

Java Herança

UA, DETI, Programação III
José Luis Oliveira, Carlos Costa
2014/15

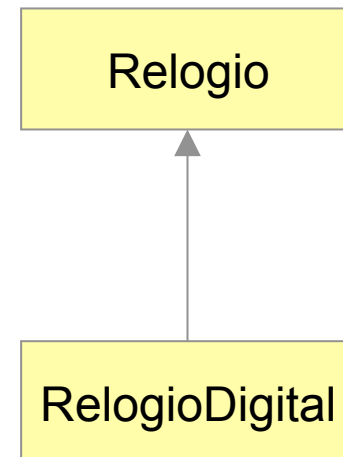
Relações entre Classes

- 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.

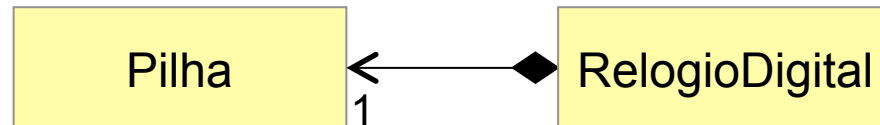
```
class Relogio {  
    /* ... */  
}  
  
class RelogioDigital extends Relogio {  
    /* ... */  
}
```



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 aspectos comuns (dados, comportamento)
 - Possuem aspectos distintos
 - Uma é uma especialização da outra
- Exemplos:
 - Rato é um Mamífero
 - BTT é uma Bicicleta
 - Cerveja é uma Bebida
 - ... e Sagres é uma Cerveja?

Questões?

- Quais as relações entre:
 1. Empregado, Motorista, Vendedor, Administrativo e Contabilista
 2. Quadrado, Triângulo, Retângulo, e Losango
 3. Professor, Aluno e Funcionário
 4. Ferrari, Carro, Roda, Motor, Pneu, Jante

Questões?

- Modelar stock de uma livraria...
 - Livro
 - Artigo
 - Jornal
 - Publicação
 - Autor
 - Periódico
 - Editora
 - LivroEditado
 - Revista

Questões?

- Modelar os *gadgets* de casa...
 - Telemóvel
 - Reprodutor de Áudio
 - Bateria
 - Carregador
 - MP3
 - Auscultador
 - Calculadora

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 herda todos os 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
- Terminologia
 - B é a classe Base / A é derivada de B
 - B é a classe Mãe (Parent) / A é a classe Filha (Child)
 - B é a classe Super / A é a classe Sub

Herança - Exemplo

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

Base

```
class Student extends Person {  
    private int nmec;  
    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

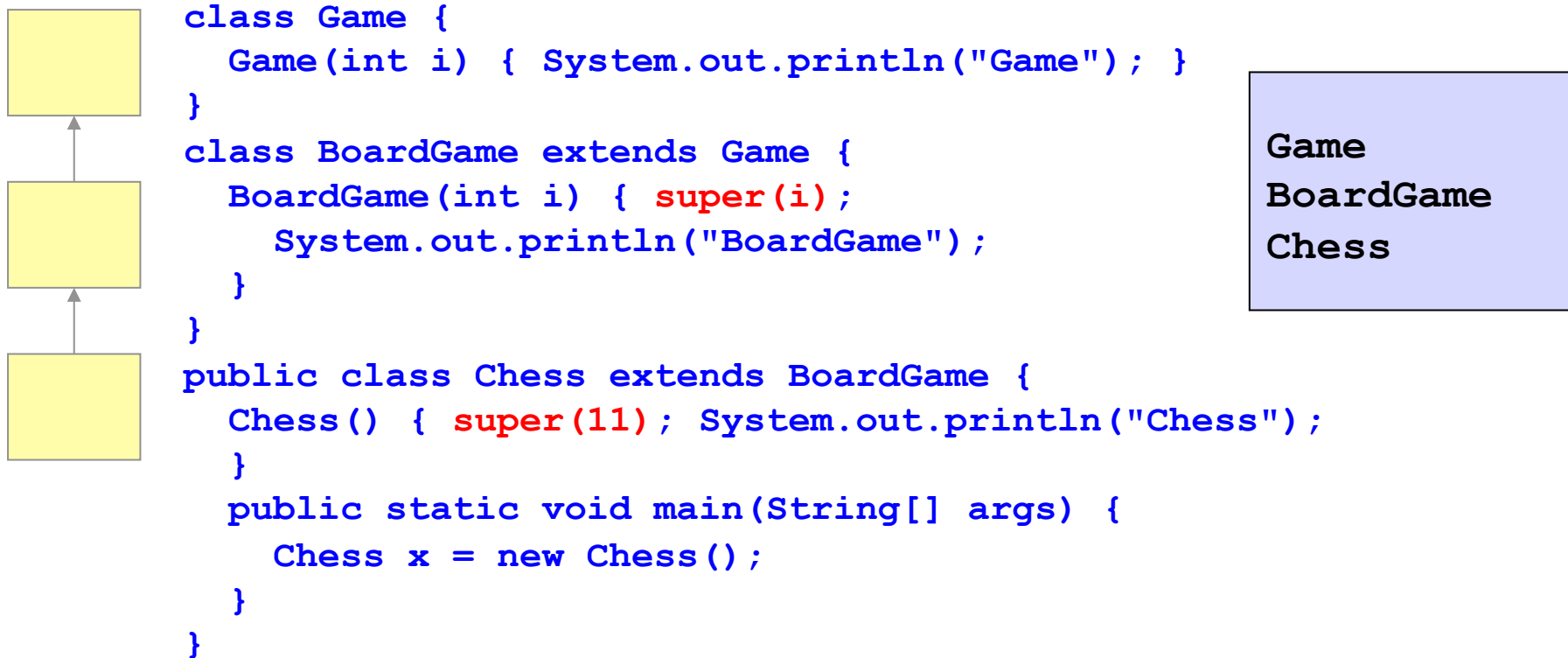
Drawing constr.

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.



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

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

```
class Student extends Person {  
    private int nmec;  
    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

- Métodos declarados como *public* na classe base também 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) podem manter ou ser *private* em subclasses
- Métodos declarados como *private* não são herdados pelo que não se aplicam as regras de visibilidade em subclasses

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 arrays
 - 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
    final int i1 = 9;
    static final int VAL_TWO = 99;
    // Typical public constant:
    public static final int VAL_THREE = 39;
    // Cannot be compile-time constants:
    final int i4 = (int)(Math.random()*20);
    static final int i5 = (int)(Math.random()*20);

    Value v1 = new Value();
    final Value v2 = new Value();
    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];
    }
}

```

Final - Dados

- Os dados final podem ser inicializados dentro do construtor

```
class Dummy { }

class BlankFinal {
    final int i = 0; // Initialized final
    final int j; // Blank final
    final Dummy p; // Blank final reference
    // Blank finals MUST be initialized in the
    constructor:

    BlankFinal() {
        j = 1; // Initialize blank final
        p = new Dummy();
    }

    BlankFinal(int x) {
        j = x; // Initialize blank final
        p = new Dummy();
    }
}
```

Final - Argumentos

- A associação de "final" a argumentos de métodos garante que essas referências não serão alteradas dentro do método.

```
class Gizmo {  
    public void spin() {}  
}  
  
public class FinalArguments {  
    void with(final Gizmo g) {  
        //! g = new Gizmo(); // Illegal -- g is final  
        g.spin();  
    }  
    void without(Gizmo g) {  
        g = new Gizmo(); // OK -- g not final  
        g.spin();  
    }  
    // void f(final int i) { i++; } // Can't change  
}
```

Herança - Exemplo

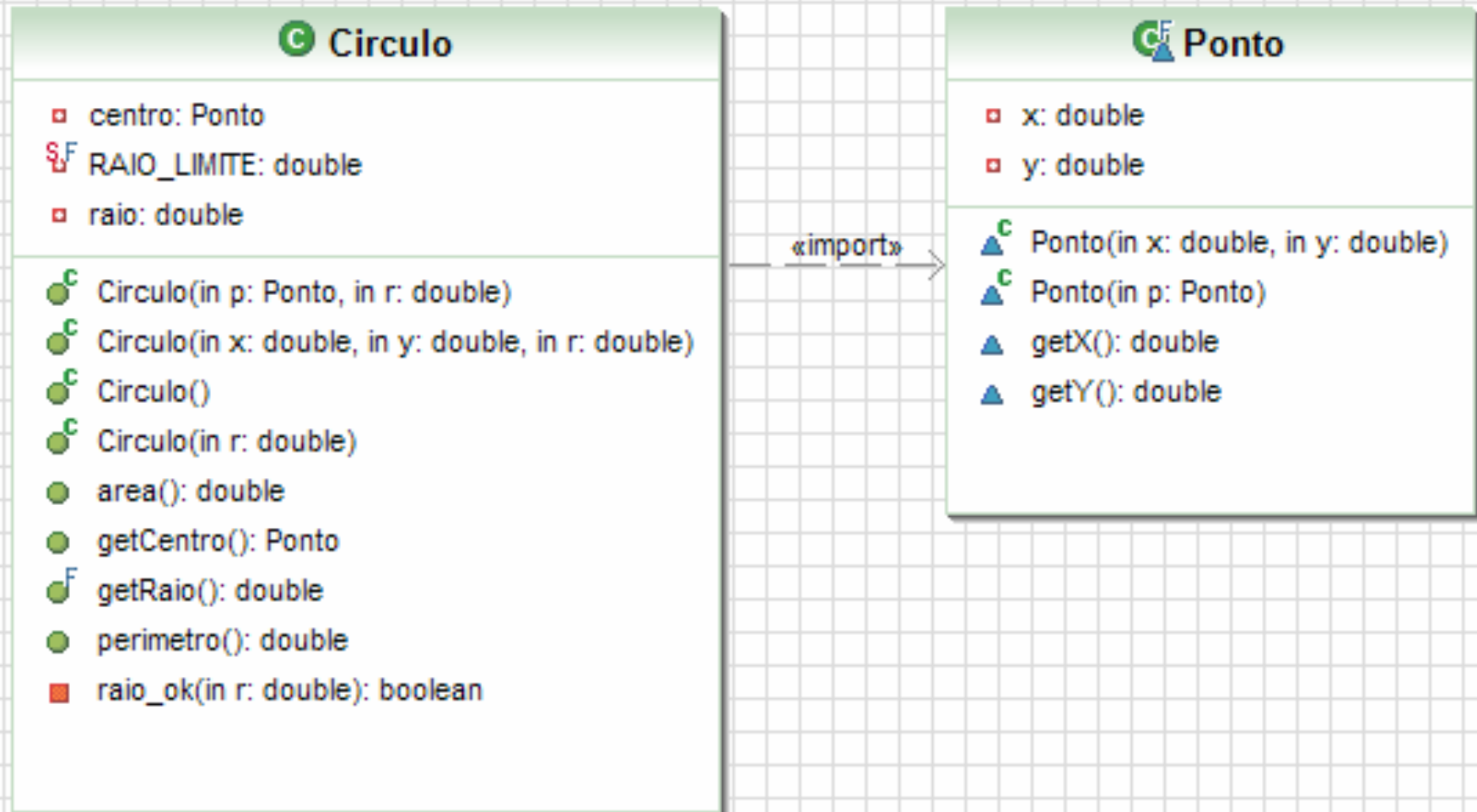
```
public final class Ponto {
    private double x;
    private double y;

