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = prime
        + ((centro == null) ? 0 : centro.hashCode());
    long temp = Double.doubleToLongBits(raio);
    result = prime * result + (int) (temp ^ (temp >>> 32));
    // ^   Bitwise exclusive OR
    // >>> Unsigned right shift
    return result;
}
```

# Sumário - Porquê herança?

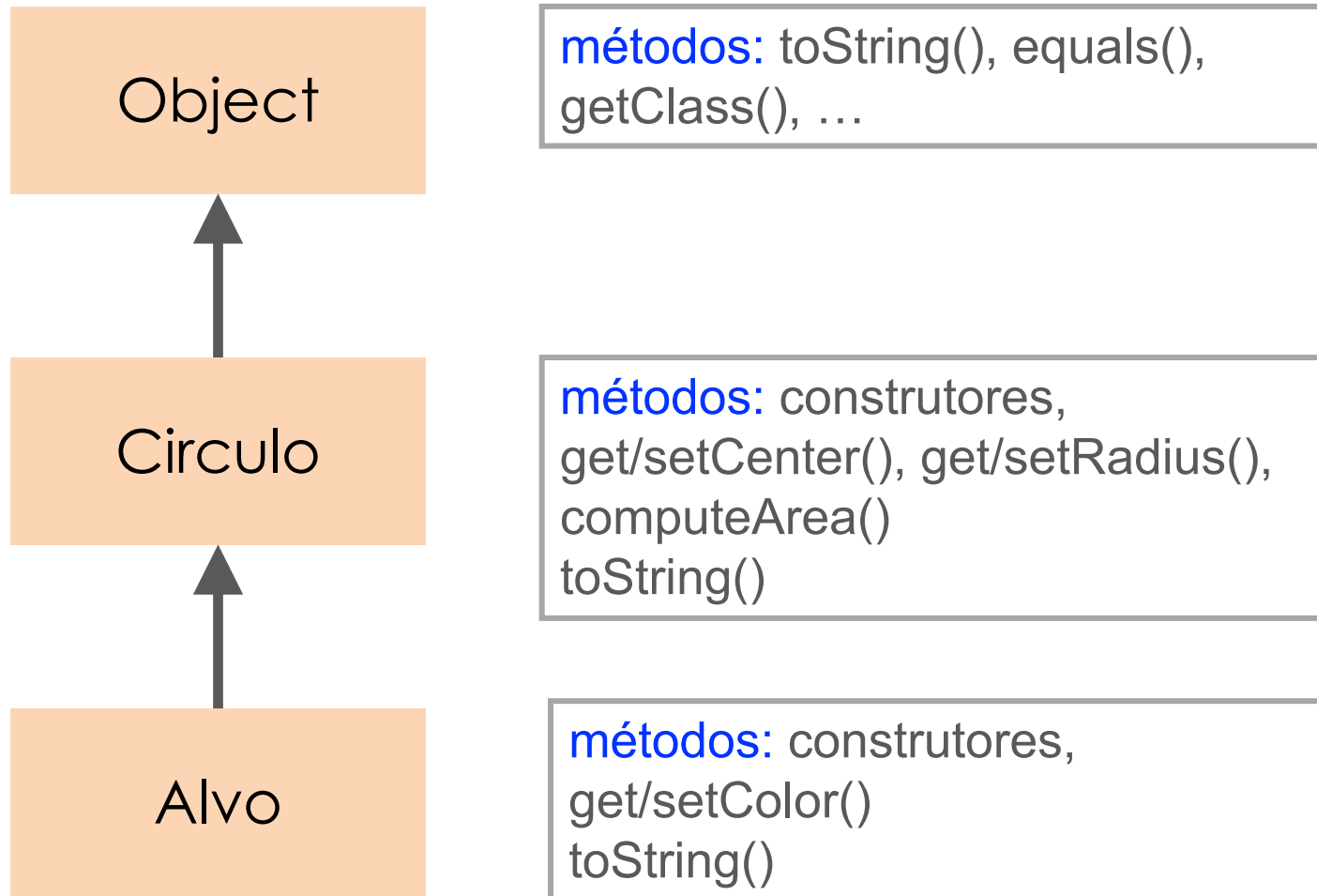
---

- ❖ Muitos objetos reais apresentam esta característica
- ❖ Permite criar classes mais simples com funcionalidades mais estanques e melhor definidas
