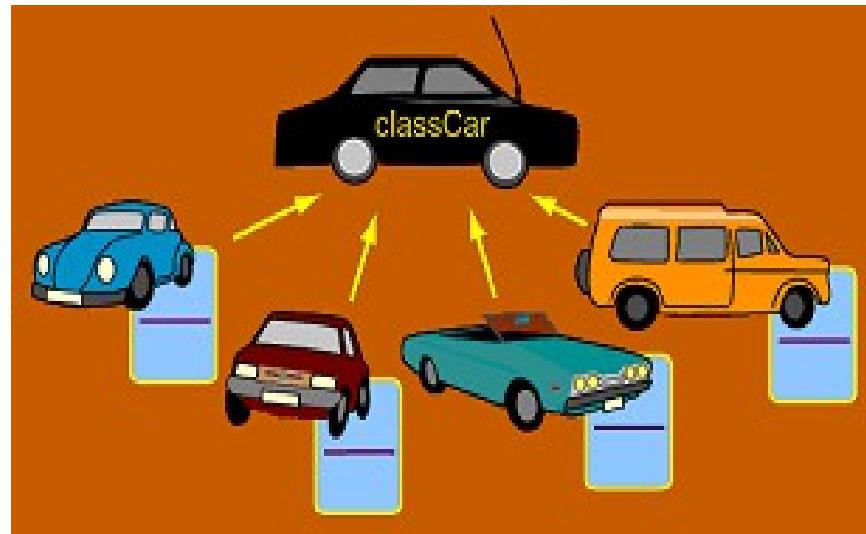


# Tema 3



## Reutilización y polimorfismo

# Contenidos



Lección	Título	Nº sesiones
3.1	Mecanismos de reutilización de código	2
3.2	Mecanismos de reutilización en UML	1
3.3	El polimorfismo	2

[http://groups.diigo.com/group/pdoo\\_ugr](http://groups.diigo.com/group/pdoo_ugr)



# Lección 3.2

Mecanismo de reutilización en  
UML

# Objetivos de aprendizaje



- Conocer la representación gráfica de la herencia en un diagrama de clases del diseño.
- Conocer la representación gráfica de las clases abstractas en un diagrama de clases del diseño
- Conocer la representación gráfica de las interfaces y sus relaciones en un diagrama de clases del diseño.
- Implementar una estructura de clases e interfaces con herencia a partir de un diagrama de clases del diseño dado.
- Conocer la representación de una clase parametrizada en UML.

# Contenidos

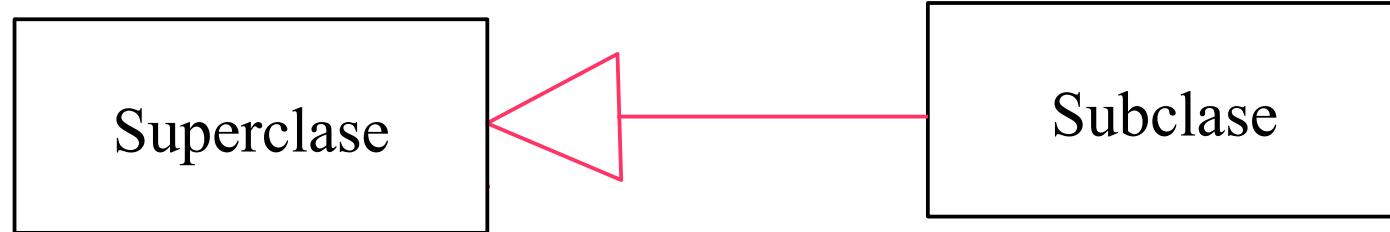


Representación en UML de:

1. Herencia.
2. Clase y método abstracto.
3. Herencia múltiple.
4. Interfaz y sus relaciones.
5. Clase parametrizada.

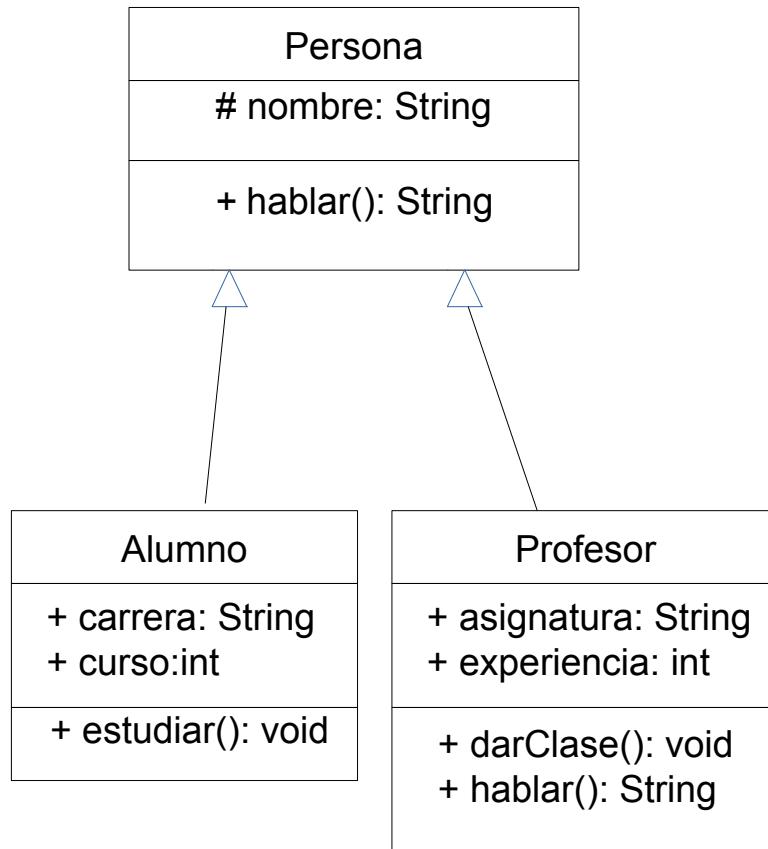
# 1. Herencia

La herencia en UML se representa como una relación entre dos clases y se denota con un símbolo especial:



En las subclases, sólo se indican las variables y los métodos nuevos que declaran. También se indican los métodos que, aunque figuren en la superclase, se redefinen en la subclase.