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

public class Circulo {
    private Ponto centro;
    private double raio;

    public static final double RAIIO_LIMITE = 100.0;
    private boolean raio_ok(double r) { return(r<=RAIO_LIMITE); }
    public Circulo(Ponto p, double r) {
        centro = p;
        if (raio_ok(r)) raio = r; else raio = RAIIO_LIMITE;
    }
    public Circulo(double x, double y, double r) { this(new Ponto(x, y), r); }
    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; }
}
```

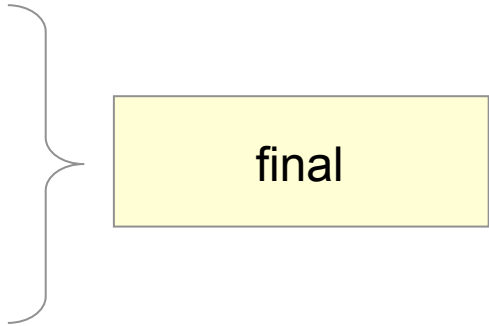
Representação UML



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

- Todas as classe em Java derivam da super classe `java.lang.Object`
 - Métodos
 - `toString()`
 - `equals()`,
 - `hashCode()`
 - `finalize()`
 - `clone()`
 - `getClass()`
 - `wait()`
 - `notify()`
 - `notifyAll()`
- 
- A diagram consisting of a right-facing curly bracket on the left side, grouping the last six methods in the list: `getClass()`, `wait()`, `notify()`, and `notifyAll()`. To the right of the bracket is a yellow rectangular box with a black border containing the word "final".

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

```
String s1 = "Aveiro";  
String s2 = "Aveiro";  
System.out.println(s1 == s2);           // true (porquê?)  
System.out.println(s1.equals(s2));      // true  
Ponto p1 = new Ponto(1, 1);  
Ponto p2 = new Ponto(1, 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: $x.equals(x) \rightarrow \text{true}$
 - simétrica: $x.equals(y) \leftrightarrow y.equals(x)$
 - transitiva: $x.equals(y) \text{ AND } y.equals(z) \rightarrow x.equals(z)$
- Devemos respeitar o contrato '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;  
    if (centro == null) {  
        if (other.centro != null)  
            return false;  
    } else if (!centro.equals(other.centro))  
        return false;  
    if (Double.doubleToLongBits(raio) != Double  
        .doubleToLongBits(other.raio))  
        return false;  
    return true;  
}
```

equals() em Herança

```
class BaseClass {
    public BaseClass( 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 == ( (BaseClass) rhs ).x;
    }
    private int x;
}

class DerivedClass extends BaseClass {
    public DerivedClass( 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 == ( (DerivedClass) 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 objectivo do hash é ajudar a identificar qualquer objeto através de um número inteiro
 - Usado em HashTables

```
// 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;
}
```

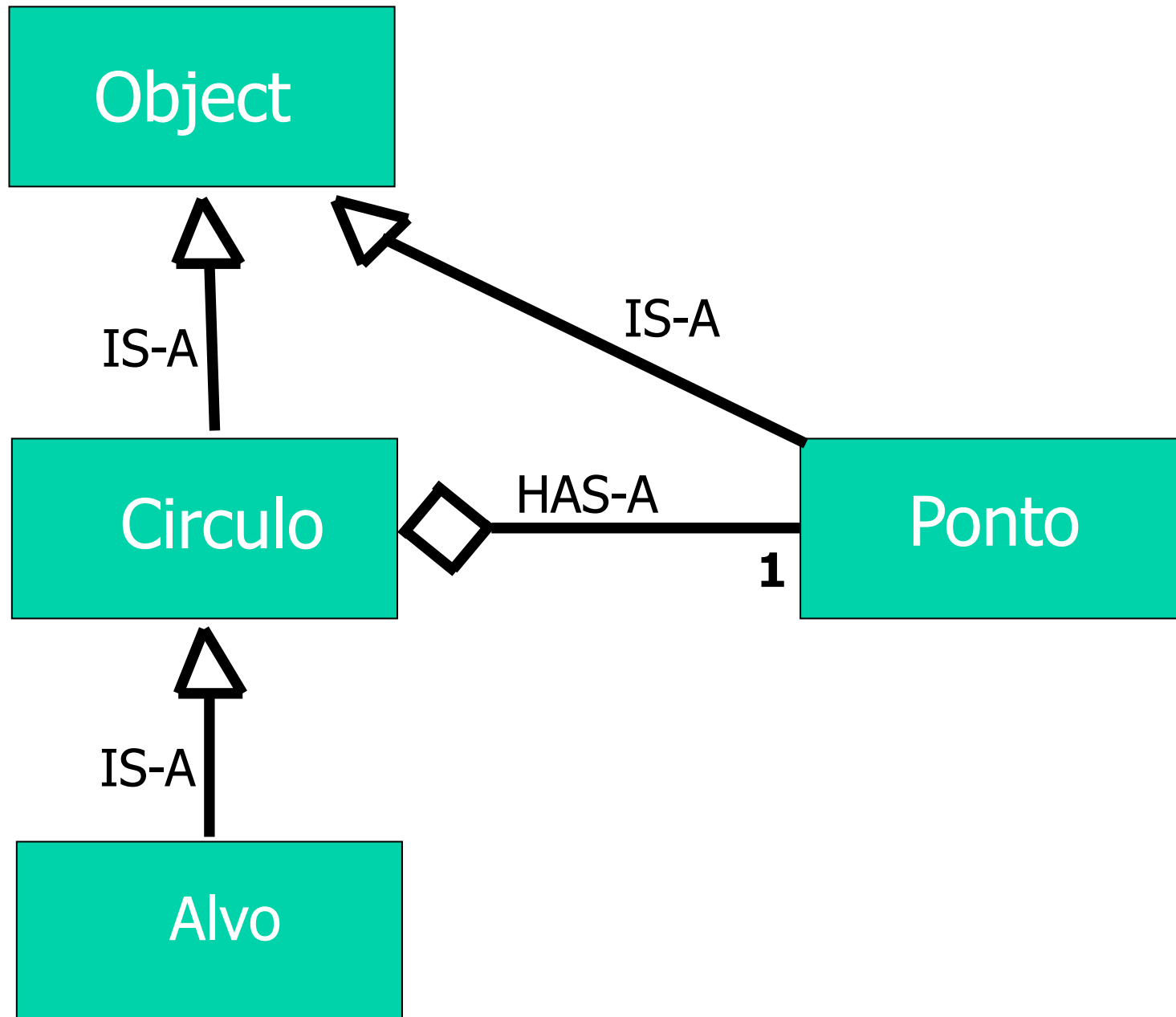
Sumário - Porquê herança?

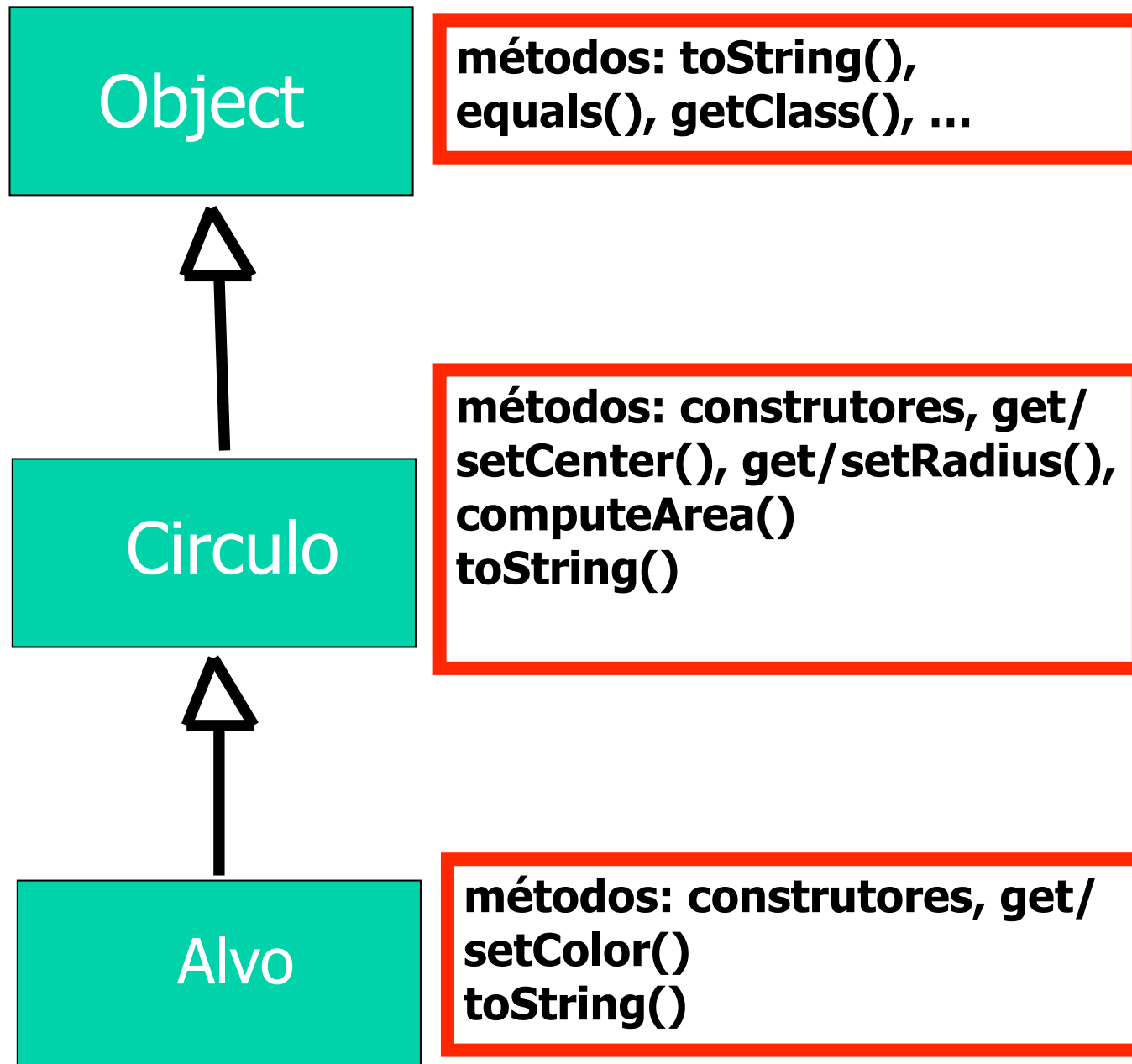
- Muitos objetos reais apresentam esta característica
- Permite criar classes mais simples com funcionalidades mais estanques e melhor definidas
 - Devemos evitar classes com interfaces muito "extensas"
- Permite reutilizar e estender interfaces e código
- Permite tirar partido do polimorfismo