  - Devemos evitar classes com interfaces muito "extensas"
- ❖ Permite reutilizar e estender interfaces e código
- ❖ Permite tirar partido do polimorfismo



# 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

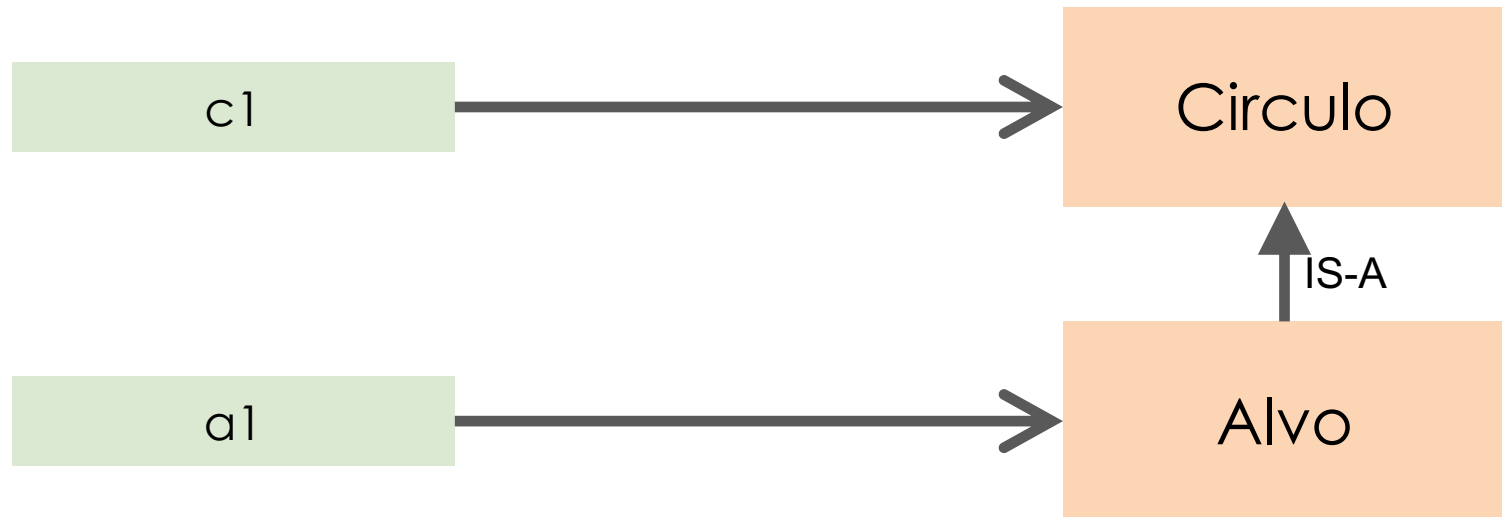
```
Circulo c1 = new Alvo(...);  
Object obj = new Circulo(...);
```

## ❖ Referência polimórfica

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

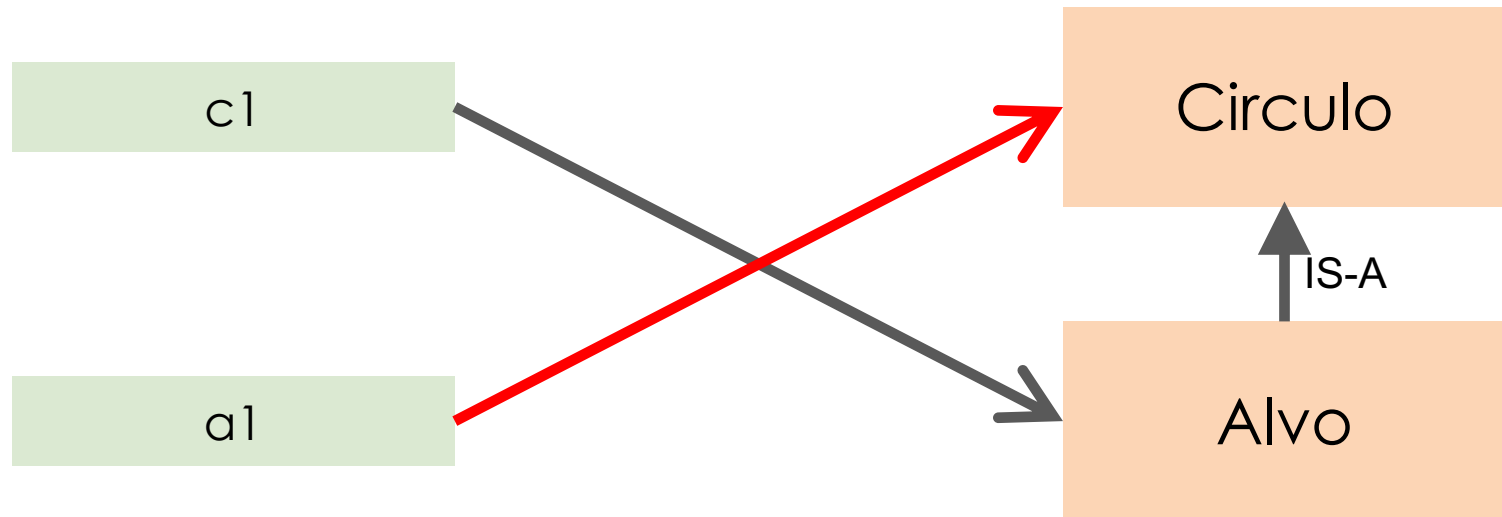
# Polimorfismo - exemplos

