

Java Interfaces

Interfaces

- ❖ Funciona como um tipo
- ❖ Contem assinaturas de métodos que devem existir
 - implicitamente, abstratos.
 - Só podem ser *public* e/ou *abstract*.
- ❖ Contem variáveis
 - implicitamente estáticas e constantes
 - `static final` ..
- ❖ pode herdar (*implements*) mais do que uma interface.
- ❖ Uma interface pode ser vazia
 - `Cloneable`, `Serializable`

```
List<Integer> l=new ArrayList<>()

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

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() { /* .. */ }
}
```

Classes e interfaces

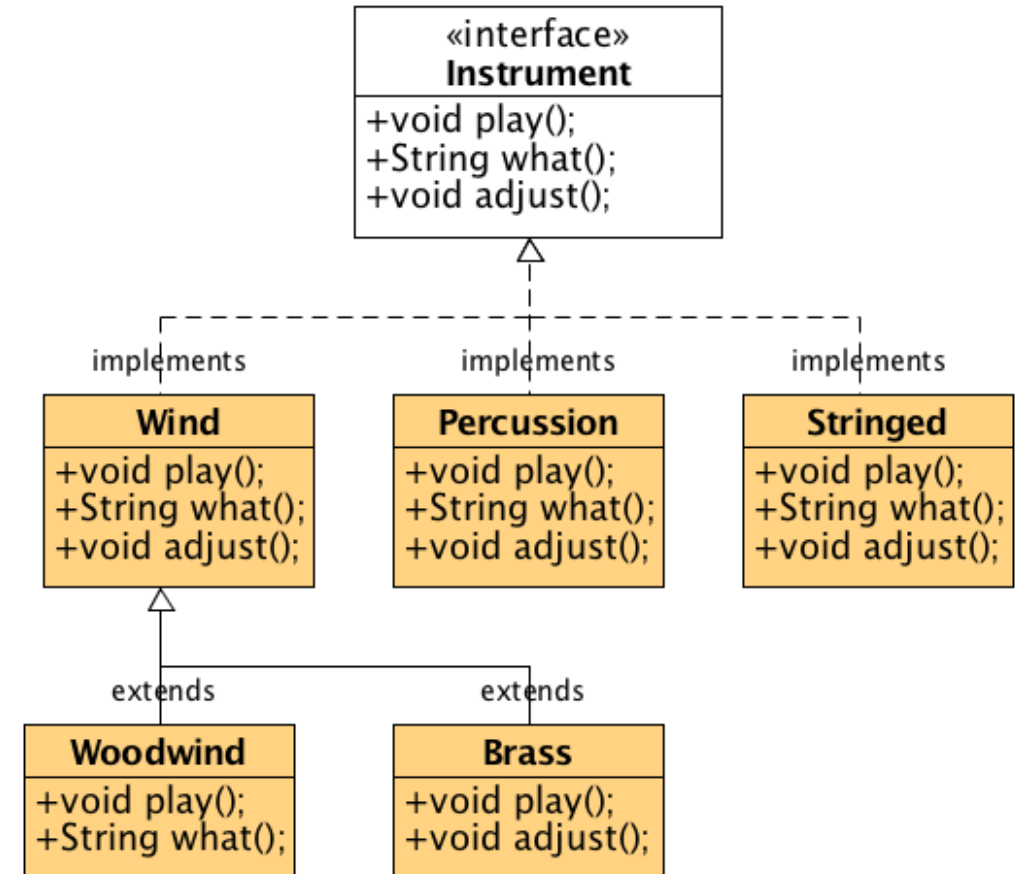
- ❖ Não é uma classe
 - não permite construtores.
 - Não se pode criar uma instância da interface
- ❖ um protocolo para as classes que as implementam.
 - implementar todos os seus métodos.
 - A partir do Java 8 passou a incluir métodos *default* e *static*.

```
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) {...}  
}
```

Classes e interfaces

- ❖ As classes implementam (*implements*) interfaces.
- ❖ O protocolo da interface é válido para as classes que a implementam
- ❖ As classes derivadas também herdam a implementação da interface



Exemplo: List

```
import java.util.ArrayList;
import java.util.List;

public class ListInterfaceAddElements {

    public static void main(String[] args) {

        List<Integer> list = new
        list.add(1);
        list.add(3);
        list.add(4);
        // Insertion order maintained. Output: [1, 3, 4]
        System.out.println(list);

        // add 2 at index 1. Output: [1, 2, 3, 4]
        list.add(1, 2);
        System.out.println(list);

        List<Integer> list2 = new
        // append list2 to list 1
        list.addAll(list2);

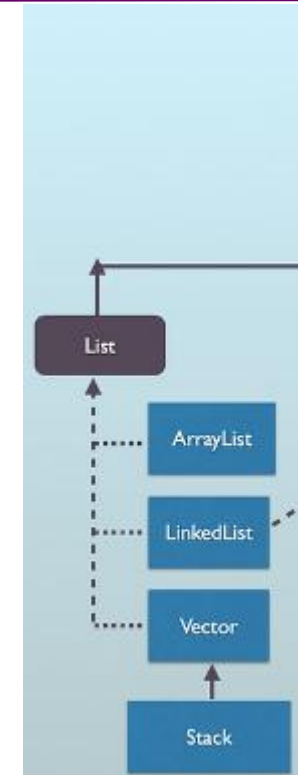
        // Order maintained and Output is: [1, 2, 3, 4, 77, 66, 55]
        System.out.println(list);

        List<Integer> list3 = new
        // adding all elements from list3 at index 3
        list.addAll(3, list3);
        // Order maintained and output is: [1, 2, 3, 44, 33, 22, 4, 77, 66, 55]
        System.out.println(list);
    }
}
```