Java

Polimorfismo

UA, DETI, Programação III
José Luis Oliveira, Carlos Costa
2014/15





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;
```

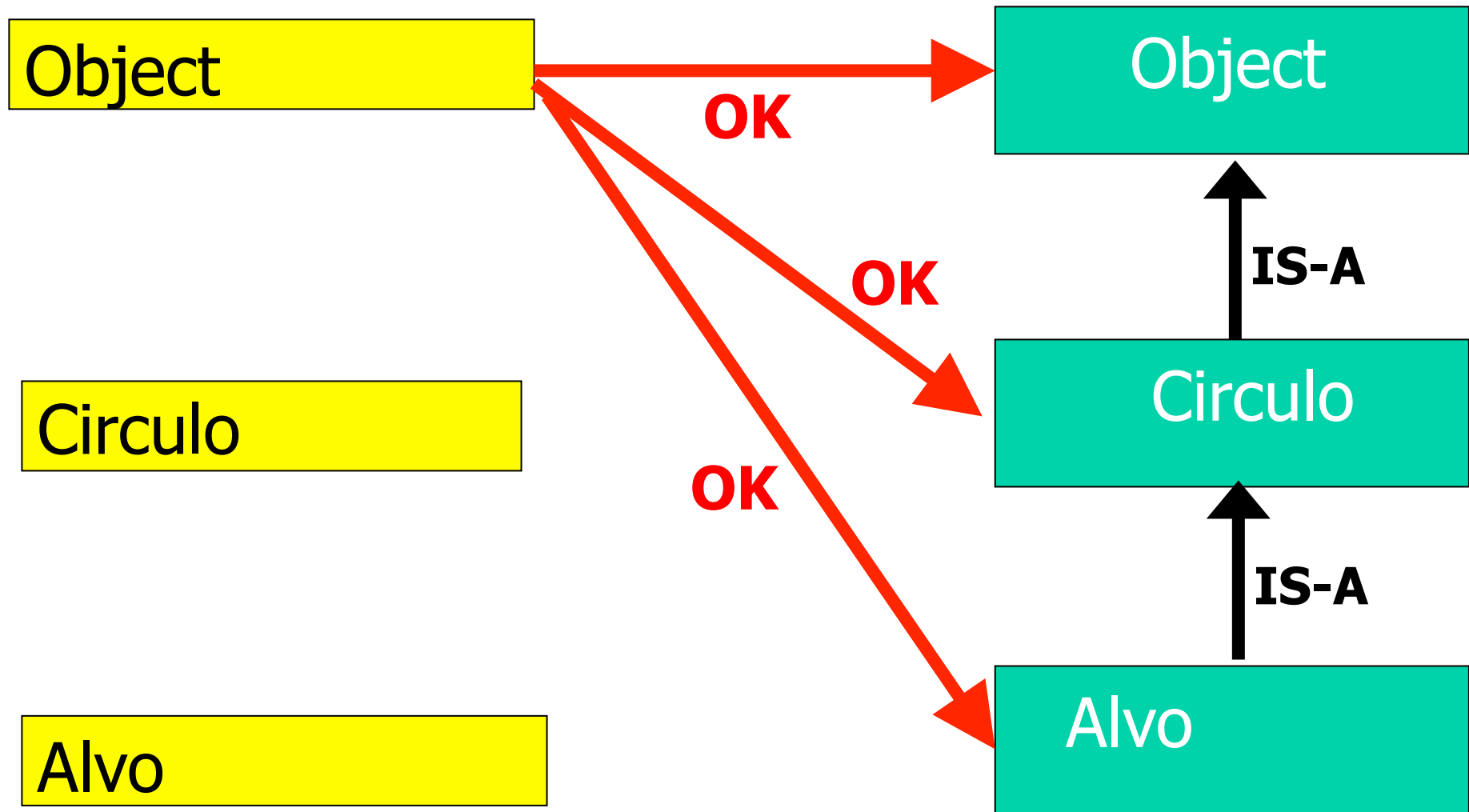
OK

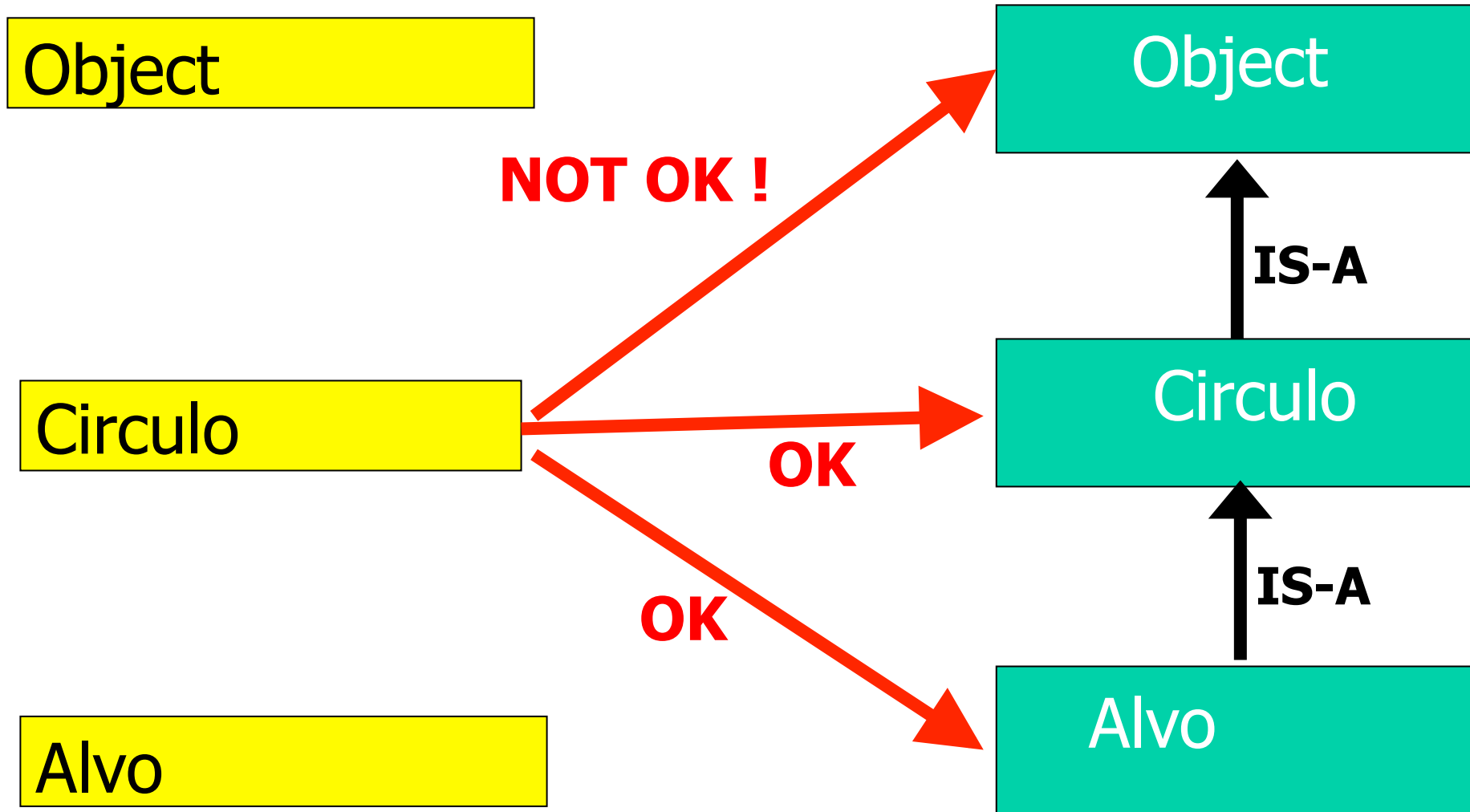
Polimorfismo

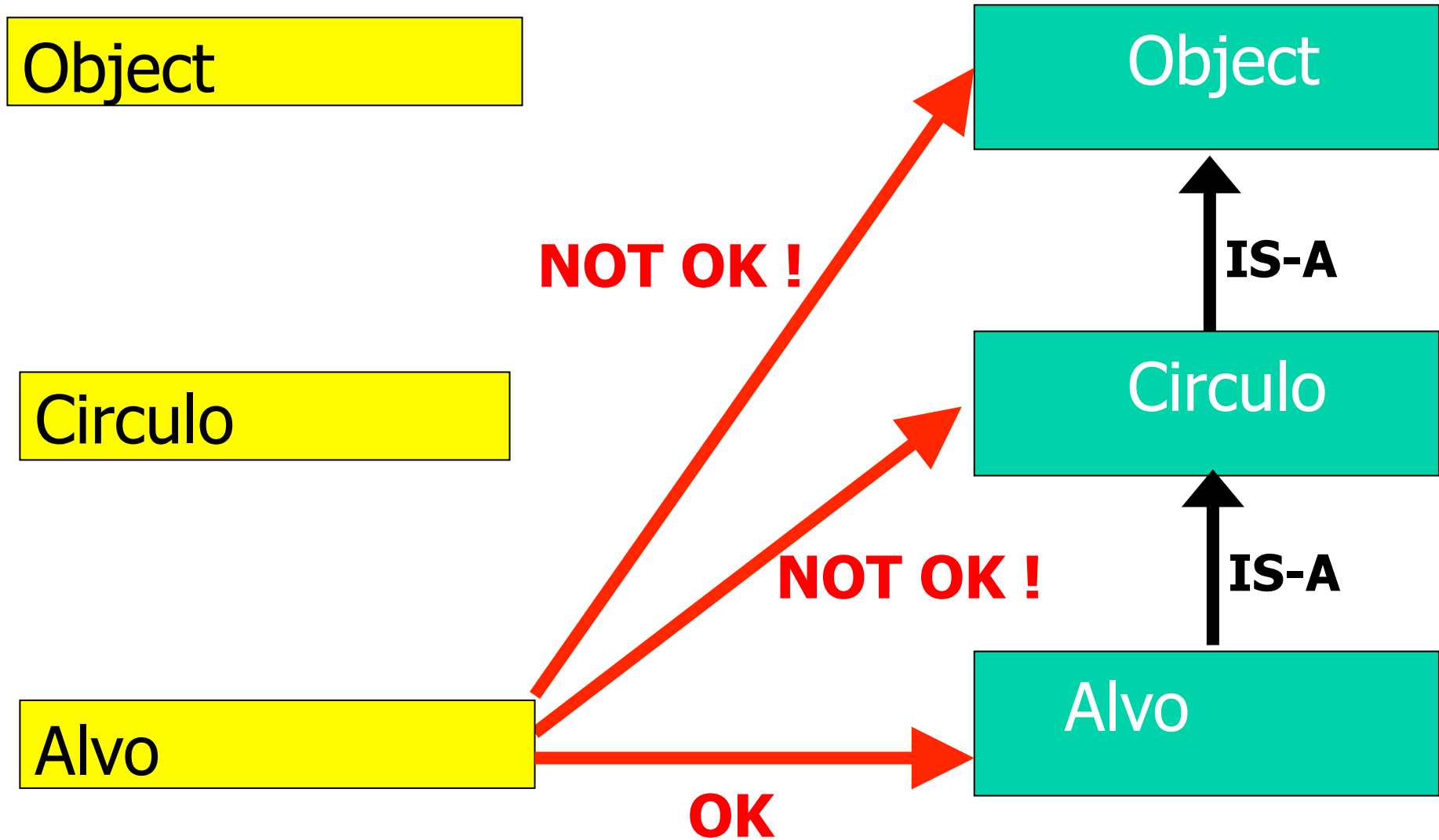
- Ideia base:
 - o tipo declarado na referência não precisa de ser exatamente o mesmo tipo do objeto para o qual aponta - pode ser de qualquer tipo derivado

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

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







Polimorfismo

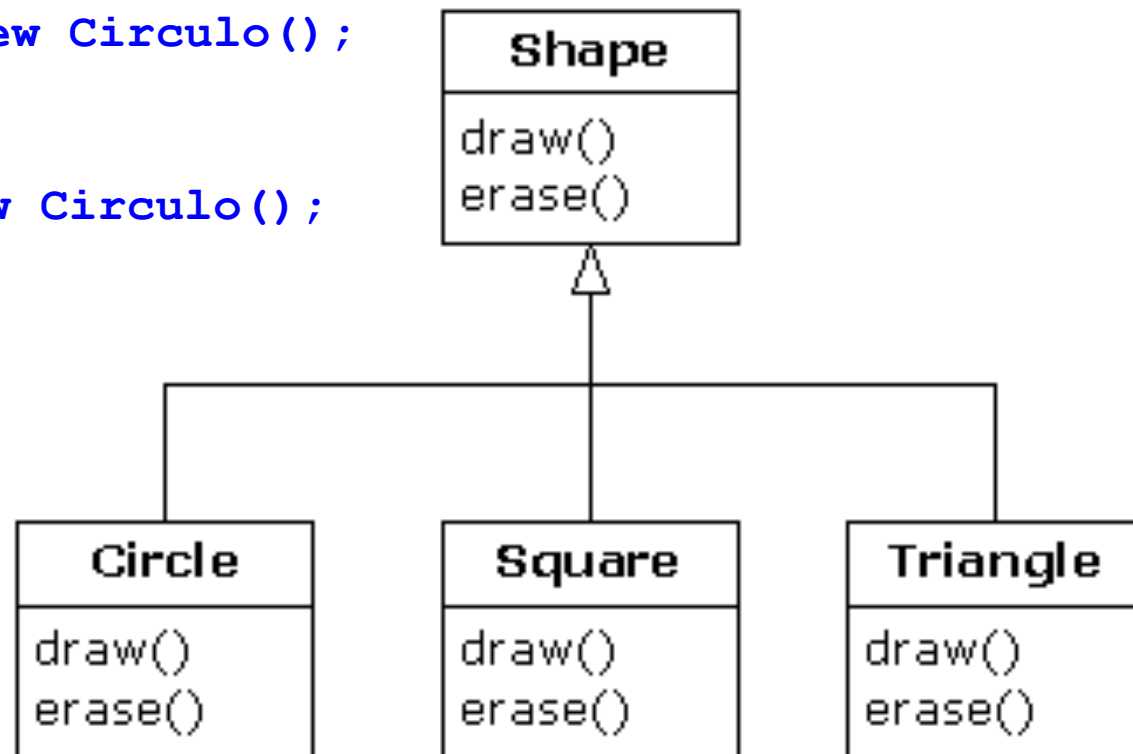
- Polimorfismo é, conjuntamente com a Herança e o Encapsulamento, uma das características fundamentais da POO.
 - Formas diferentes com interfaces semelhantes.
- Outras designações:
 - Ligação dinâmica (Dynamic binding), late binding ou run-time binding
- Esta característica permite-nos tirar mais partido 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 excepção dos *final*) 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 com o 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() {} }

class Circle extends Shape {
    void draw() {    System.out.println("Circle.draw()");
}

class Square extends Shape {
    void draw() {    System.out.println("Square.draw()");
}

public class Shapes {
    public static Shape randShape() {
        switch((int) (Math.random() * 2)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
        }
    }

    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        for(int i = 0; i < s.length; i++)
            s[i] = randShape(); // Fill up the array with shapes:
        for(int i = 0; i < s.length; i++)
            s[i].draw(); // Make polymorphic method calls:
    }
}
```