```
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.

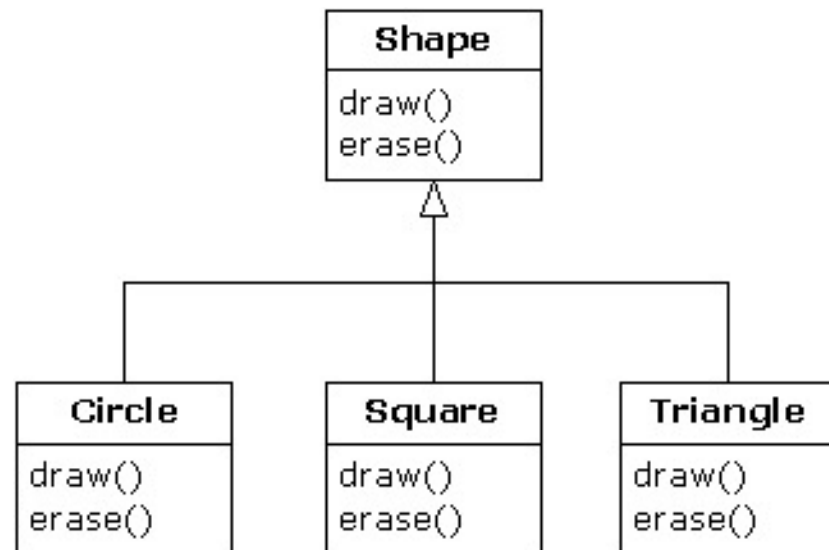


# Exemplo 1

```
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;
```

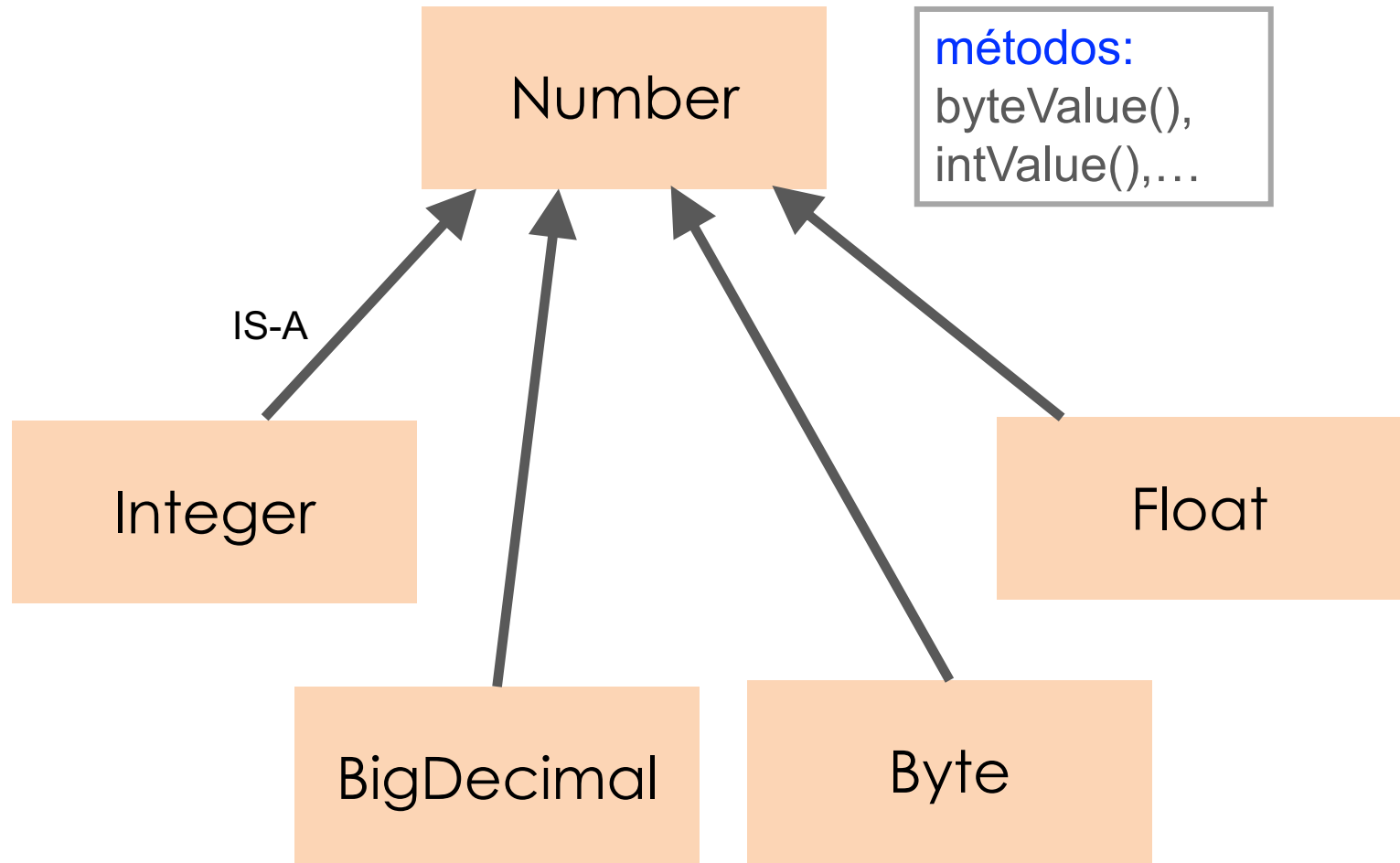
- ❖ Abstrair implementações diferentes para operações semelhantes em classes abstratas num nível superior.