Substituir aqui

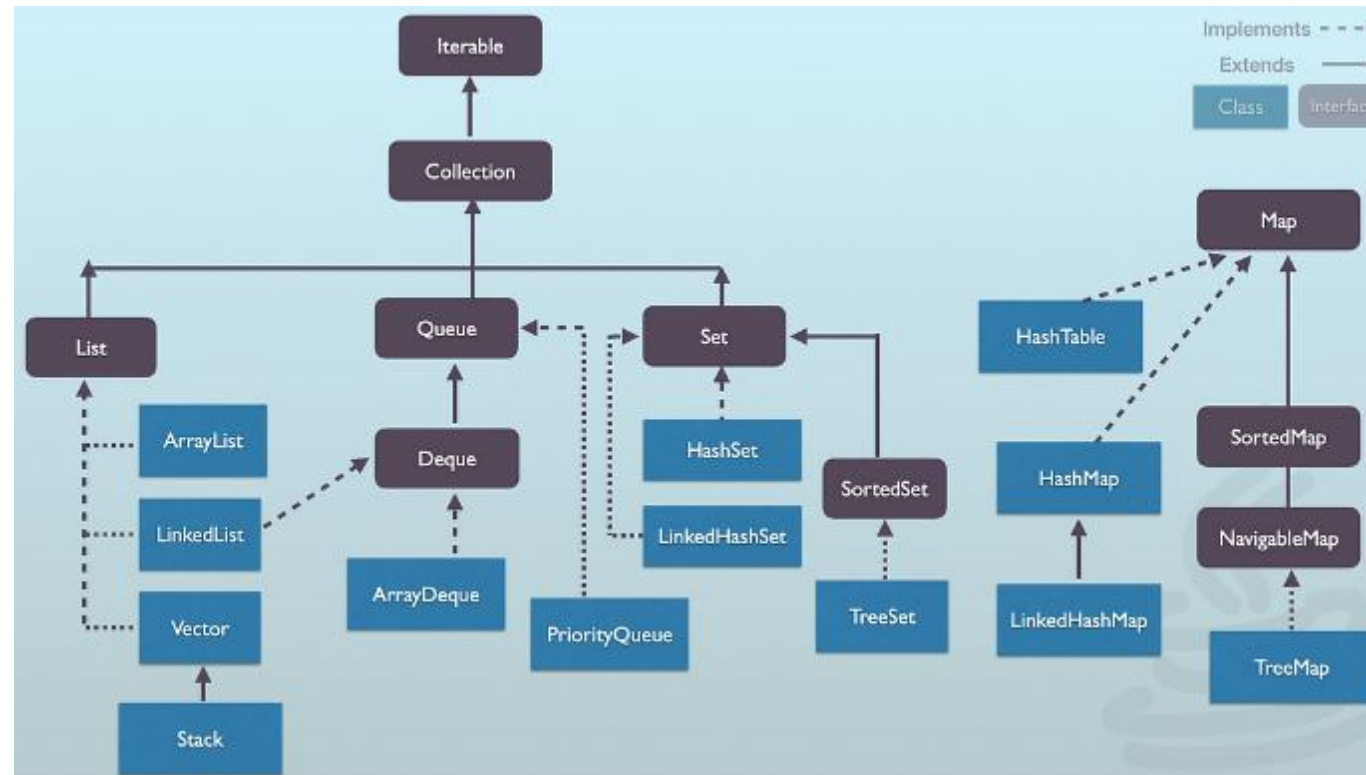
Substituir aqui

Substituir aqui



Exemplo: collections

❖ List é um interface



<https://medium.com/@rmg007/the-list-interface-in-java-cb62d524fed4>

https://www.tutorialspoint.com/java/java_list_interface.htm

List é um interface

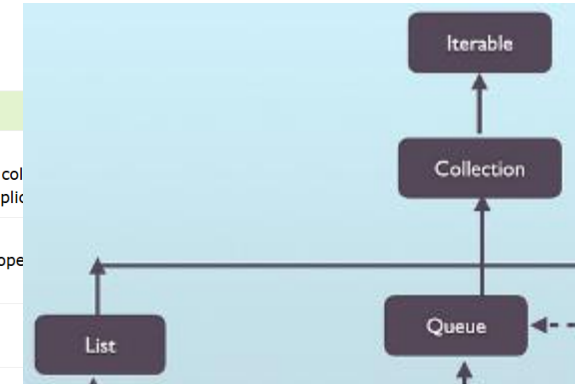
Sr.No.	Method & Description
1	void add(int index, Object obj) Inserts obj into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	boolean addAll(int index, Collection c) Inserts all elements of c into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	Object get(int index) Returns the object stored at the specified index within the invoking collection.
4	int indexOf(Object obj) Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.
5	int lastIndexOf(Object obj) Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.
6	ListIterator listIterator() Returns an iterator to the start of the invoking list.
7	ListIterator listIterator(int index) Returns an iterator to the invoking list that begins at the specified index.
8	Object remove(int index) Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
9	Object set(int index, Object obj) Assigns obj to the location specified by index within the invoking list.
10	List subList(int start, int end) Returns a list that includes elements from start to end-1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

List extends interface Collection

Sr.No.	Method & Description
1	void add(int index, Object obj) Inserts obj into the invoking list at the index passed in the index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	boolean addAll(int index, Collection c) Inserts all elements of c into the invoking list at the index passed in the index. Elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	Object get(int index) Returns the object stored at the specified index within the invoking collection.
4	int indexOf(Object obj) Returns the index of the first instance of obj in the invoking list. If obj is not found, returns -1.
5	int lastIndexOf(Object obj) Returns the index of the last instance of obj in the invoking list. If obj is not found, returns -1.
6	ListIterator listIterator() Returns an iterator to the start of the invoking list.
7	ListIterator listIterator(int index) Returns an iterator to the invoking list that begins at the specified index.
8	Object remove(int index) Removes the element at position index from the invoking list and returns the element. The resulting list is compacted. That is, the indexes of subsequent elements are adjusted.
9	Object set(int index, Object obj) Assigns obj to the location specified by index within the invoking list.
10	List subList(int start, int end) Returns a list that includes elements from start to end-1 in the invoking list. The returned list is also referenced by the invoking object.

LIST

Sr.No.	Method & Description
1	boolean add(Object obj) Adds obj to the invoking collection. Returns true if obj was added to the collection, or if the collection does not allow duplicates.
2	boolean addAll(Collection c) Adds all the elements of c to the invoking collection. Returns true if the collection changes (i.e., elements were added). Otherwise, returns false.
3	void clear() Removes all elements from the invoking collection.
4	boolean contains(Object obj) Returns true if obj is an element of the invoking collection. Otherwise, returns false.
5	boolean containsAll(Collection c) Returns true if the invoking collection contains all elements of c. Otherwise, returns false.
6	boolean equals(Object obj) Returns true if the invoking collection and obj are equal. Otherwise, returns false.
7	int hashCode() Returns the hash code for the invoking collection.
8	boolean isEmpty() Returns true if the invoking collection is empty. Otherwise, returns false.
9	Iterator iterator() Returns an iterator for the invoking collection.
10	boolean remove(Object obj) Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
11	boolean removeAll(Collection c) Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
12	boolean retainAll(Collection c) Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
13	int size() Returns the number of elements held in the invoking collection.
14	Object[] toArray() Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.



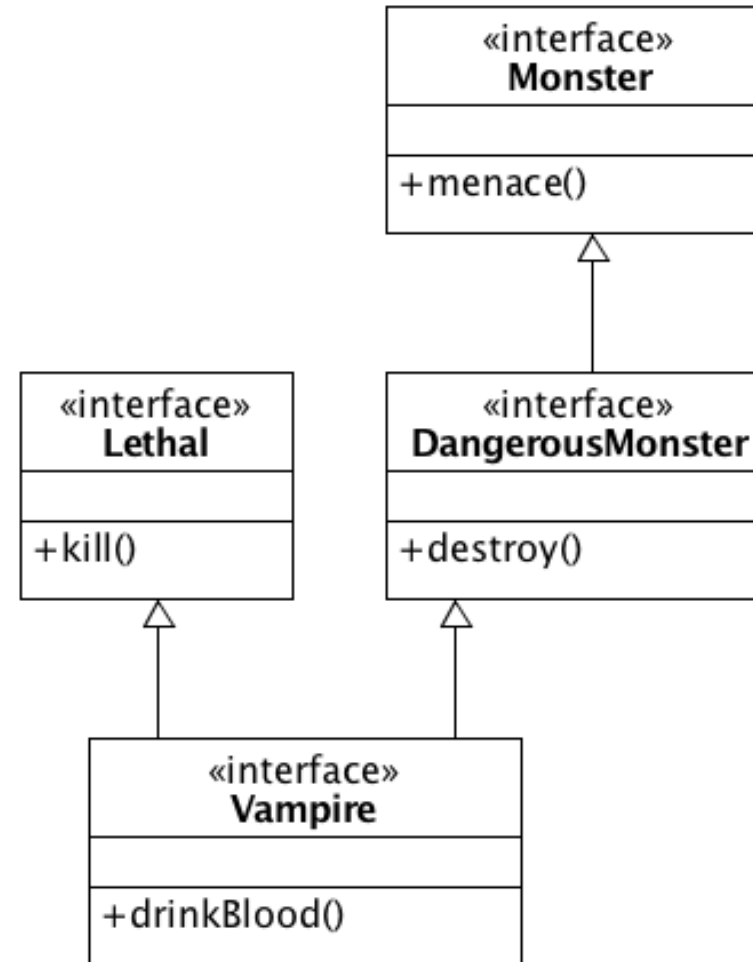
COLLECTION

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();  
}
```



Definição de interfaces

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...)*
-
- ❖ **Para que servem 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  
    }  
}
```