Circulo.draw()
Circulo.draw()
Circulo.draw()

Circulo.draw()
Square.draw()
Square.draw()
Square.draw()

Square.draw()
Square.draw()
Square.draw()
Square.draw()
Circulo.draw()
Circulo.draw()
Square.draw()
Square.draw()
Circulo.draw()
Square.draw()

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.

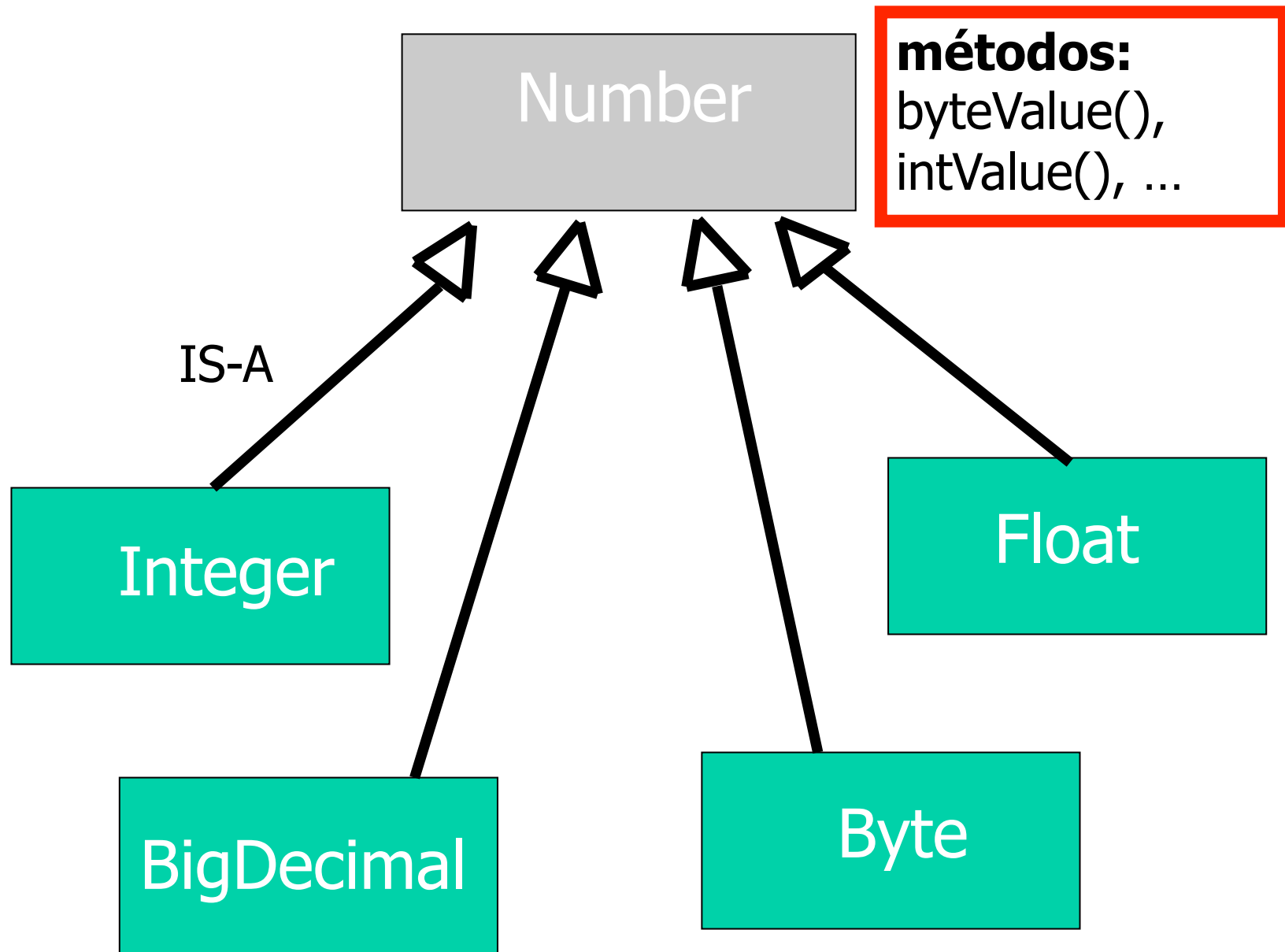
```
class ZooAnimal;
```

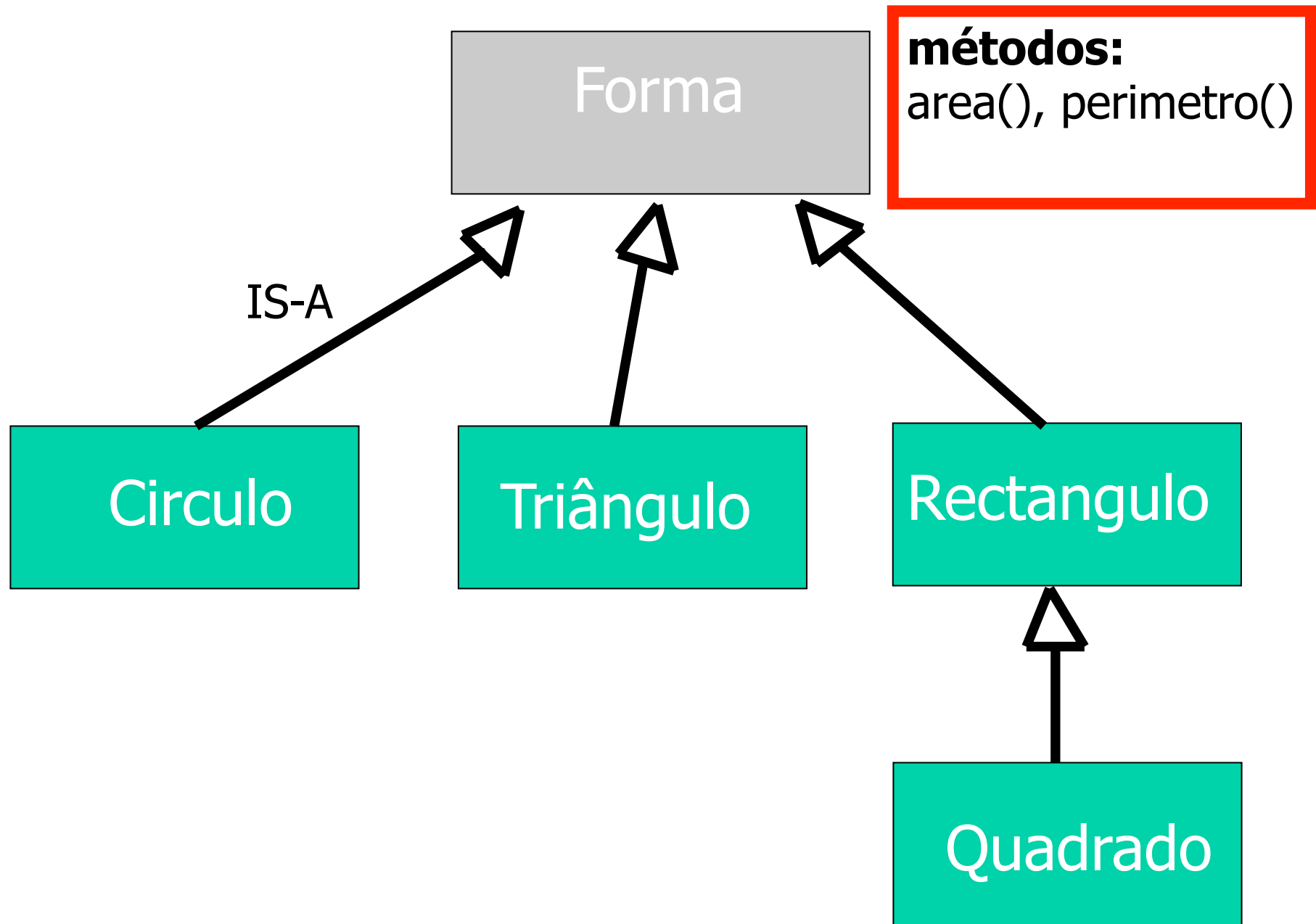
- Abstrair implementações diferentes para operações semelhantes em classes abstractas num nível superior.

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

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

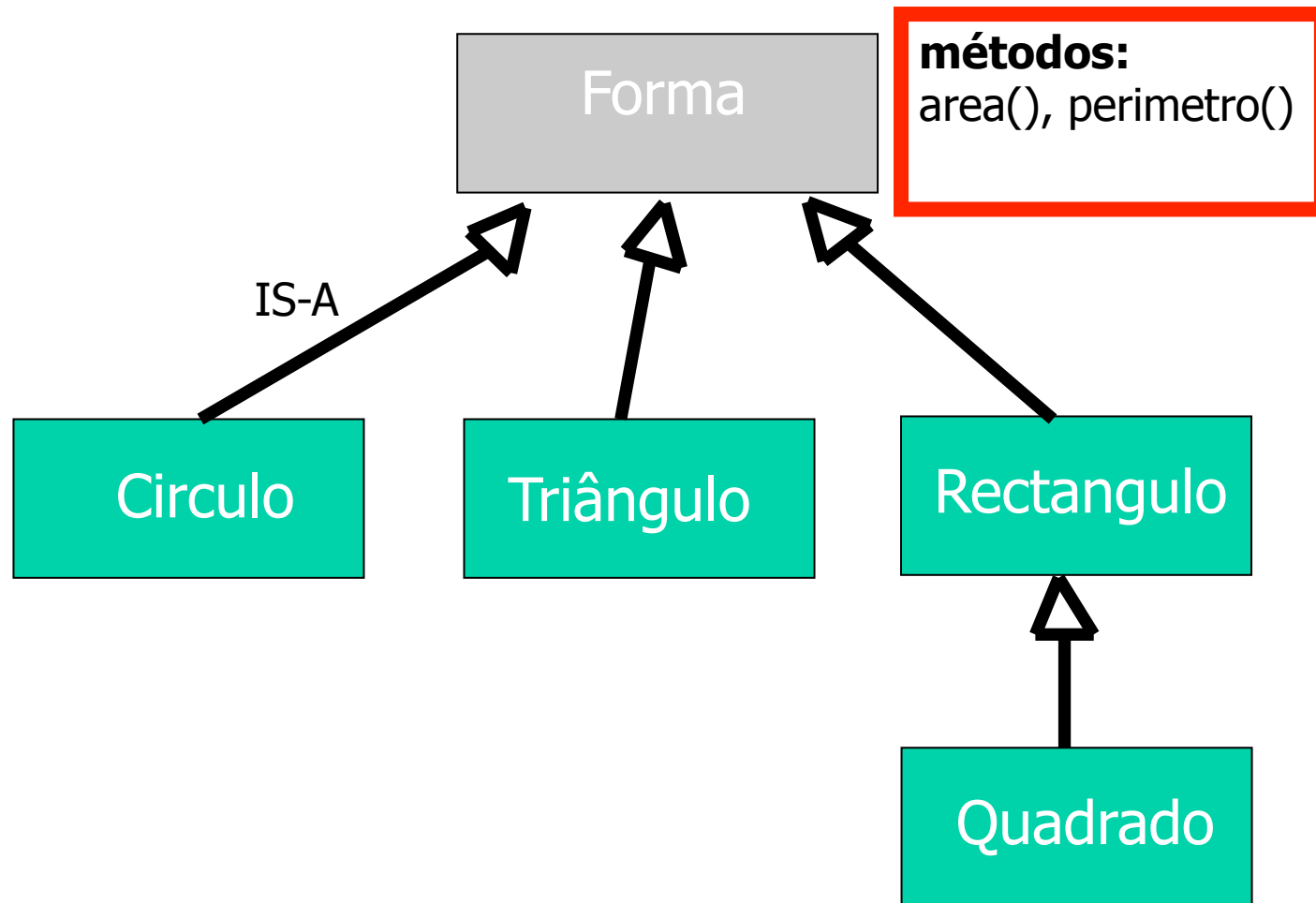
```
ZooAnimal.peso;
```





Questões?

- Como é que implementamos os métodos de Forma?



Classes abstractas

- Uma classe é abstracta se contiver pelo menos um método abstracto.

- Um método abstracto 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 abstracta não é instanciável.

```
Forma f;           // OK. Podemos criar uma referência para Forma
f = new Forma();   // Erro! Não podemos criar Formas
```

Classes abstractas

- Num processo de herança a classe só deixa de ser abstracta quando implementar todos os métodos abstractos.

```
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 abstractas 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) --> Quadrado
Figura(8) --> Circulo
Figura(9) --> Quadrado
```

Java Interfaces

UA, DETI, Programação III
José Luis Oliveira, Carlos Costa
2014/15

Interfaces

- Uma interface é uma classe abstracta pura (só contém assinaturas).