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

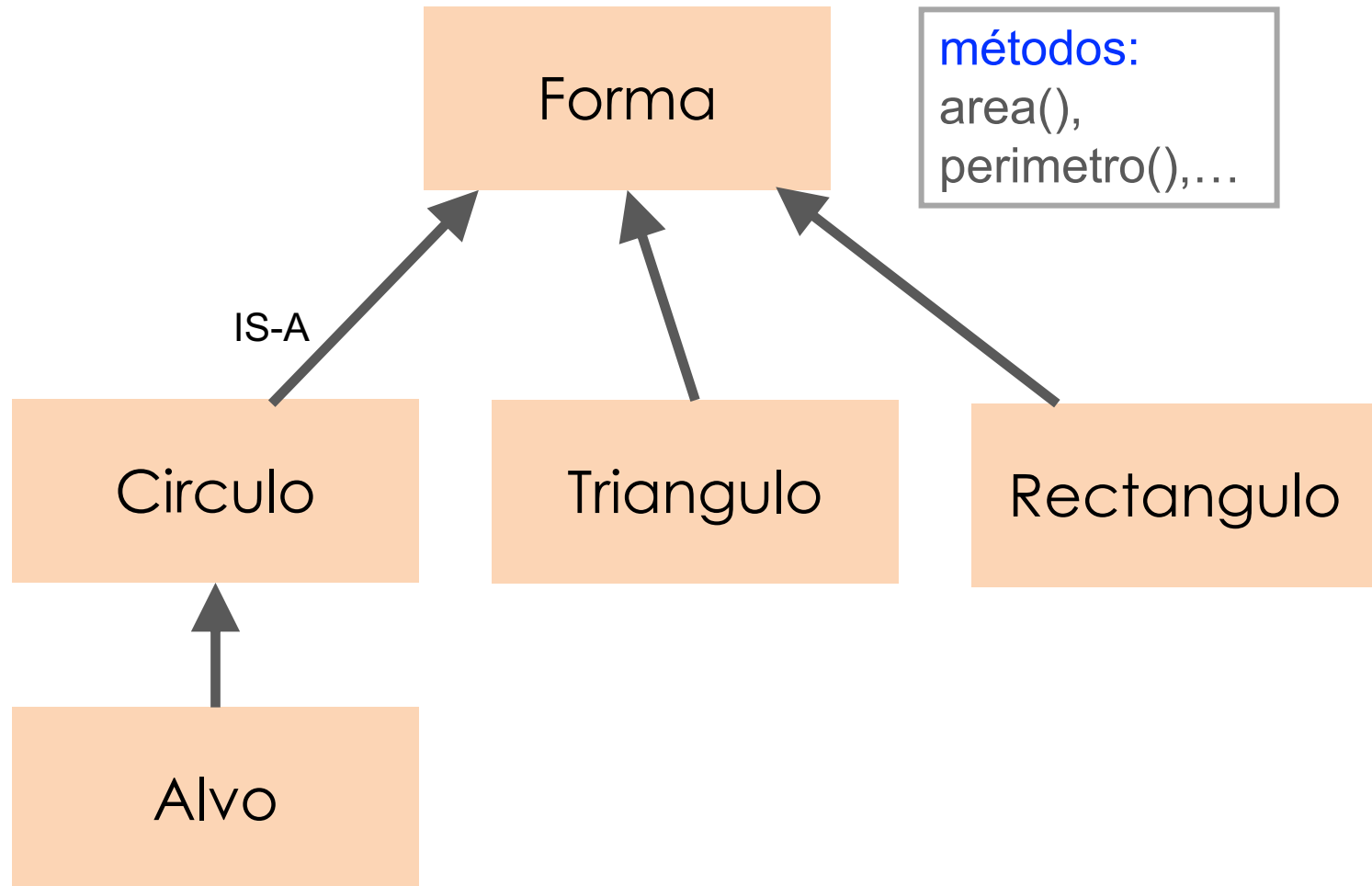
- ❖ Reunir comportamentos e características e fazê-los subir o mais possível na hierarquia de classes.

```
ZooAnimal.peso;
```