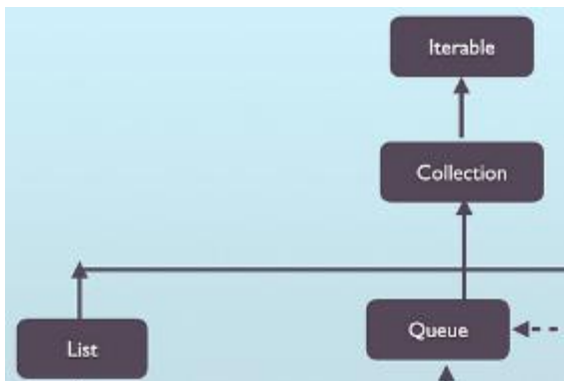
```
public class MyClass implements  
Interface2 {  
    @Override  
    public void stMeth() { // ...  
do something  
    }  
}
```



Polimorfismo

Polimorfismo

```
List<Integer> l = new ArrayList<>();  
Collection<Integer> c1 = l;  
  
l.add(1);  
c1.add(2);  
  
// Convert Collection to an Integer array  
Integer[] arrayFromCollection = c1.toArray(new Integer[0]);  
Integer[] arrayFromList = l.toArray(new Integer[0]);  
  
// Print the arrays  
System.out.println("Array from Collection: " +  
Arrays.toString(arrayFromCollection));  
System.out.println("Array from List: " +  
Arrays.toString(arrayFromList));
```



Polimorfismo

- ❖ Formas aparentemente diferentes (*i.e., classes diferentes*) que implementam interfaces compatíveis:
 - Classes derivadas são compatíveis com as classes bases através de herança
 - Classes distintas podem implementar mesma interface

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: dynamic binding

- ❖ Num objeto polimórfico, os métodos a executar são selecionados de acordo com a forma (i.e., classe) da referência usada na invocação ao objeto. O java tem um mecanismo de ligação dinâmica:
 - *Dynamic binding*, também referido como *late binding* ou *run-time binding*
 - Todos os métodos **são late binding**.

```
class Parent {  
  
    public void showMessage() {  
        System.out.println("This is a method from the Parent class.");  
    }  
}  
  
// Child class that tries to inherit from Parent  
class Child extends Parent {  
  
    public void showMessage() {  
        System.out.println("Trying to override a method.");  
    }  
}
```

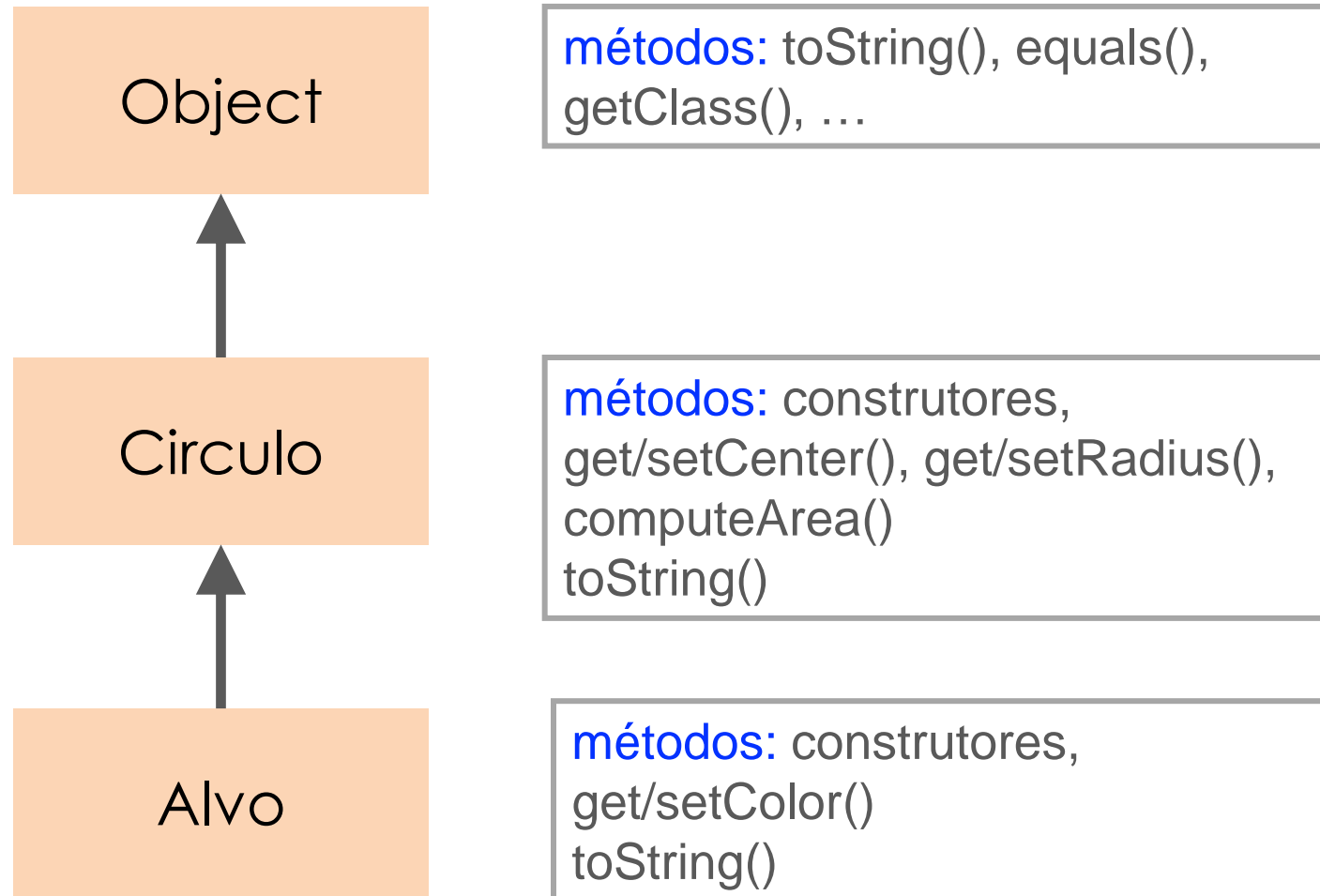
Polimorfismo: final

❖ Final numa função

- impede que seja redefinida
- indicação para ligação estática (*early binding*) –
- que é o único modo de ligação em linguagens como C.

```
class Parent {  
    // This method cannot be overridden  
    public final void showMessage() {  
        System.out.println("This is a final method from the Parent class.");  
    }  
}  
  
// Child class that tries to inherit from Parent  
class Child extends Parent {  
    // Uncommenting this will cause a compilation error  
    // public void showMessage() {  
    //     System.out.println("Trying to override a final method.");  
    // }  
}
```

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);
```

OK – um Alvo é um Circulo

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

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

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

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

OK

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) ..
```