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

- Actua 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

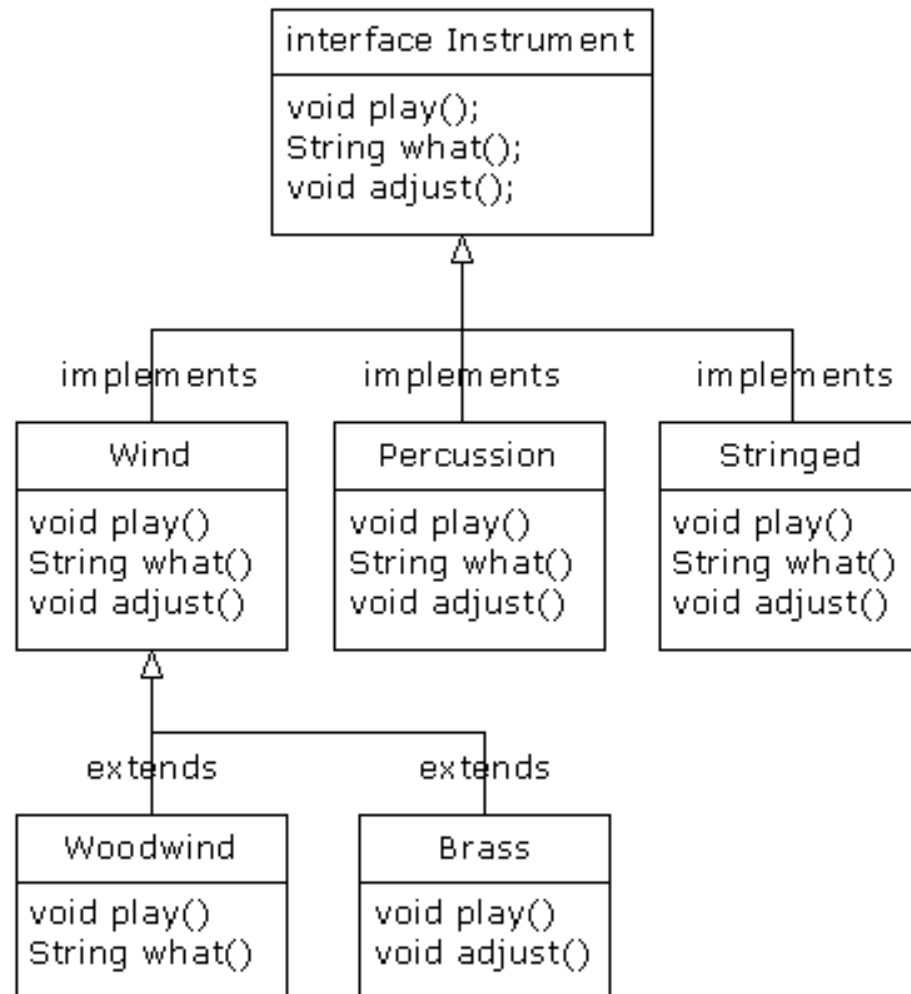
```
interface Desenhavel {  
    public void cor(Color c);  
    public void corDeFundo(Color cf);  
    public void posicao(double x, double y);  
    public void desenha(DrawWindow dw);  
}  
  
class CirculoGrafico extends Circulo implements Desenhavel {  
    public void cor(Color c) {...}  
    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, abstractos.
 - Os únicos modificadores permitidos são `public` e `abstract`.
- Uma interface pode herdar (extends) mais do que uma interface.
- Não são permitidos construtores nem métodos estáticos.
- As variáveis são implicitamente estáticas e constantes
 - `static final ..`
- Uma classe (não abstracta) 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 actuar as regras sobre classes

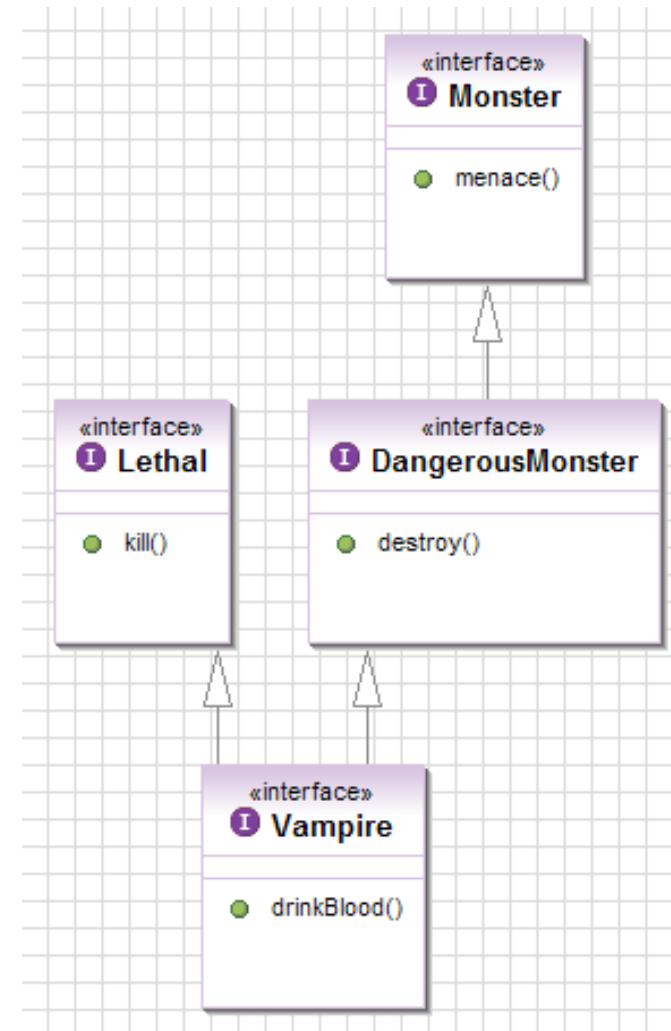


Interfaces - Exemplos