# Exemplo de herança

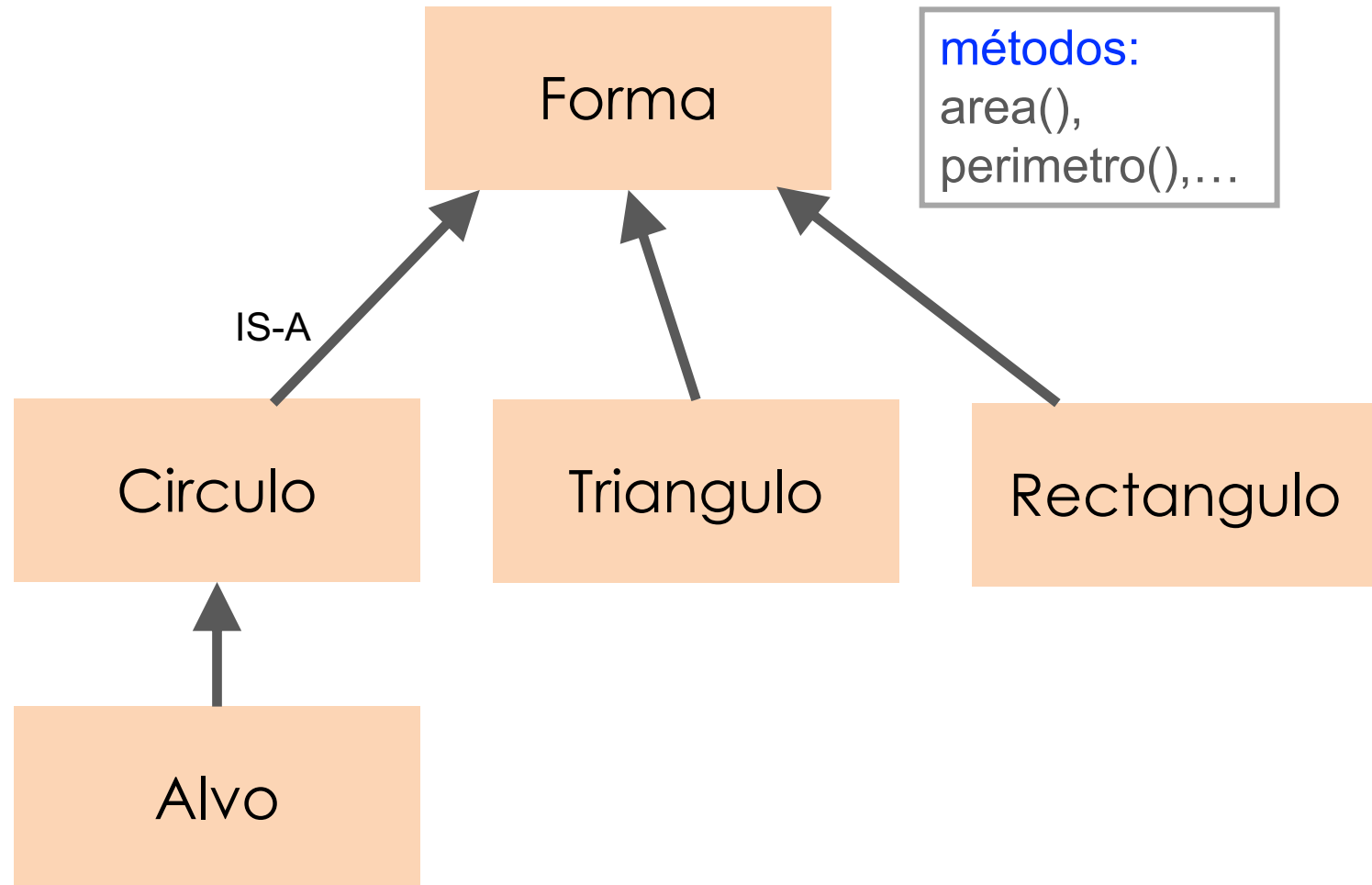


# Exemplo de herança



# 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 referência para 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) {...}  
}
```

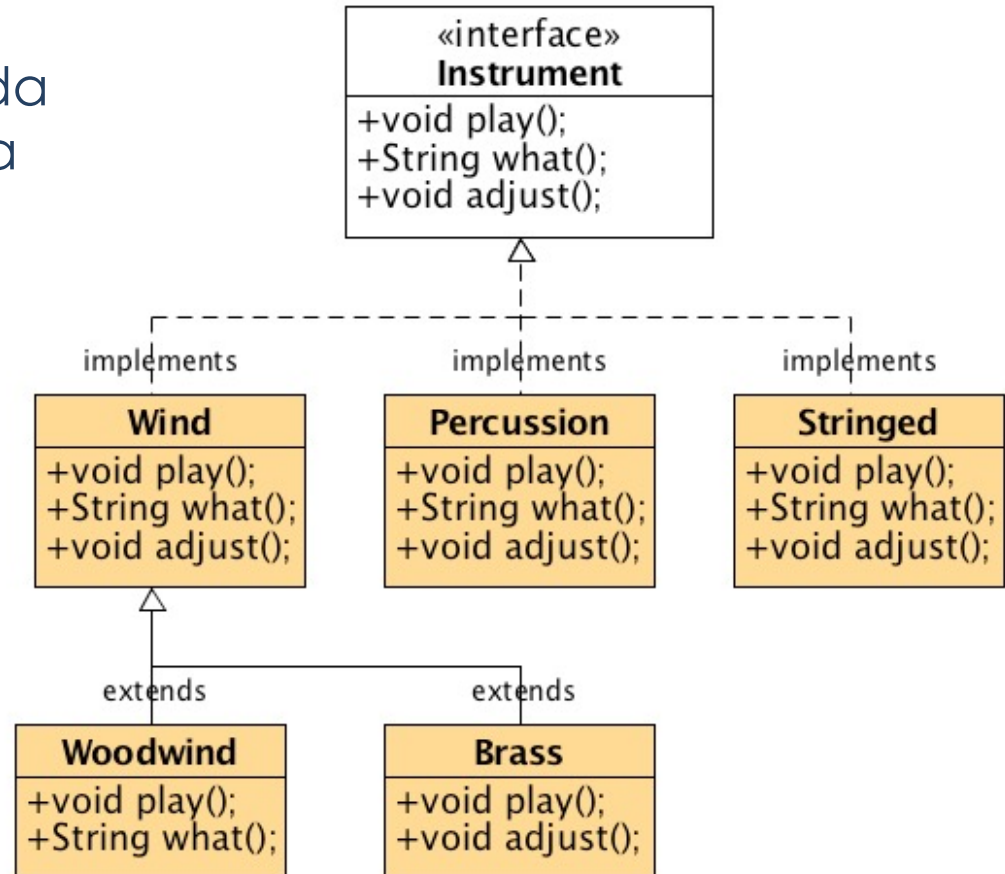
# Características principais

---

- ❖ 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() { /* .. */ }  
}
```