# 1. Herencia



Implementar este  
diagrama en Ruby



```
public class Persona {
    protected String nombre;
    public String hablar(){...}
}

public class Alumno extends Persona{
    public String carrera;
    public int curso;
    public void estudiar(){...}
}

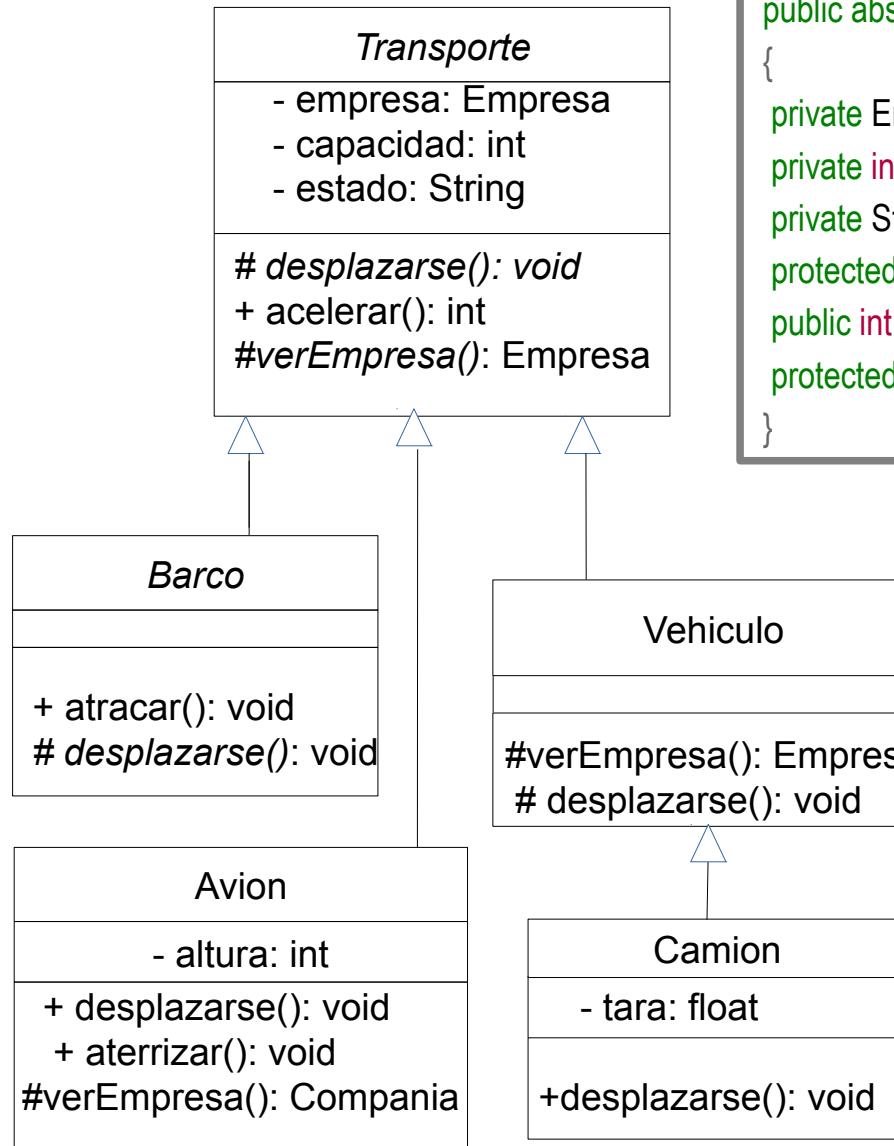
public class Profesor extends Persona {
    public String asignatura;
    public int experiencia;
    @Override
    public String hablar(){...}
    public void darClase(){...}
}
```

## 2. Clase y método abstracto

---

- El **nombre de la clase** aparece en **cursiva**.
- Los **métodos abstractos** (no implementados) aparecen en **cursiva**.
- Una clase abstracta debe ser heredada por subclases.
- La cabecera de un método de una clase abstracta debe coincidir en las subclases que lo heredan. La visibilidad puede modificarse en las subclases siempre que sea a mayor. También el tipo de retorno, que puede ser un subtipo en las subclases.
- Al implementar, en Java se usa la palabra **abstract**.

# 2. Clase y método abstracto



```

public abstract class Transporte {
    private Empresa empresa;
    private int capacidad;
    private String estado;
    protected abstract void desplazarse();
    public int acelerar() {}
    protected abstract Empresa verEmpresa();
}
  