<https://www.baeldung.com/java-instanceof>

<https://www.programiz.com/java-programming/instanceof>

Upcasting e downcasting

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

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

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

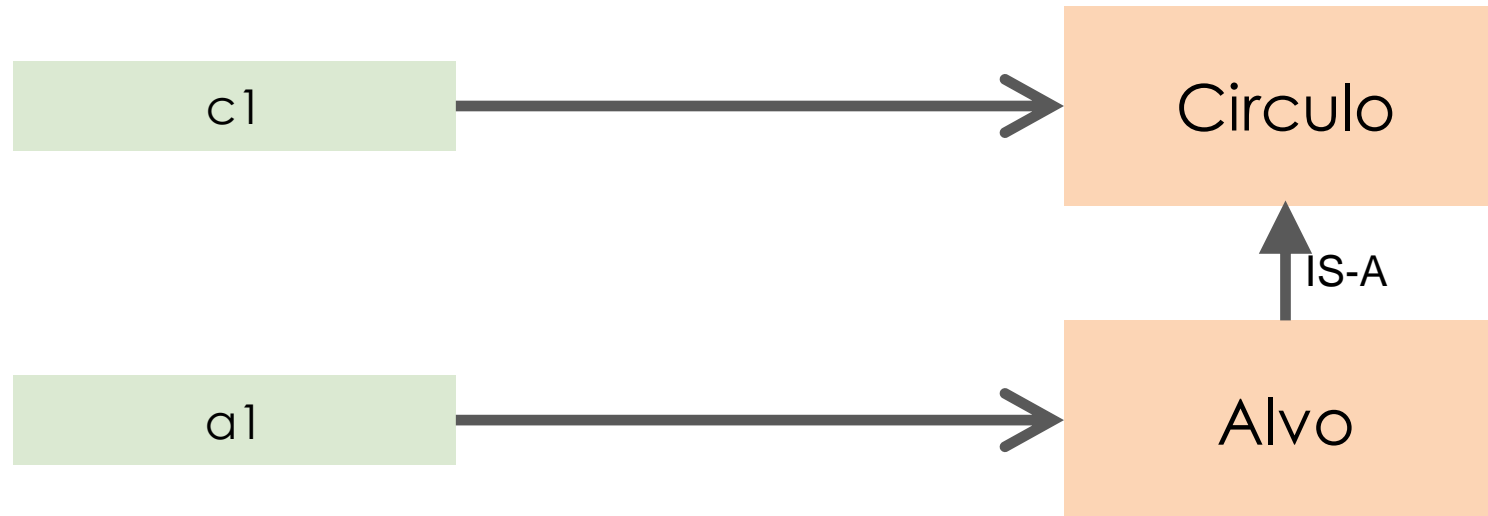
run-time error:
ClassCast exception

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

OK

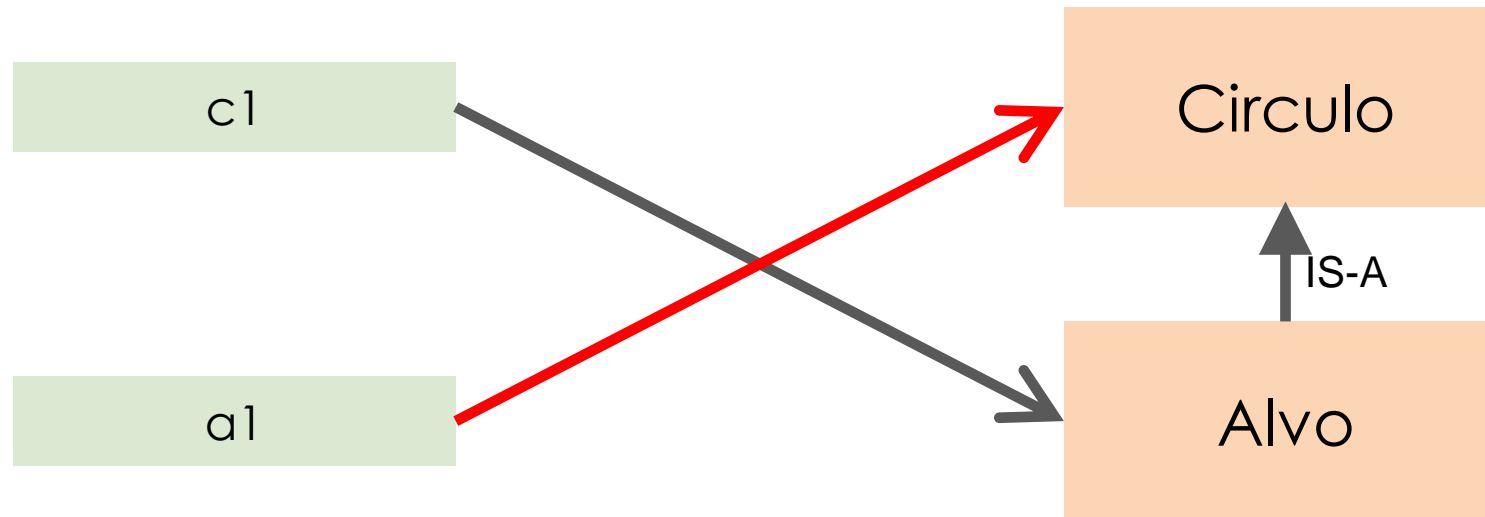
Polimorfismo - exemplos

```
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
```