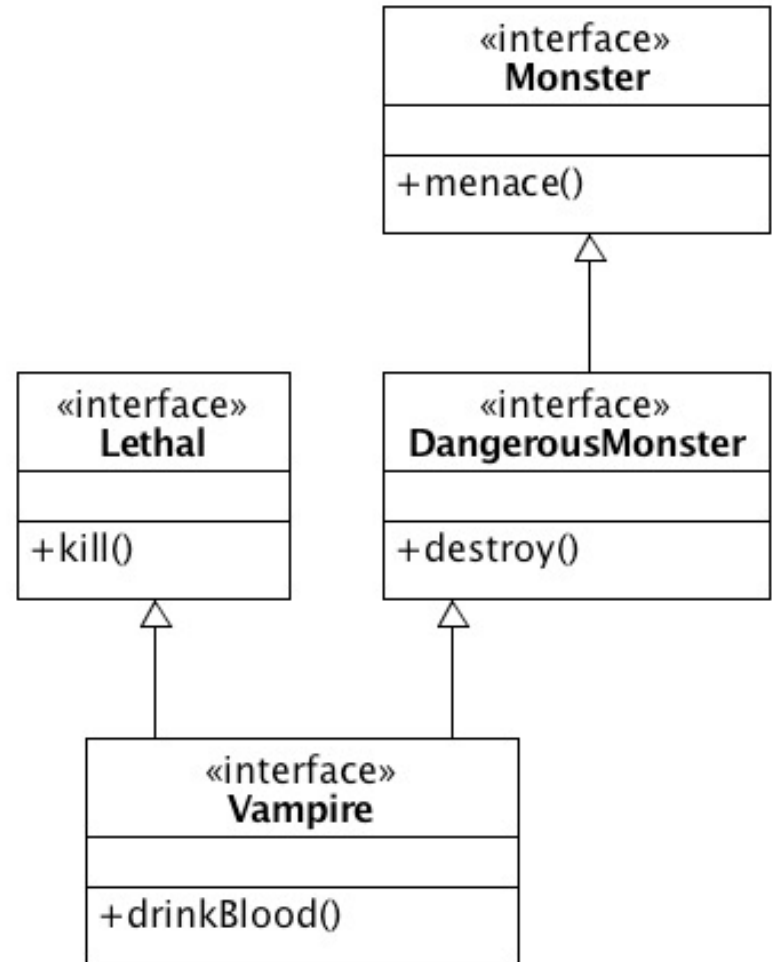
# Herança em Interfaces

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

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

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

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





# Interfaces em 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();  
        // ...  
    }  
}
```

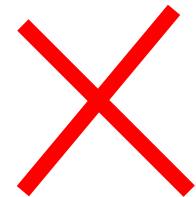
# Interfaces a partir de Java 8

## ❖ Static Methods

- Similares aos default methods
- Não podem ser reescritos nas classes que implementam a interface