```

```

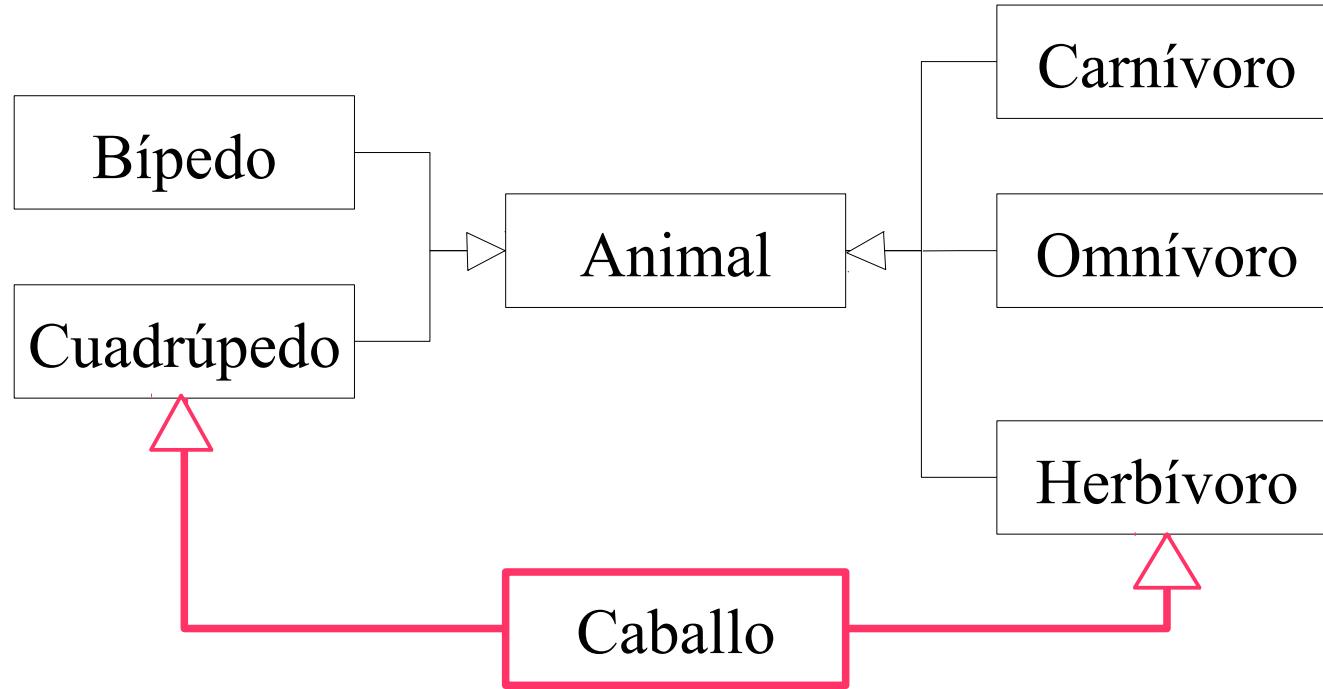
public class Avion extends Transporte {
    private int altura;
    @Override
    public void desplazarse(){...}
    public void aterrizar(){...}
    @Override
    protected Compania verEmpresa(){...}
}
//Compania es subclase de Empresa
  
```

The code shows the implementation of the *Transporte* class. It includes private fields for *empresa*, *capacidad*, and *estado*, and protected abstract methods *desplazarse()*, *acelerar()*, and *verEmpresa()*. The *Avion* class extends *Transporte* and overrides the abstract methods. A note at the bottom states that *Compania* is a subclass of *Empresa*.

Completa la implementación



### 3. Herencia múltiple



Recordad:  
¿Qué lenguajes la proporcionan?  
¿Cómo se podría simular?



# 4. Interfaz y sus relaciones

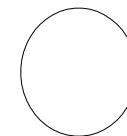
**Interfaz**, dos formas de representarla:

- Como una clase estereotipada, sobre el nombre de la interfaz aparece la palabra <<interface>>.
- Como un círculo, con su nombre y opcionalmente los métodos.

