



# El Caso del Volador


**Cuándo y por qué usar clases  
abstractas e interfaces**


## Contexto

Tenemos cuatro clases:

 <b>Avion</b>
+ volar() + despegar() + aterrizar()

 <b>Laptop</b>
+ encender() + apagar() + procesar()

 <b>Superman</b>
+ luchar() + volar() + salvarAlMundo()

 <b>Paloma</b>
+ comer() + volar() + dormir()

## Contexto

Y definimos cuatro listas:

- `List<Avion>` **aviones**
- `List<Superman>` **supermen**
- `List<Paloma>` **palomas**
- `List<Laptop>` **laptops**

Cada una de las listas contiene objetos del tipo correspondiente.

## Requerimiento I

*Dadas las 4 listas se pide:*

- A. *Implementar un método **dameTodoLoQueVuela** que reciba listas y devuelva todos los objetos de esas listas que pueden **volar**.*
- B. *Implementar un método que reciba una lista de objetos que pueden **volar** y ejecutar el método correspondiente sobre cada instancia.*

**NIVEL 1: sólo conocemos el  
concepto de clases concretas**

## Requerimiento 1 / sólo clases concretas

```
public List<Object> dameTodoLoQueVuele(List<Superman> supermen, List<Paloma> palomas,  
                                       List<Avion> aviones, List<Laptop> laptops) {  
    ArrayList<Object> voladores = new ArrayList<>();  
    voladores.addAll(supermen);  
    voladores.addAll(palomas);  
    voladores.addAll(aviones);  
    return voladores;  
}
```

## Requerimiento I / sólo clases concretas

```
public void mandarAVolar(List<Object> voladores) {  
    for(Object volador : voladores) {  
        if(volador instanceof Avion) {  
            ((Avion) volador).volar();  
        } else if (volador instanceof Paloma) {  
            ((Paloma) volador).volar();  
        } else if(volador instanceof Superman) {  
            ((Superman) volador).volar();  
        }  
    }  
}
```

## Requerimiento I / sólo clases concretas



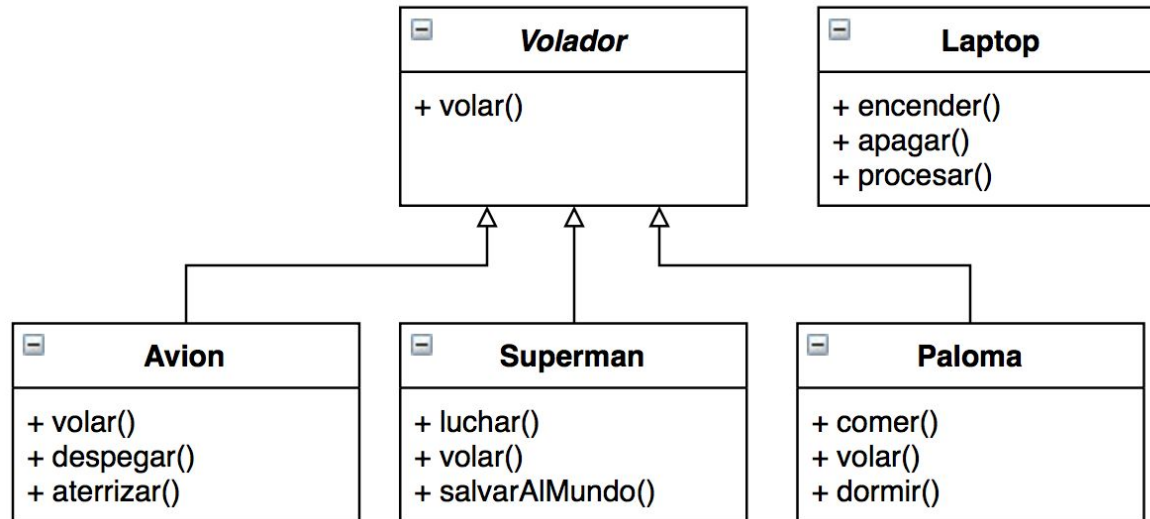
El requerimiento se cumple, pero la implementación no es muy elegante/extensible



## **NIVEL 2: clases abstractas, la salvación?**

## Requerimiento I / clase abstracta

Definimos una clase abstracta para aprovechar herencia y polimorfismo



## Requerimiento 1 / clase abstracta

```
public List<Volador> dameTodoLoQueVuele(List<Superman> supermen, List<Paloma> palomas,  
                                         List<Avion> aviones, List<Laptop> laptops) {  
    ArrayList<Volador> voladores = new ArrayList<>();  
    voladores.addAll(supermen);  
    voladores.addAll(palomas);  
    voladores.addAll(aviones);  
    return voladores;  
}
```

## Requerimiento 1 / clase abstracta

```
public void mandarAVolar(List<Volador> voladores) {  
    for(Volador volador : voladores) {  
        volador.volar();  
    }  
}
```

## Requerimiento I / clase abstracta



El requerimiento se cumple, y la  
implementación es elegante/extensible

## Requerimiento I / clase abstracta



a menos que...