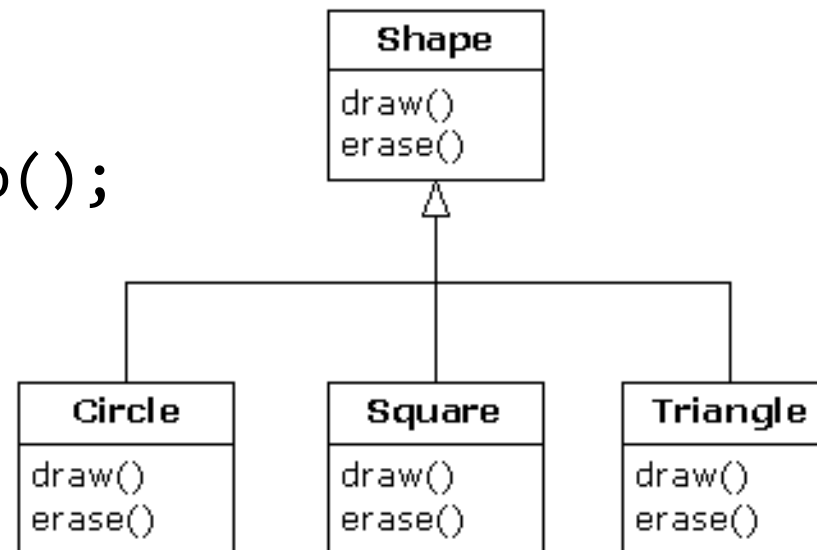


Exemplo 1

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

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

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

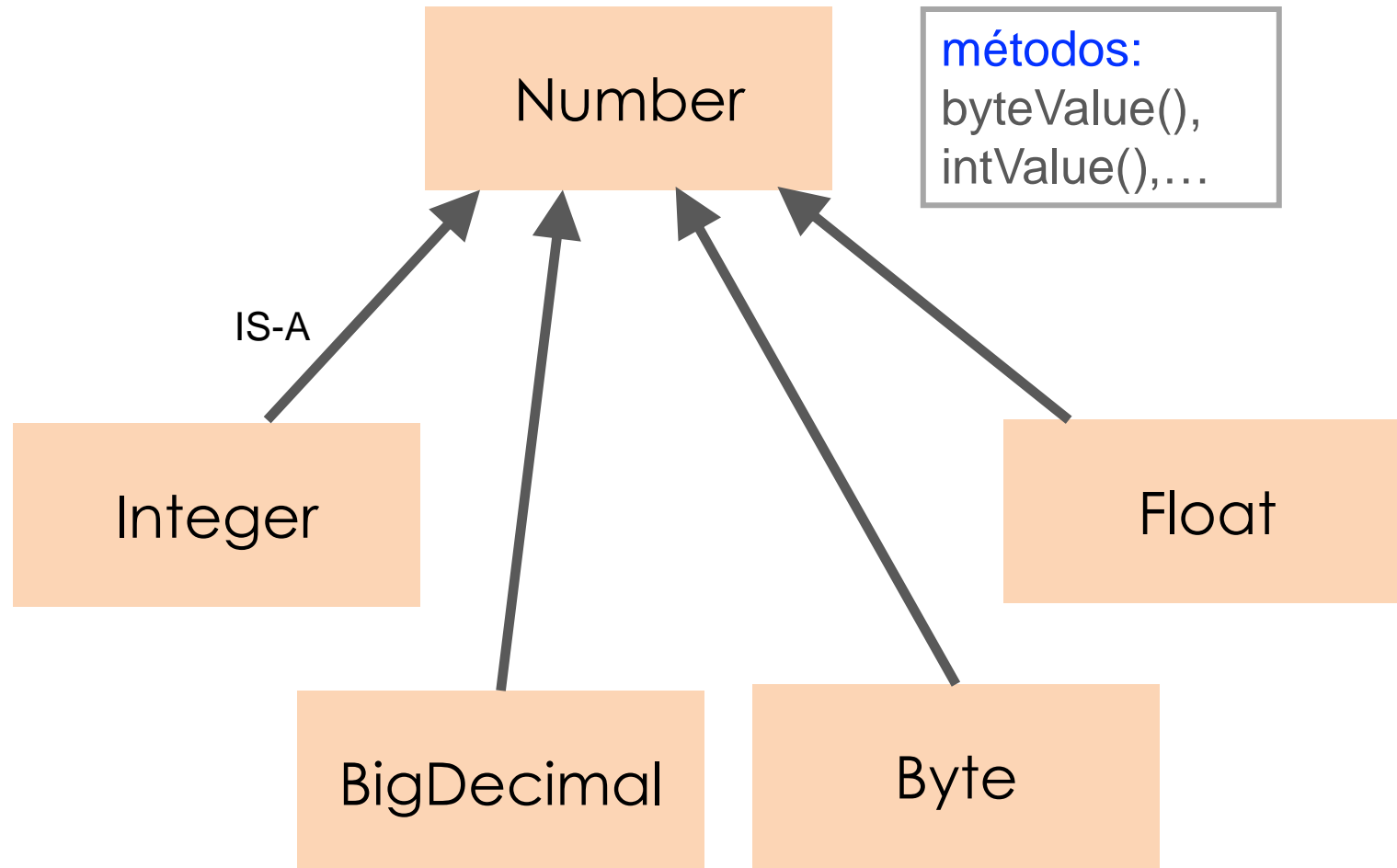


Exemplo 2

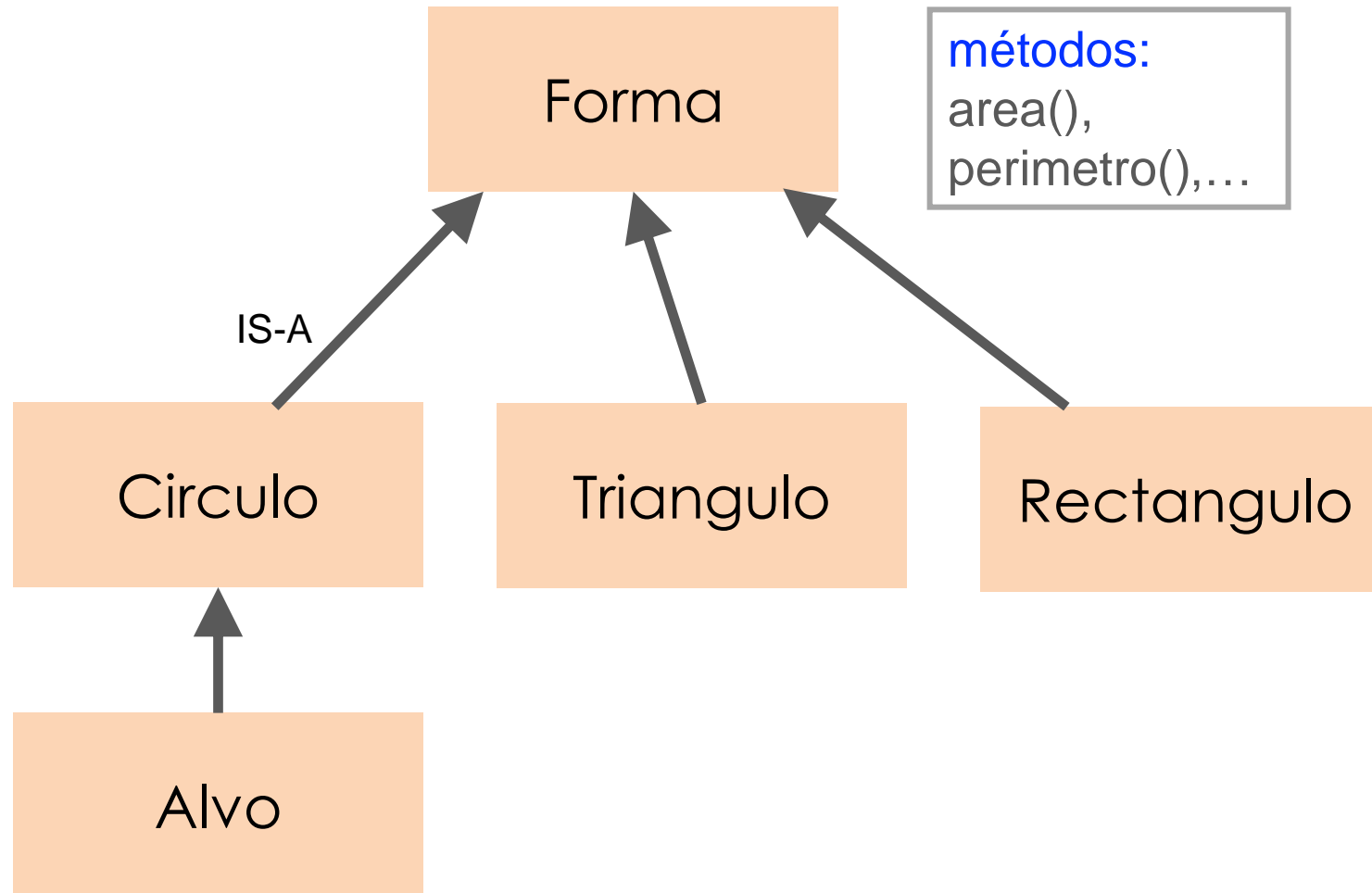
```
class Shape {  
    void draw() { System.out.println("I am a Shape"); }  
}  
  
class Circle extends Shape {  
    void draw() { System.out.println("I am a Circle"); }  
}  
  
class Square extends Shape {  
    void draw() { System.out.println("I am a Square"); }  
}  
  
public class ShapeSet {  
  
    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

Exemplo de herança

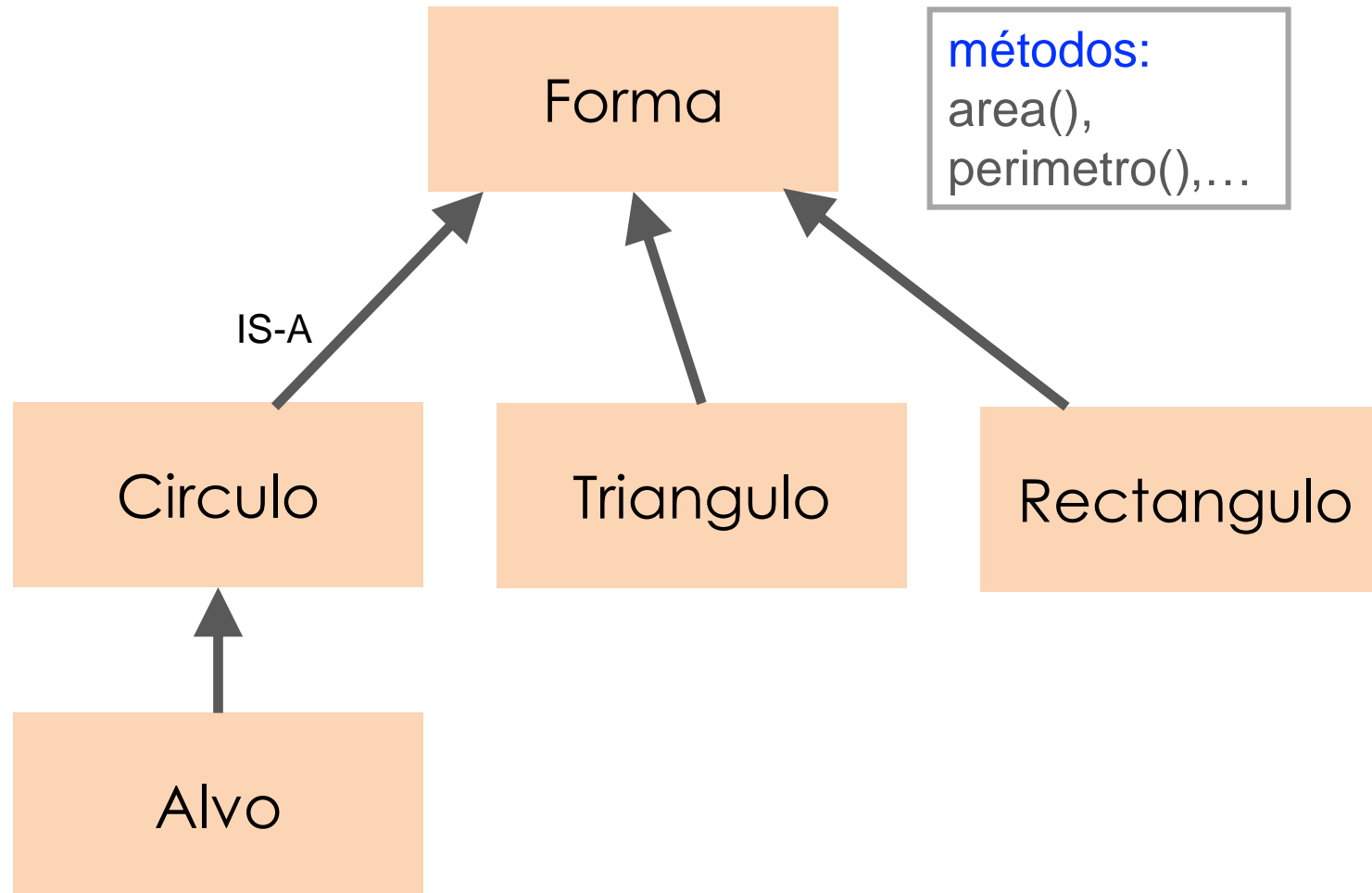


Exemplo de herança



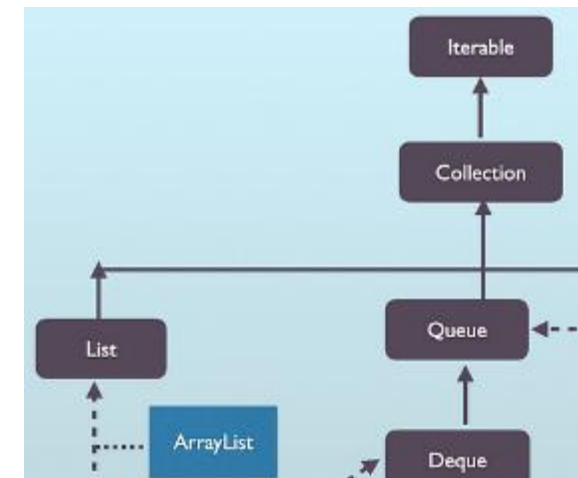
Exemplo de herança

❖ Como implementamos os métodos de Forma?



Polimorfismo, herança e interfaces