```
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();  
}
```



Classes Abstractas versus Interfaces

Classes Abstractas

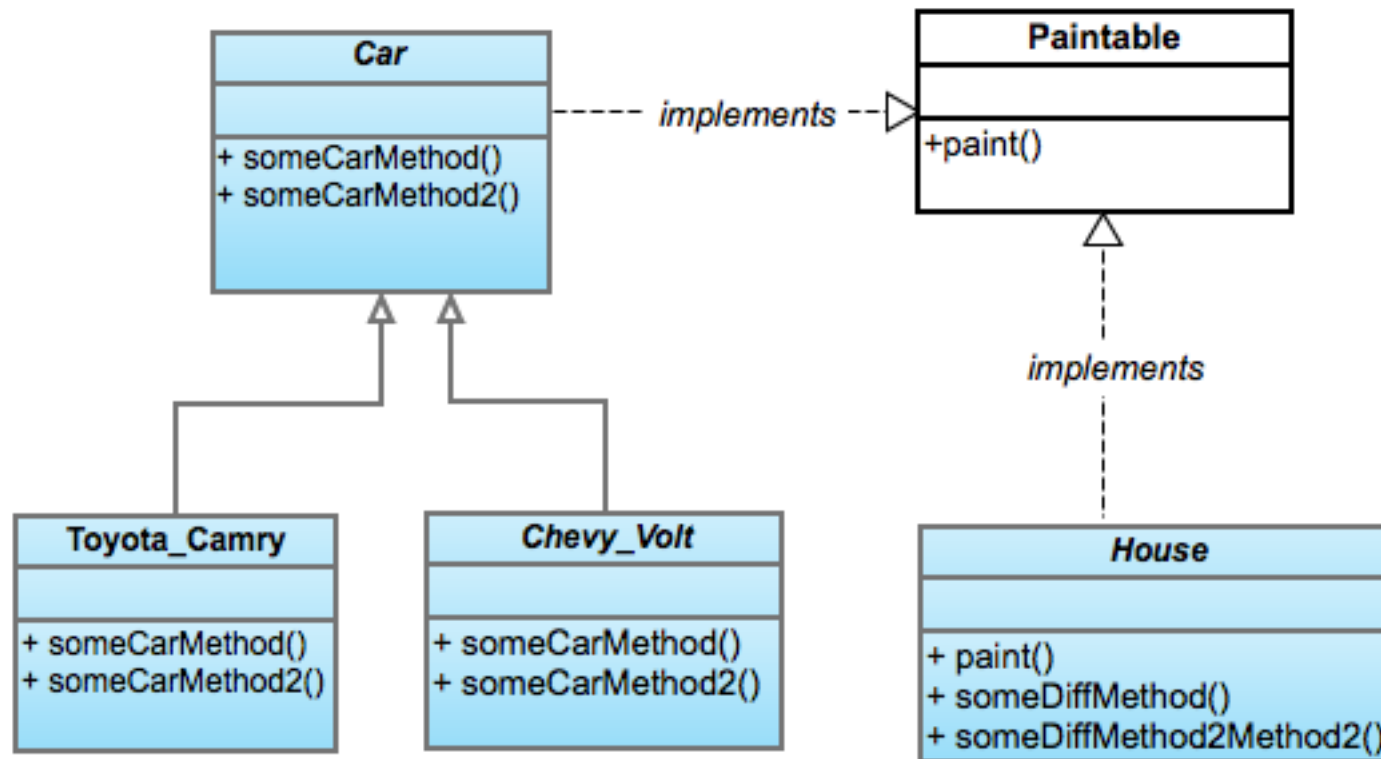
- pode não ser 100% abstracta
- escrever software genérico, parametrizável e extensível
- relacionamento na hierarquia simples de classes

Interfaces

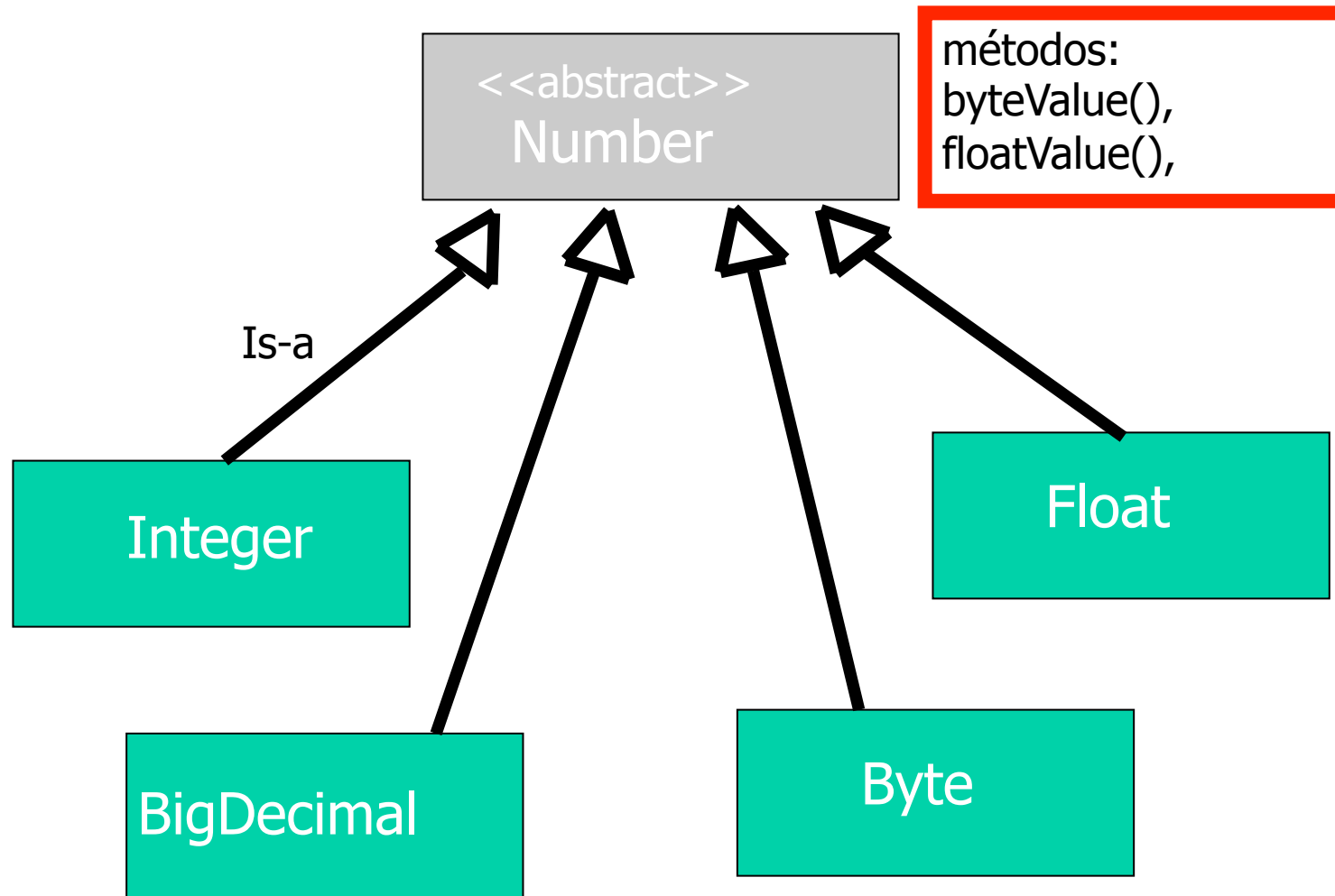
- 100% abstractas
- especificar um conjunto adicional de comportamentos / propriedades funcionais
- implementação horizontal na hierarquia

Não há regras ou metodologia: "... neste caso usa-se interfaces, no outro ..."

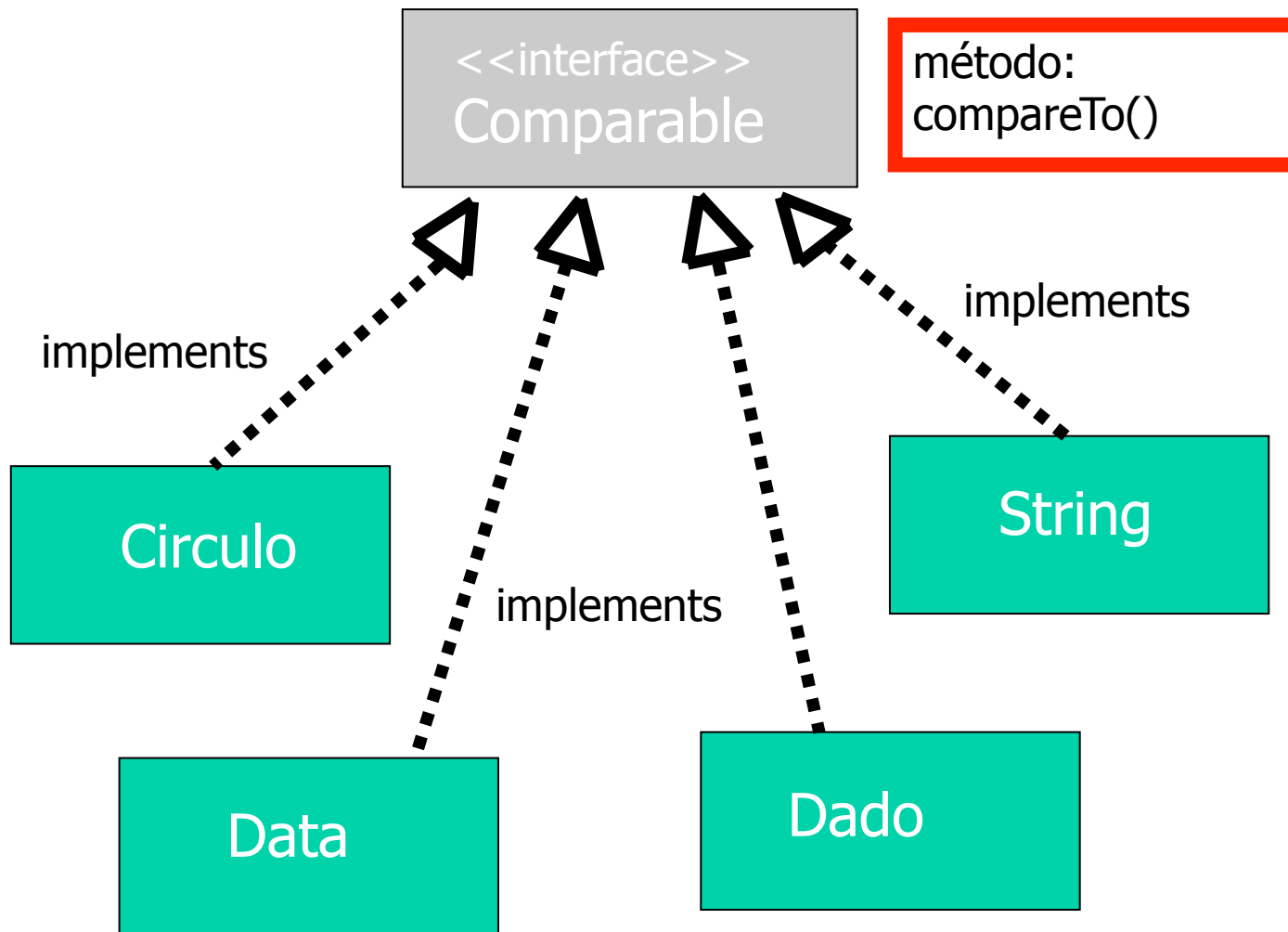
Classes Abstractas versus Interfaces



Classes Abstractas versus Interfaces

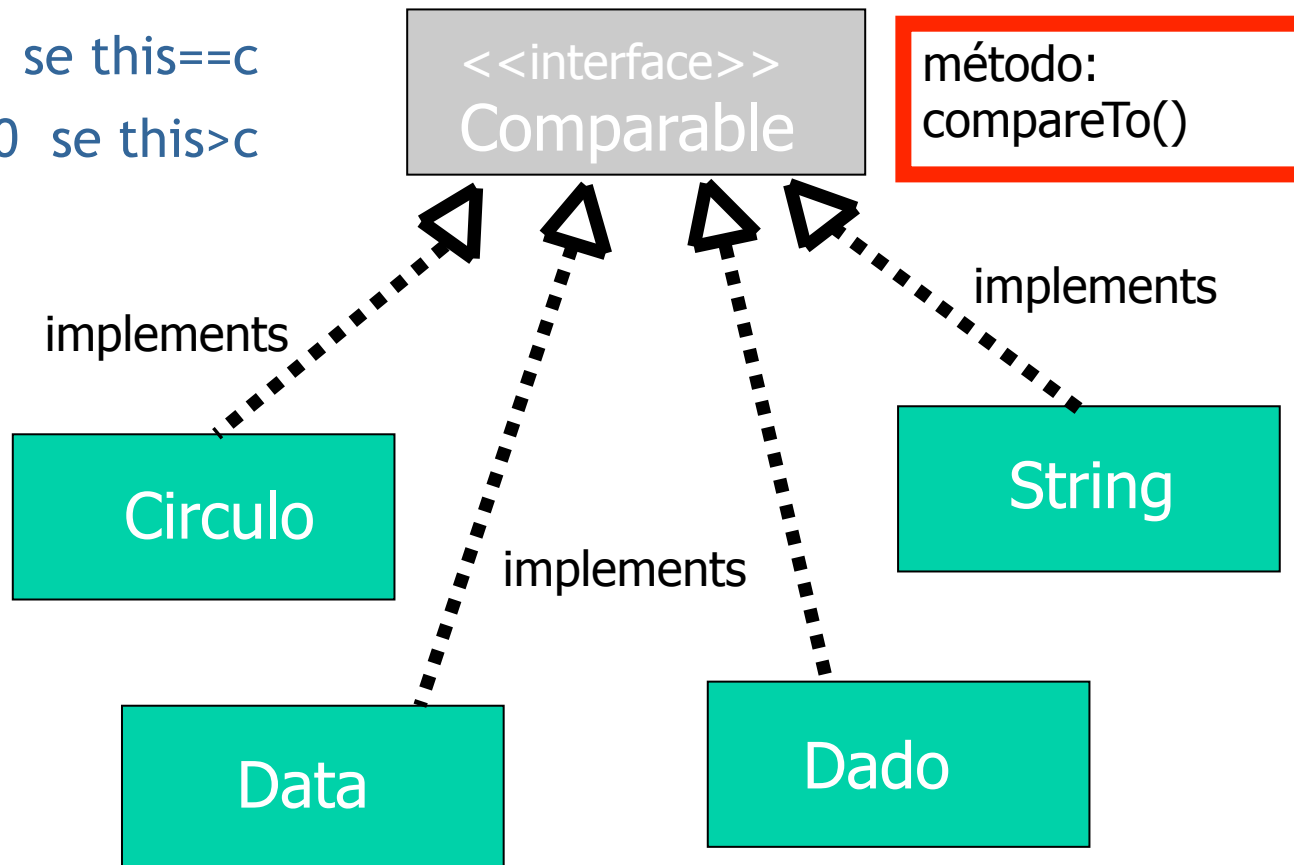


Classes Abstractas versus Interfaces



Questões?

- Qual o interesse de usar uma interface neste caso?
- Note que o método *int compareTo(Object 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 ) {
        assert (irhs!=null);
        return area() - rhs.area();
    }
}
```

```

class FindMaxDemo {

    public static Comparable findMax( Comparable[] a ) {
        int maxIndex = 0;

        for( int i = 1; i < a.length; i++ )
            if( a[i].compareTo(a[maxIndex]) > 0 )
                maxIndex = i;

        return a[maxIndex];
    }

    public static void main( String [ ] args ) {
        Shape [ ] sh1 = { new Circle( 2.0 ),
                           new Square( 3.0 ),
                           new Rectangle( 3.0, 4.0 ) };

        String [ ] st1 = { "Joe", "Bob", "Bill", "Zeke" };

        System.out.println( findMax( sh1 ) );
        System.out.println( findMax( st1 ) );
    }
}