```
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

---

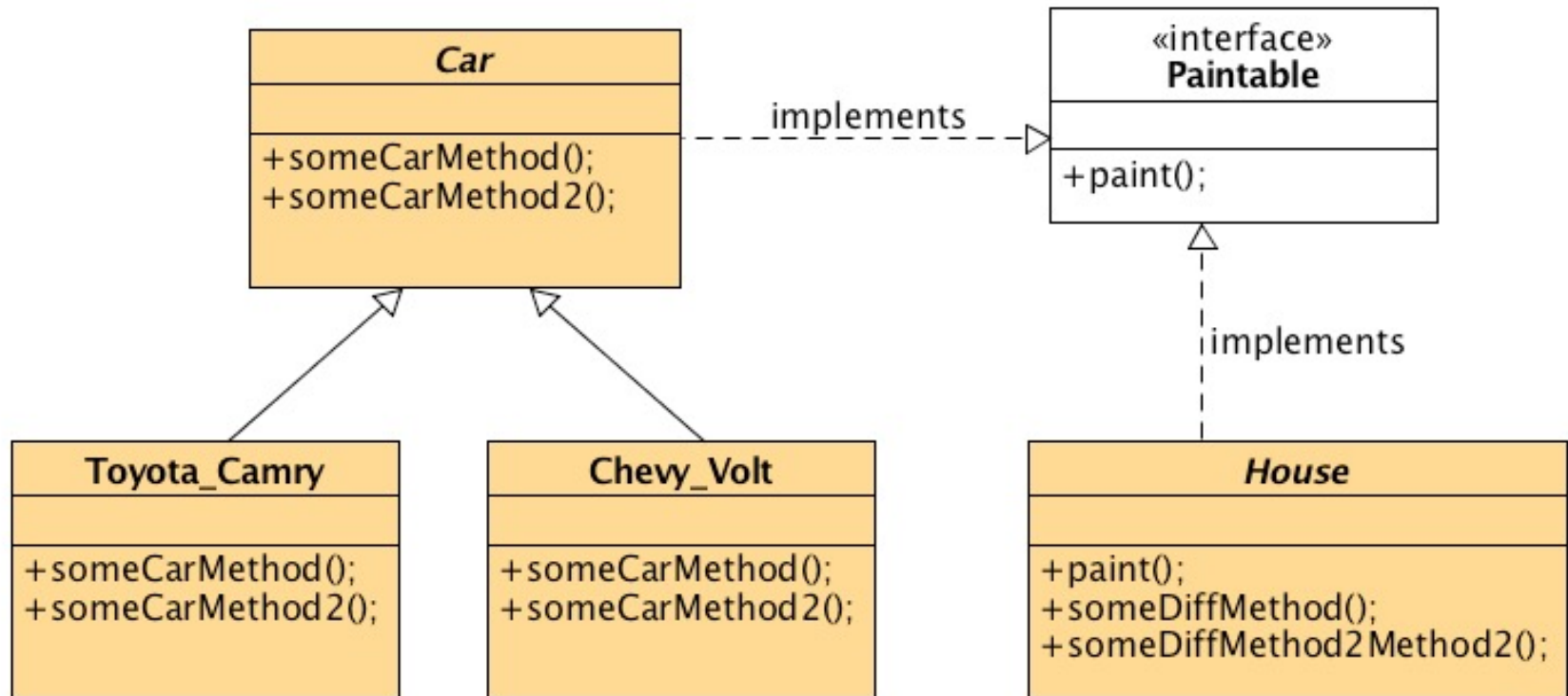
## Classes Abstratas

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

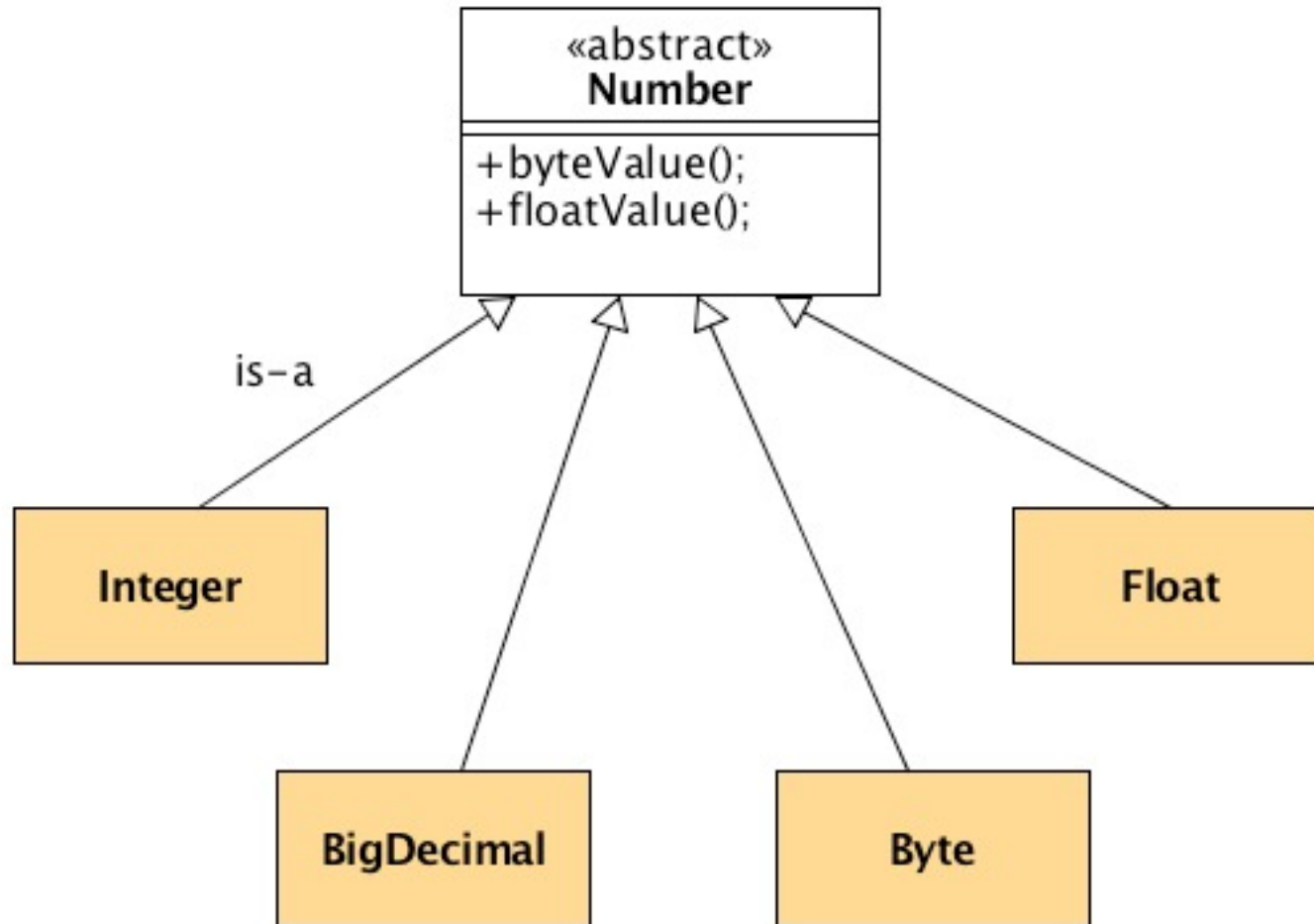
## Interfaces

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