<<interface>>

FiguraGrafica

+ *pintarBorde(): void*



FiguraGrafica

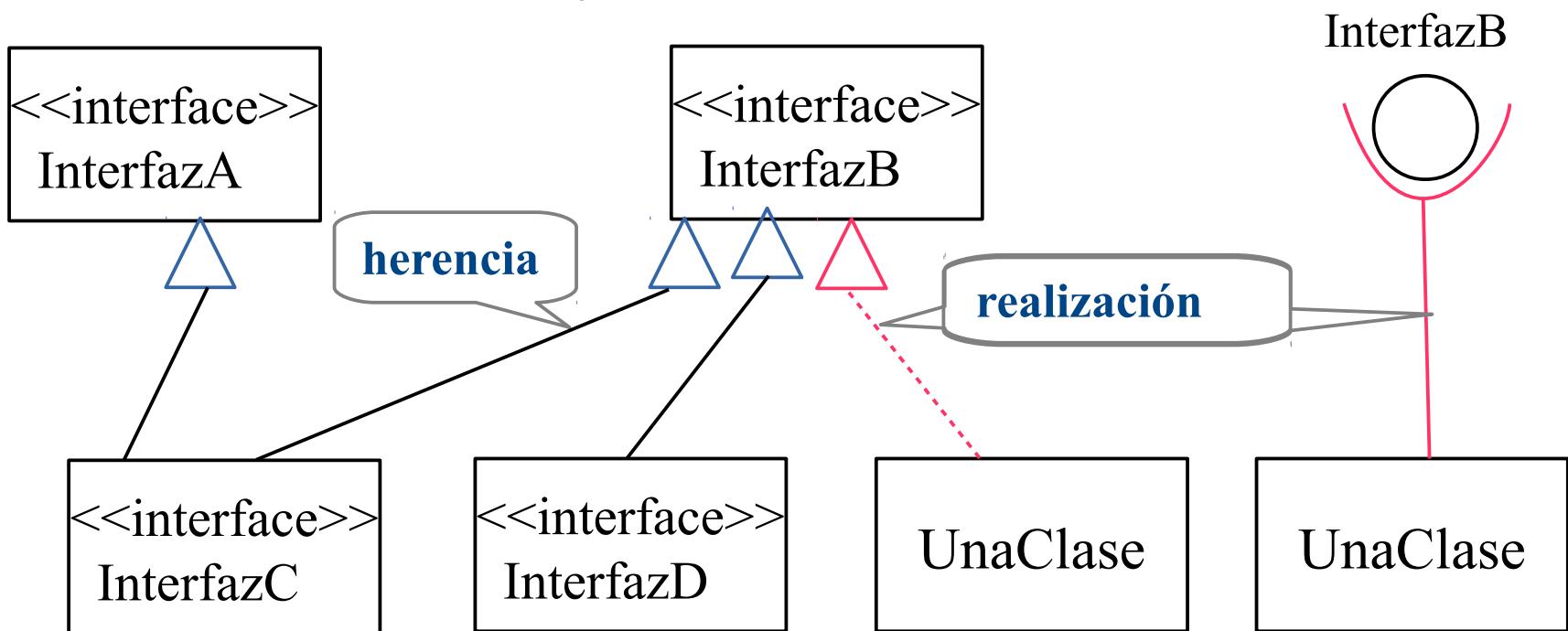
¿Cuándo se utiliza una u otra representación?



# 4. Interfaz y sus relaciones

## Relaciones

- Relación de **Generalización/herencia** con una o varias interfaces
- Relación de **realización** por una o varias clases (clases que la implementan).



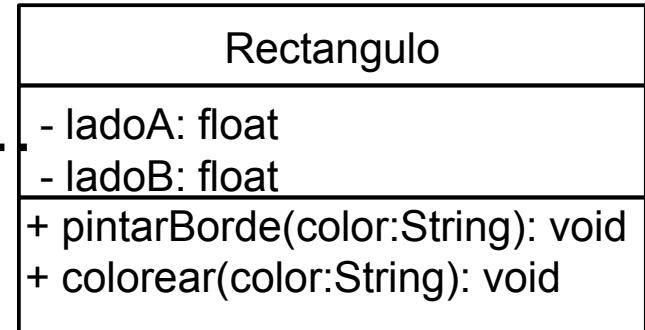
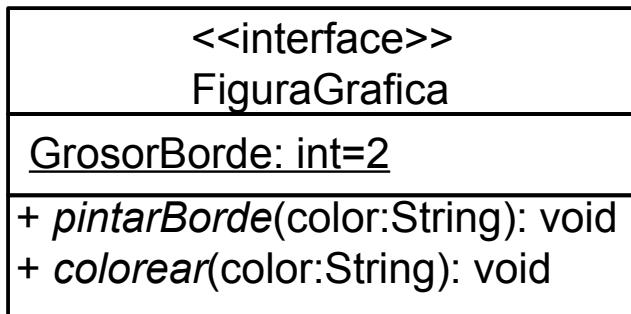
## 4. Interfaz y sus relaciones: Implementación