```

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) ..
```

Copiar objetos (clone)

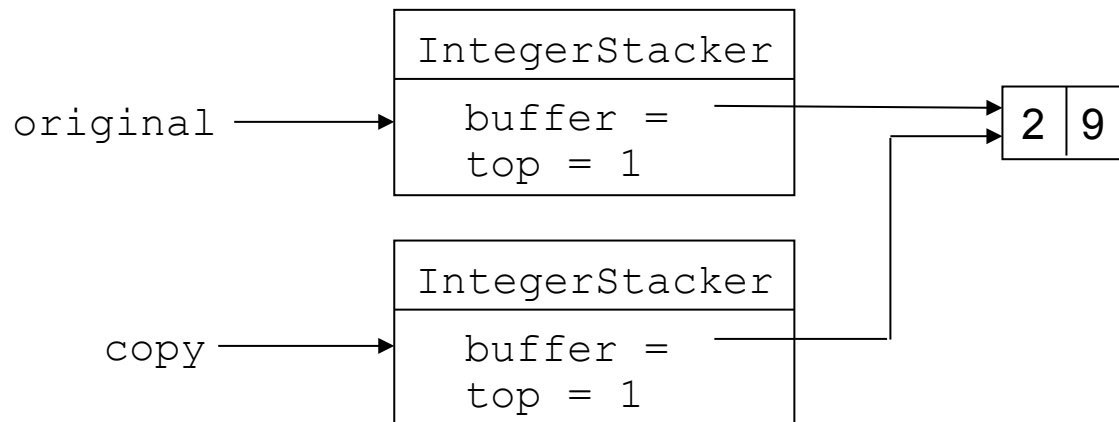
- `protected Object clone()`
 - Retorna um novo objeto cujo estado inicial é uma cópia do objeto sobre o qual o método foi invocado.
 - As alterações subseqüente na réplica não afetarão o original.
 - Este método realiza uma cópia simples de todos os campos. Nem sempre é adequado.
- Construtor de cópia
 - Construtor cujo argumento é um objeto da mesma classe

```
public Figura(Figura original) {  
    ...  
}
```

Shallow cloning

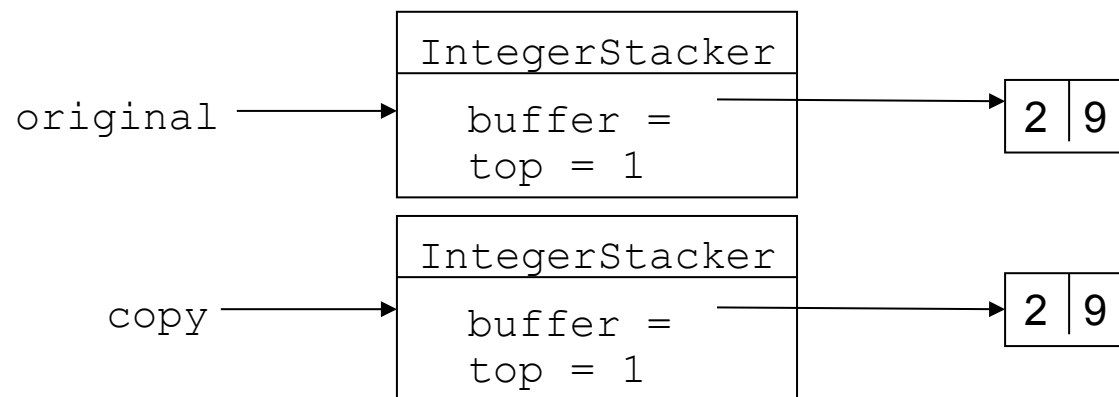
- Cópia campo a campo.
 - This might be wrong if it duplicates a reference to an object that shouldn't be shared.

```
public class IntegerStack {  
    private int[] buffer; // a stacker of integers  
    private int top;      // largest index in the stacker  
                          // (starting from 0)  
    . . .  
}
```



Deep cloning

- Cria uma réplica de todos os objectos que podem ser alcançados a partir do objeto que estamos a replicar



Interface java.lang.Cloneable

- Se quisermos fazer uso de `Object.clone()` temos de implementar a interface `Cloneable`
 - não tem métodos nem constantes (vazia) e funciona como um marcador

```
public class Rectangle implements Cloneable{
    ...
}
```
 - Shallow copy

```
@Override protected Rectangle clone() throws
CloneNotSupportedException {
    return (Rectangle) super.clone();
}
```
 - Deep copy - temos de ser nós a garantir a implementação local de `clone()`

```
@Override protected Rectangle clone() throws
CloneNotSupportedException {
    return new Rectangle(...);
}
```


Java

Classes internas

UA, DETI, Programação III
José Luis Oliveira, Carlos Costa
2014/15

Classes internas

- Classes podem ser membros de classes, de objetos ou locais a métodos. Podem até serem criadas sem nome, apenas com corpo no momento em que instanciam um objeto
 - Há poucas situações onde classes internas podem ou devem ser usadas. Devido à complexidade do código que as utiliza, deve evitar-se usos não convencionais
 - Usos típicos incluem tratamento de eventos em GUIs, criação de threads, manipulação de coleções e sockets
- Classes internas podem ser classificadas em quatro tipos
 - Classes estáticas - classes membros de classe (*nested classes*)
 - Classes de instância - classes membros de objetos
 - Classes locais - classes dentro de métodos
 - Classes anónimas - classes dentro de instruções

Classes estáticas

- São declaradas como *static* dentro de uma classe
- A classe externa age como um pacote para uma ou mais classes internas estáticas
 - *Externa.Coisa, Externa.InternaUm, ..*
- O compilador gera arquivos tipo *Externa\$InternaUm.class*

```
class Externa {  
    private static class InternaUm {  
        public int campo;  
        public void metodoInterno() {...}  
    }  
    public static class InternaDois  
        extends InternaUm {  
        public int campo2;  
        public void metodoInterno() {...}  
    }  
    public static interface Coisa {  
        void existe();  
    }  
    public void metodoExterno() {...}  
}
```

Classes de instância

- São membros do objeto, como métodos e atributos
- Requerem que objeto exista antes que possam ser usadas.
 - Externamente usa-se *referência.new* para criar objetos
- Deve usar-se *NomeDaClasse.this* para aceder a campos internos

```
class Externa {  
    public int campoUm;  
    public class Interna {  
        public int campoUm;  
        public int campoDois;  
        public void metodoInterno() {  
            this.campoUm = 10; // Externa.campoUm  
            Interna.this.campoUm = 15;  
        }  
    }  
    public static void main(String[] args){  
        Interna e = (new Externa()).new Interna();  
    }  
}
```

Classes locais

- Servem para tarefas temporárias já que deixam de existir quando o método acaba
 - Têm o mesmo alcance de variáveis locais.

```
public Multiplicavel calcular(final int a, final int b) {  
    class Interna implements Multiplicavel {  
        public int produto() {  
            return a * b; // usa a e b, que são constantes  
        }  
    }  
    return new Interna();  
}  
  
public static void main(String[] args){  
    Multiplicavel mul = (new Externa()).calcular(3,4);  
    int prod = mul.produto();  
}
```

Classes anónimas

- Servem para criar um único objeto
 - A classe abaixo estende ou implementa SuperClasse, que pode ser uma interface ou classe abstracta (o new, neste caso, indica a criação da classe entre chavetas, não da SuperClasse)
`Object i = new SuperClasse() { implementação };`
 - O compilador gera arquivo Externa\$1.class, Externa\$2.class,

```
public Multiplicavel calcular(final int a, final int b) {  
    return new Multiplicavel() {  
        public int produto() {  
            return a * b;  
        }  
    };  
}  
public static void main(String[] args){  
    Multiplicavel mul = (new Externa()).calcular(3,4);  
    int prod = mul.produto();  
}
```

Compare com parte em preto e vermelho do slide anterior!

A classe está dentro da instrução: preste atenção no ponto-e-vírgula!

Classes internas

- São sempre classes dentro de classes. Exemplo:

```
class Externa {  
    private class Interna {  
        public int campo;  
        public void metodoInterno() {...}  
    }  
    public void metodoExterno() {...}  
}
```

- Podem ser *private*, *protected*, *public* ou *package-private*
 - Excepto as que aparecem dentro de métodos, que são locais
- Podem ser estáticas:
 - E chamadas usando a notação `Externa.Interna`
- Podem ser de instância e depender da existência de objetos:

```
Externa e = new Externa();  
Externa.Interna ei = e.new Externa.Interna();
```
- Podem ser locais (dentro de métodos)
 - E nas suas instruções podem não ter nome (anónimas)

Sumário

- Polimorfismo
- Generalização
- Classes abstractas
- Interfaces
- Classes internas