# Classes Abstratas versus Interfaces

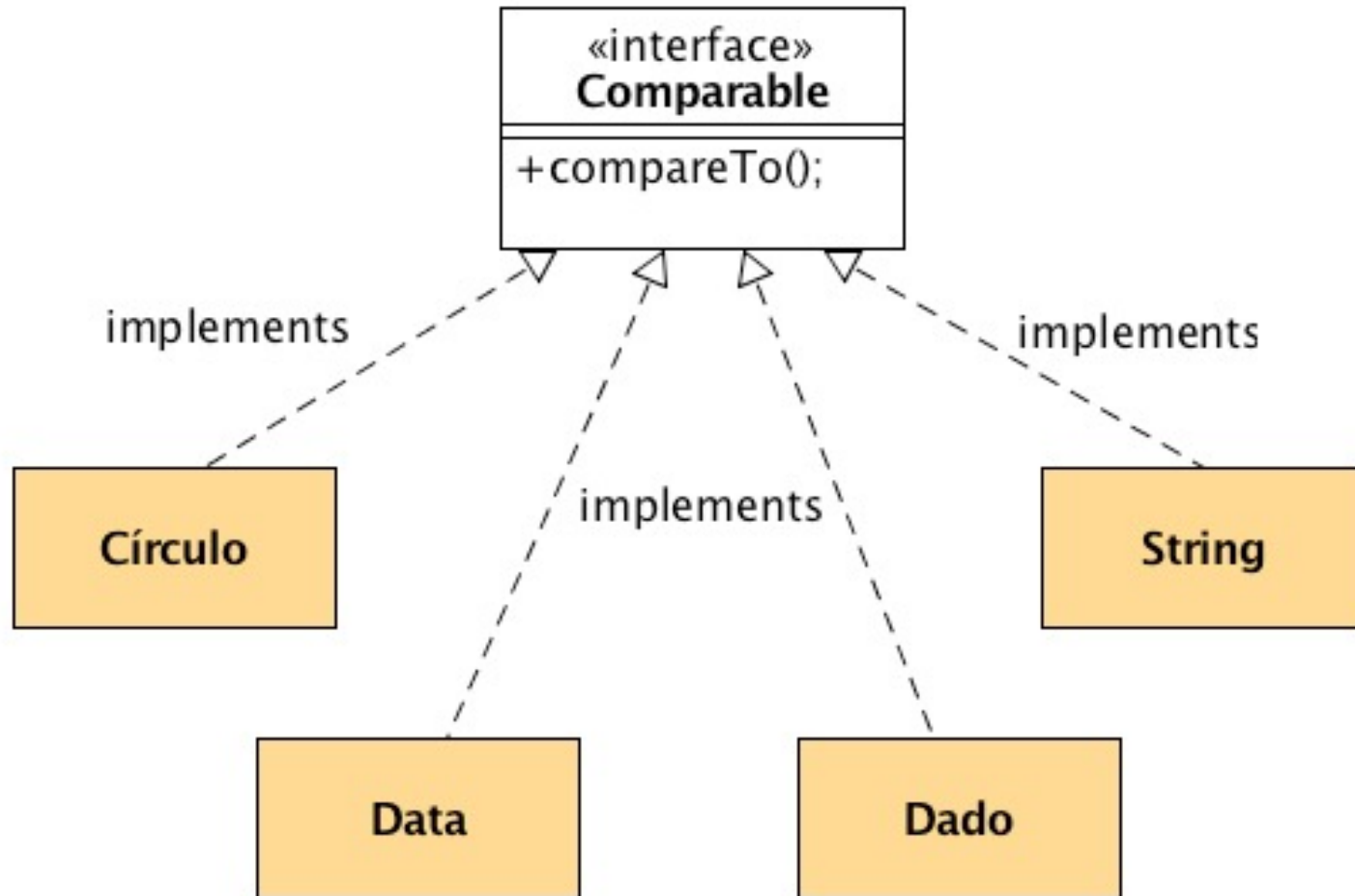


# Classes Abstratas versus Interfaces



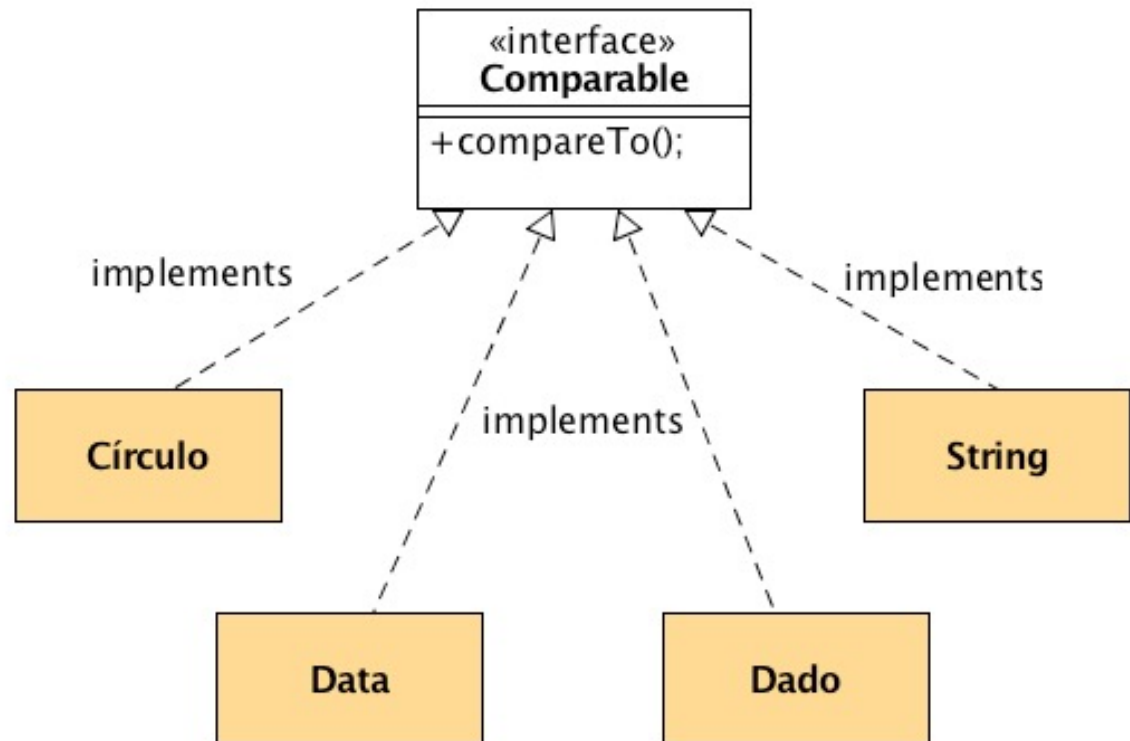


# Classes Abstratas versus Interfaces



# 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

---

- ❖ Instrução 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