```
List<Integer> l = new ArrayList<>();  
Collection<Integer> cl = l;  
  
l.add(1);  
cl.add(2);  
  
// Convert Collection to an Integer array  
Integer[] arrayFromCollection = cl.toArray(new Integer[0]);  
Integer[] arrayFromList = l.toArray(new Integer[0]);  
  
// Print the arrays  
System.out.println("Array from Collection: " + Arrays.toString(arrayFromCollection));  
System.out.println("Array from List: " + Arrays.toString(arrayFromList));
```



Classes abstractas

Classes abstratas

- ❖ Contem pelo menos um método abstrato.
 - Um método abstrato é um método cujo corpo não é definido.
- ❖ Uma classe abstrata não é instanciável.

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

Classes abstratas

- ❖ Contem pelo menos um método abstrato.
 - Um método abstrato é um método cujo corpo não é definido.
- ❖ Uma classe abstrata não é instanciável.

```
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"; }  
}  
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

- ❖ 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
```

```
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"; }  
}
```

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

Generalização

- ❖ 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;`

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

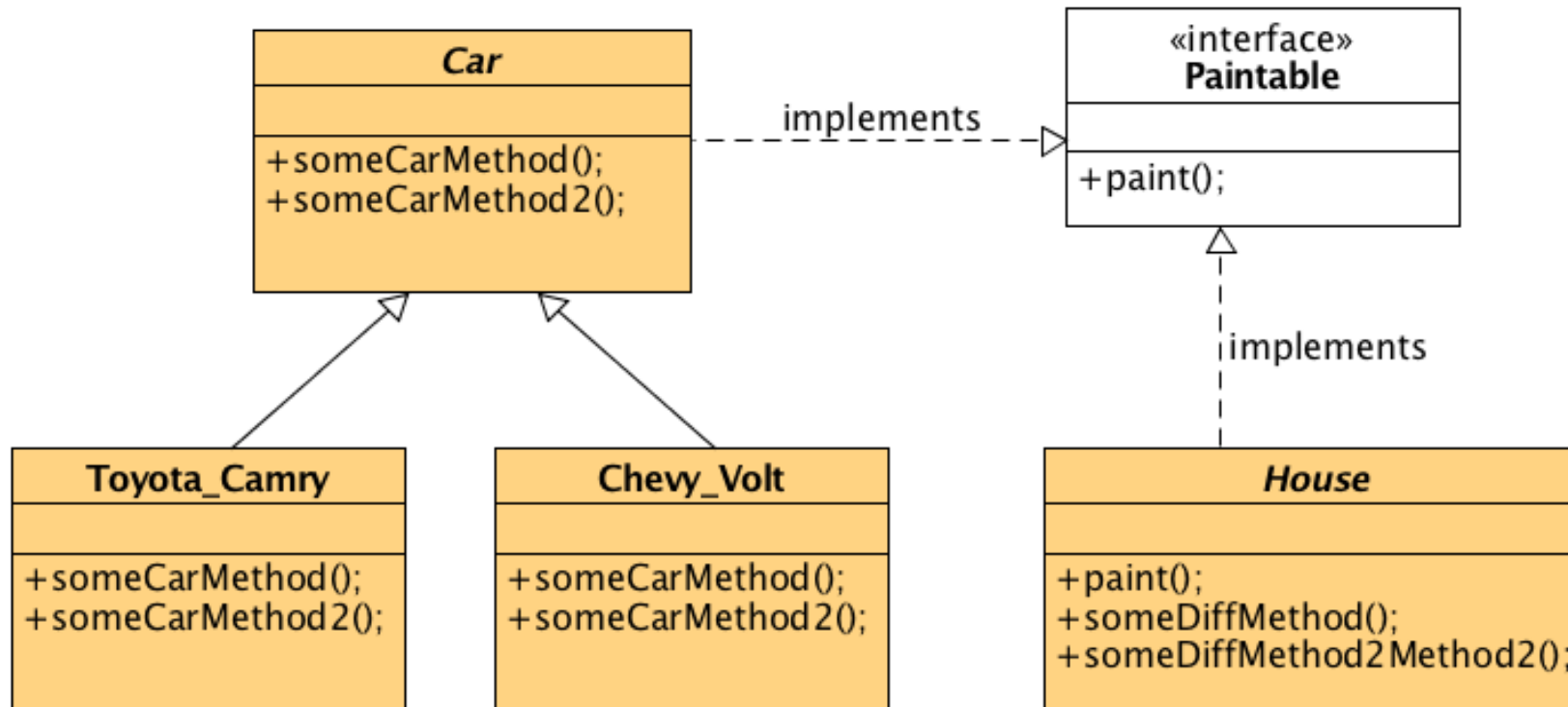
Classes Abstratas

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

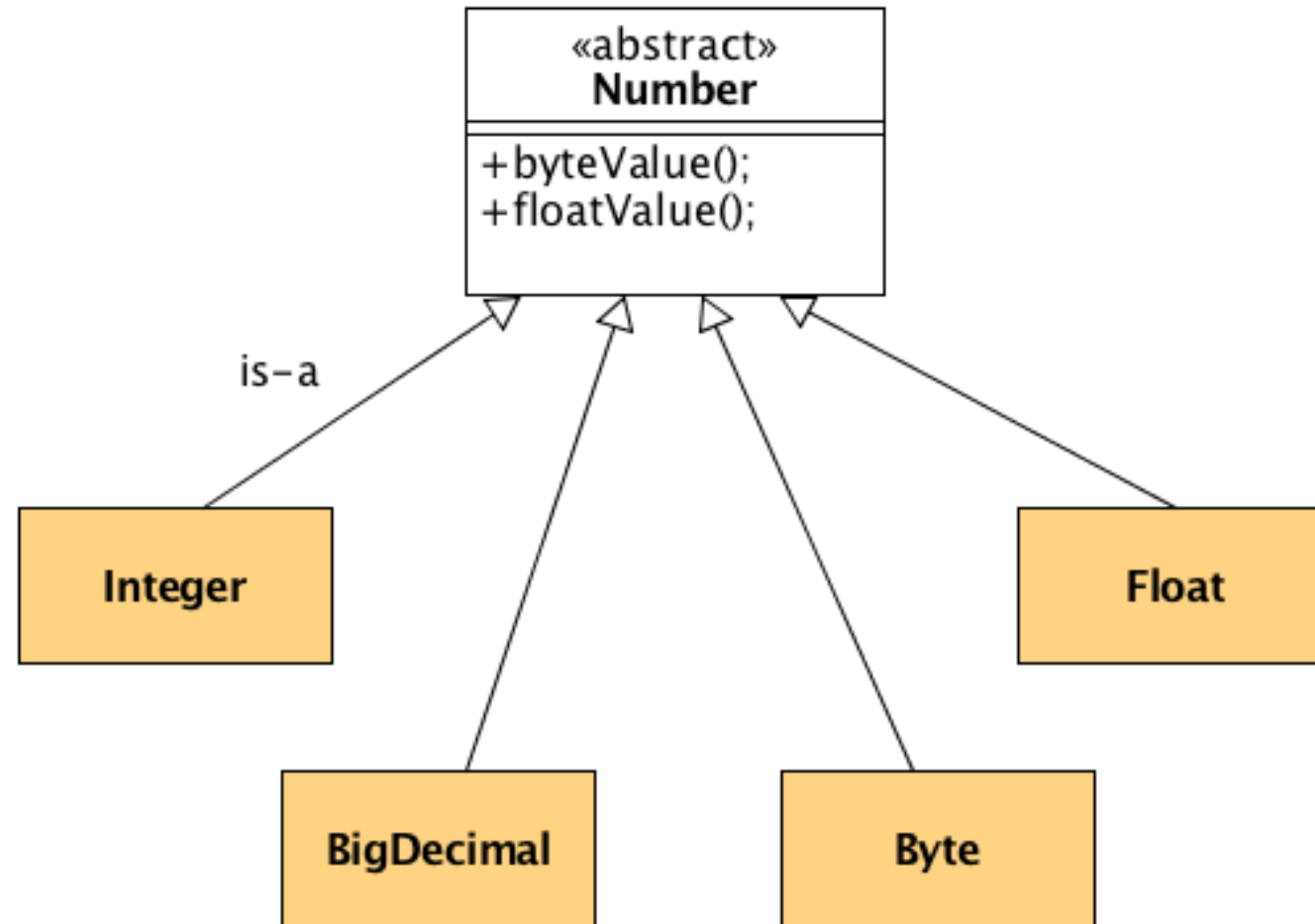
Interfaces

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

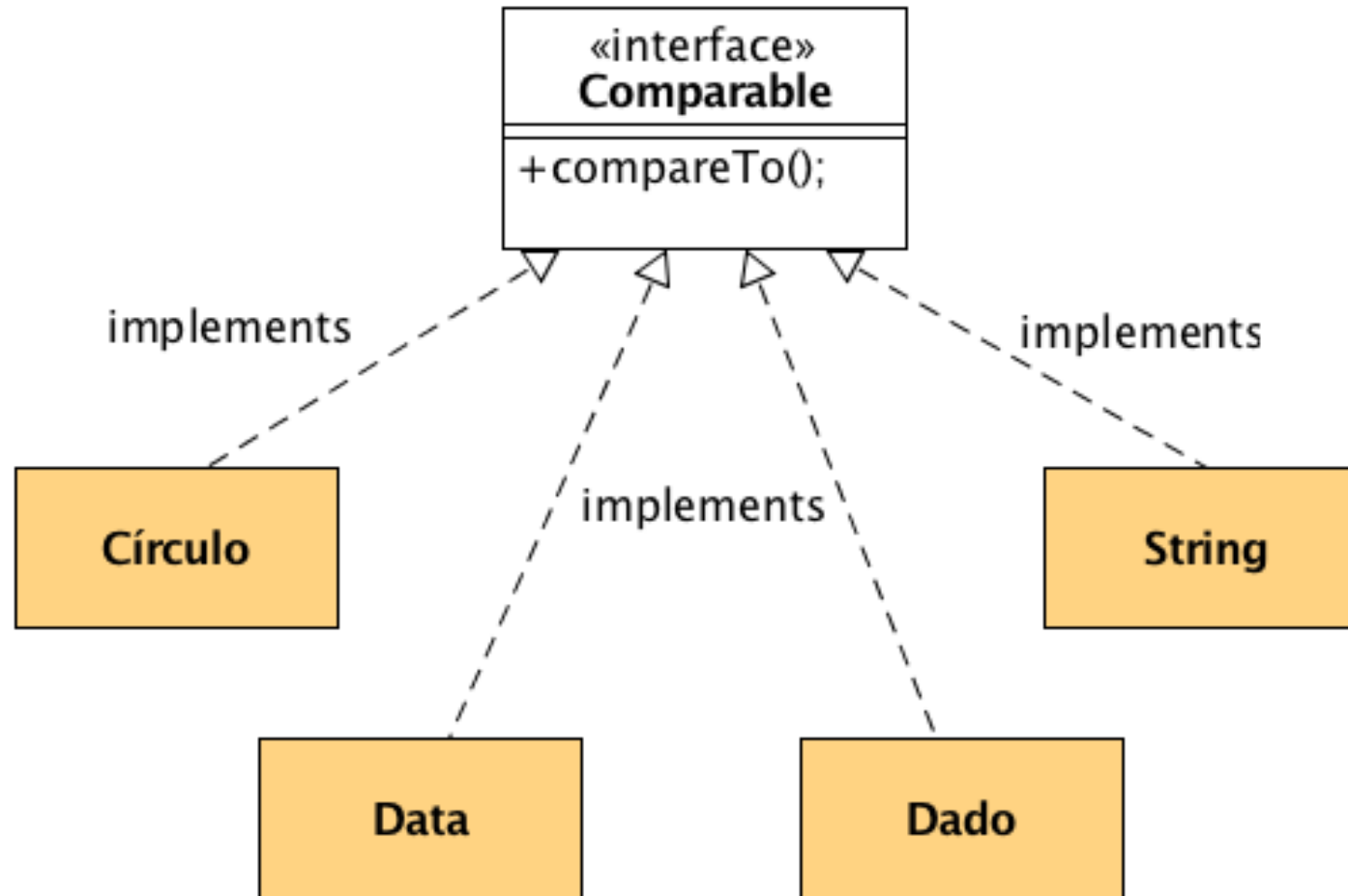
Classes Abstratas versus Interfaces



Classes Abstratas versus Interfaces

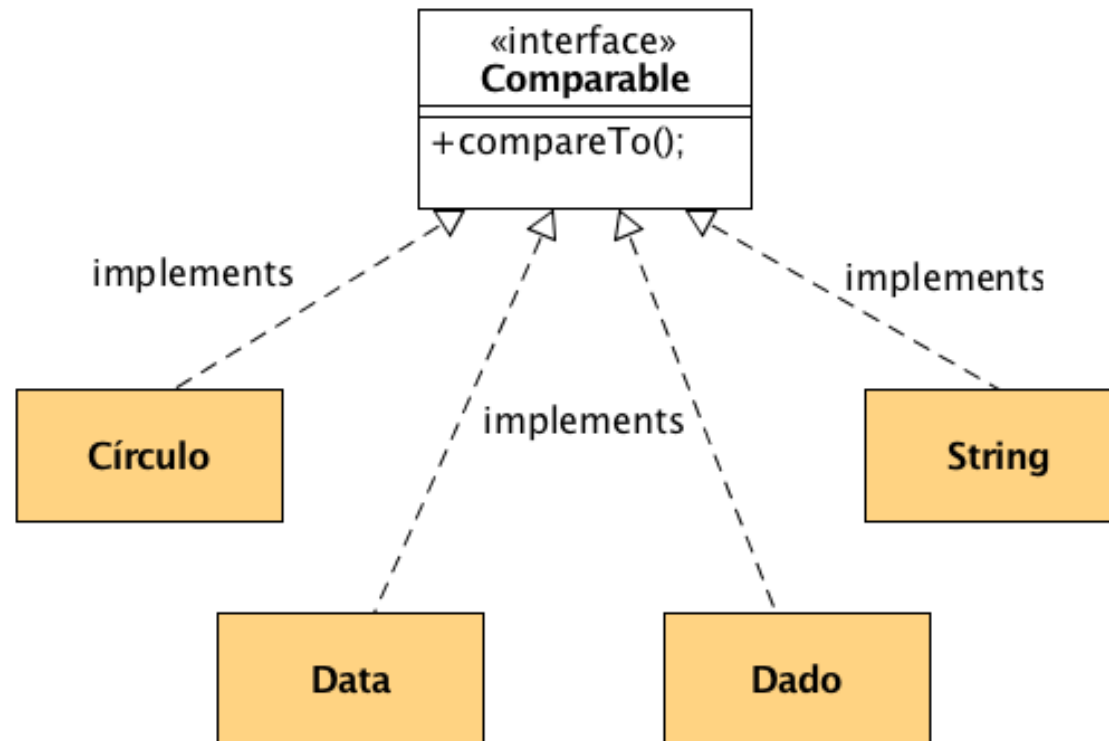


Classes Abstratas versus Interfaces



Questões?

- ❖ Porquê 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

Why comparable?

Comparator and Comparable in Java

Last updated: March 17, 2024



Written by:
baeldung



Reviewed by:
Loredana Crusoveanu

<https://www.baeldung.com/java-comparator-comparable>
https://www.geeksforgeeks.org/comparable-interface-in-java-with-examples/?ref=header_outind
https://www.w3schools.com/java/java_advanced_sorting.asp

Sorting in collections