## Requerimiento II

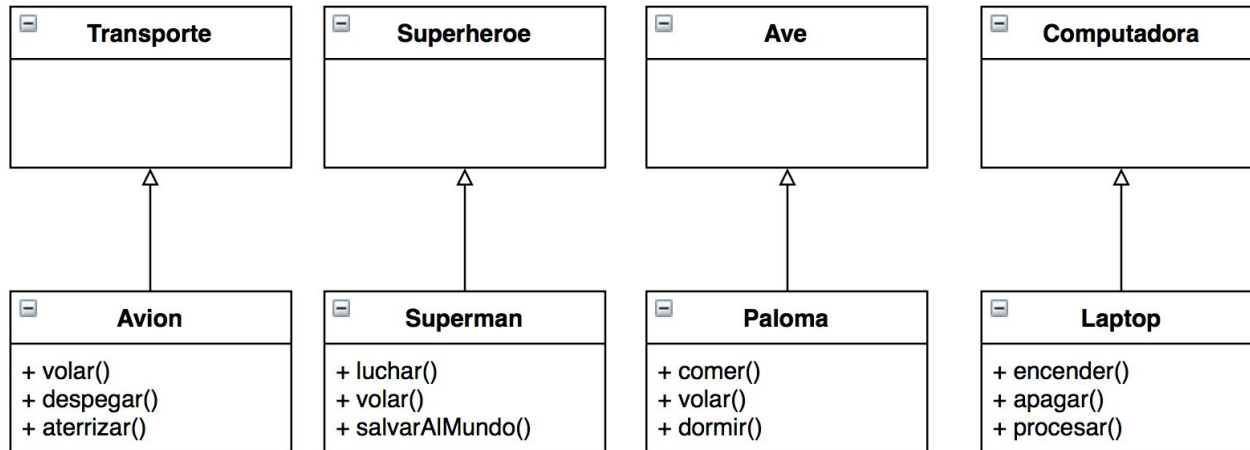
*Extender el dominio para poder abarcar mayor cantidad de elementos:*

- A. Definir nuevo tipo Transporte*
- B. Definir nuevo tipo Superheroe*
- C. Definir nuevo tipo Computadora*
- D. Definir nuevo tipo Ave*

*Adaptar las clases existentes a estas nuevas jerarquías*

## Requerimiento II

Nuevo dominio:





## Requerimiento II

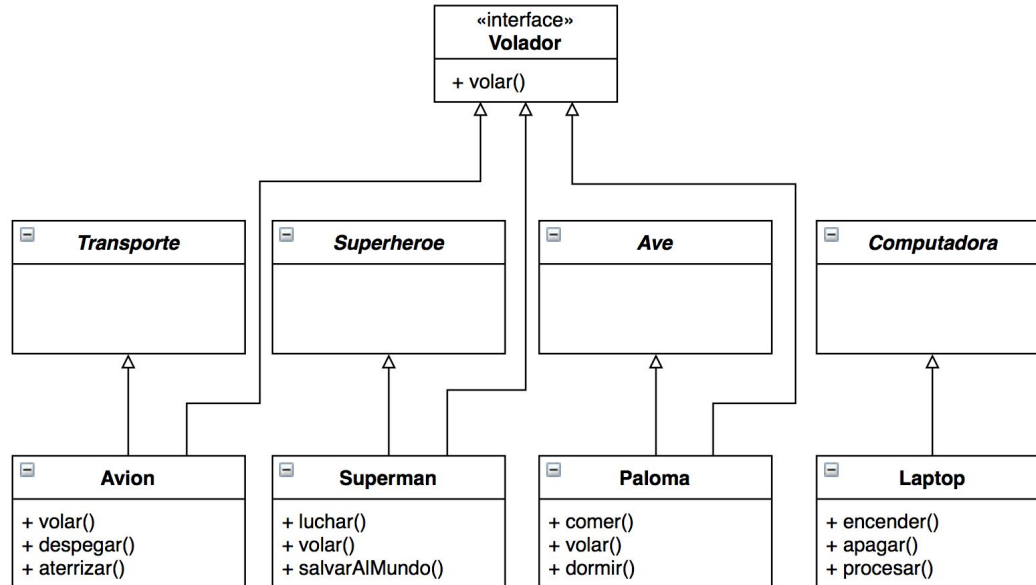


Qué hacemos con **Volador??**  
El lenguaje no soporta herencia múltiple!

## **NIVEL 3: interfaces, la salvación!!**

## Requerimiento II / con interfaz

Defino la **interfaz Volador** con el método `volar()`



## Requerimiento II / con interfaz

```
public List<Volador> dameTodoLoQueVuele(List<Superheroe> superheroes, List<Ave> aves,
                                         List<Transporte> transportes, List<Computadora> computadoras) {
    ArrayList<Volador> voladores = new ArrayList<>();
    voladores.addAll(extraerVoladores(superheroes));
    voladores.addAll(extraerVoladores(aves));
    voladores.addAll(extraerVoladores(transportes));
    voladores.addAll(extraerVoladores(computadoras));
    return voladores;
}

private List extraerVoladores(List<? extends Object> items) {
    return items.stream()
        .filter(item -> item instanceof Volador)
        .collect(Collectors.toList());
}
```

## Requerimiento II / con interfaz

```
public void mandarAVolar(List<Volador> voladores) {  
    for(Volador volador : voladores) {  
        volador.volar();  
    }  
}
```

## Requerimiento II / con interfaz



Ambos requerimientos se cumplen, y la implementación es elegante/extensible

## Conclusión

Usamos **HERENCIA** cuando definimos una FAMILIA DE CLASES: relación “**ES UN**”

Usamos **INTERFAZ** cuando queremos agrupar clases que tienen **comportamiento en común**