```
// Person class implementing Comparable
class Person implements Comparable<Person> {
    private String name;
    private int age;
    public Person(String name, int age) { (...) }

    public String getName() (...) }
    public int getAge() { (...) }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age); // Ascending order
    }

    public String toString() {
        return name + " (Age: " + age + ")";
    }
}
```

Sorting in collections

```
// Person class implementing Comparable
class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person other) {
        // Sort the list (uses compareTo method in Person)
        Collections.sort(people);

        System.out.println("Sorted by age:");
        for (Person p : people) {
            System.out.println(p);
        }
    }
}
```

Sorting in collections

```
public class Main {  
    public static void main(String[] args) {  
        List<Person> people = new ArrayList<>();  
        people.add(new Person("Alice", 30));  
        people.add(new Person("Bob", 25));  
        people.add(new Person("Charlie", 35));  
  
        // Custom Comparator for sorting by name  
        Comparator<Person> nameComparator =  
            Comparator.comparing(Person:: getAge);  
  
        // Sort the list using the nameComparator  
        Collections.sort(people, nameComparator);  
  
        System.out.println("Sorted by age:");  
        for (Person p : people) {  
            System.out.println(p);  
        }  
    }  
}
```

Sorting in collections

```
public class Main {  
    public static void main(String[] args) {  
        List<Person> people = new ArrayList<>();  
        people.add(new Person("Alice", 30));  
        people.add(new Person("Bob", 25));  
        people.add(new Person("Charlie", 35));  
  
        // Custom Comparator for sorting by name  
        Comparator<Person> nameComparator =  
            Comparator.comparing(Person:: getName);  
  
        // Sort the list using the nameComparator  
        Collections.sort(people, nameComparator);  
  
        System.out.println("Sorted by name:");  
        for (Person p : people) {  
            System.out.println(p);  
        }  
    }  
}
```

Sorting in collections

```
public class Main {  
    public static void main(String[] args) {  
        List<Person> people = new ArrayList<>();  
        people.add(new Person("Alice", 30));  
        people.add(new Person("Bob", 25));  
        people.add(new Person("Charlie", 35));  
  
        // Custom Comparator that also uses Comparable-style comparison  
        Comparator<Person> nameComparator =  
            (p1, p2) -> p1.getName().compareTo(p2.getName());  
  
        // Sort by name using the custom Comparator  
        Collections.sort(people, nameComparator);  
  
        System.out.println("Sorted by name:");  
        for (Person p : people) {  
            System.out.println(p);  
        }  
    }  
}
```

Sorting in collections

```
public class Main {  
    public static void main(String[] args) {  
        List<Person> people = new ArrayList<>();  
        people.add(new Person("Alice", 30));  
        people.add(new Person("Bob", 25));  
        people.add(new Person("Charlie", 35));  
  
        // Custom Comparator that also uses Comparable-style comparison  
        Comparator<Person> ageComparator =  
            (p1, p2) -> p1.getAge() < p2.getAge ();  
  
        // Sort by name using the custom Comparator  
        Collections.sort(people, ageComparator);  
  
        System.out.println("Sorted by name:");  
        for (Person p : people) {  
            System.out.println(p);  
        }  
    }  
}
```

Sumário

- ❖ Herança
- ❖ Polimorfismo
- ❖ Generalização
- ❖ Classes abstratas
- ❖ Interfaces