Recordad, en Java:

- Antes del nombre de la interfaz aparece la palabra **interface**.
- Cuando una clase implementa una interfaz, al declarar la clase se usa la palabra clave **implements** seguida del nombre de la interfaz.
- Todas las variables definidas en una interfaz automáticamente son definidas como globales y constantes (static y final) aunque no se indique explícitamente, es por lo que deben tener un valor asignado, además su visibilidad es pública.
- Cuando una clase implementa una interfaz tiene que implementar todos sus métodos, si no lo hace sería una clase abstracta.

Recordad, en Ruby no existe el concepto de interfaz

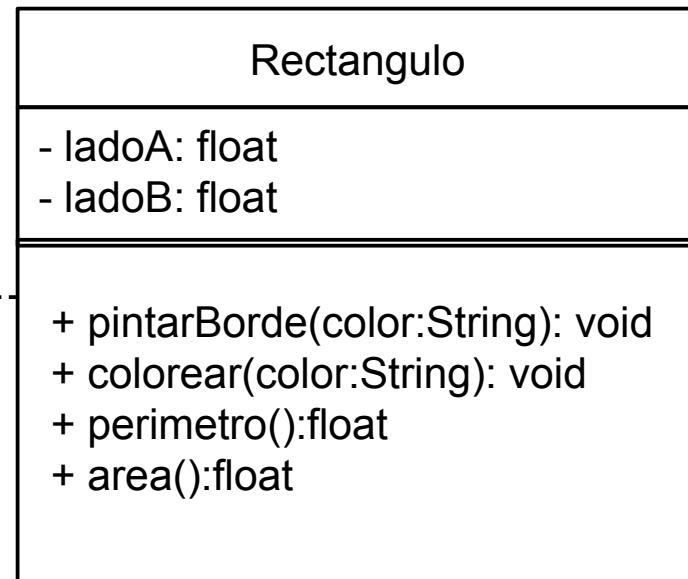
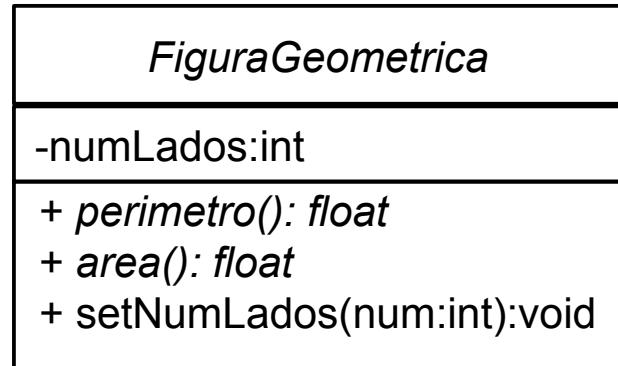
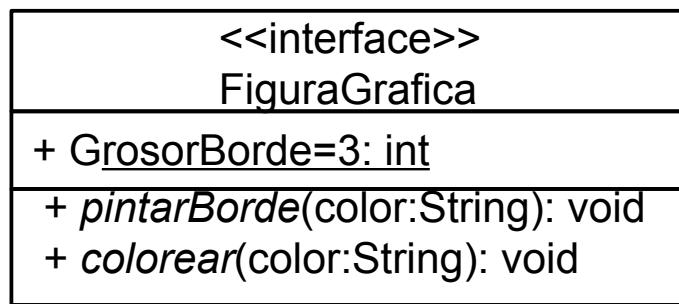
# 4. Interfaz y sus relaciones: Implementación



```
public interface FiguraGrafica {
    static int GrosorBorde=2; // variable global al paquete
    public void pintarBorde(String color);
    public void colorear(String color);
}
```

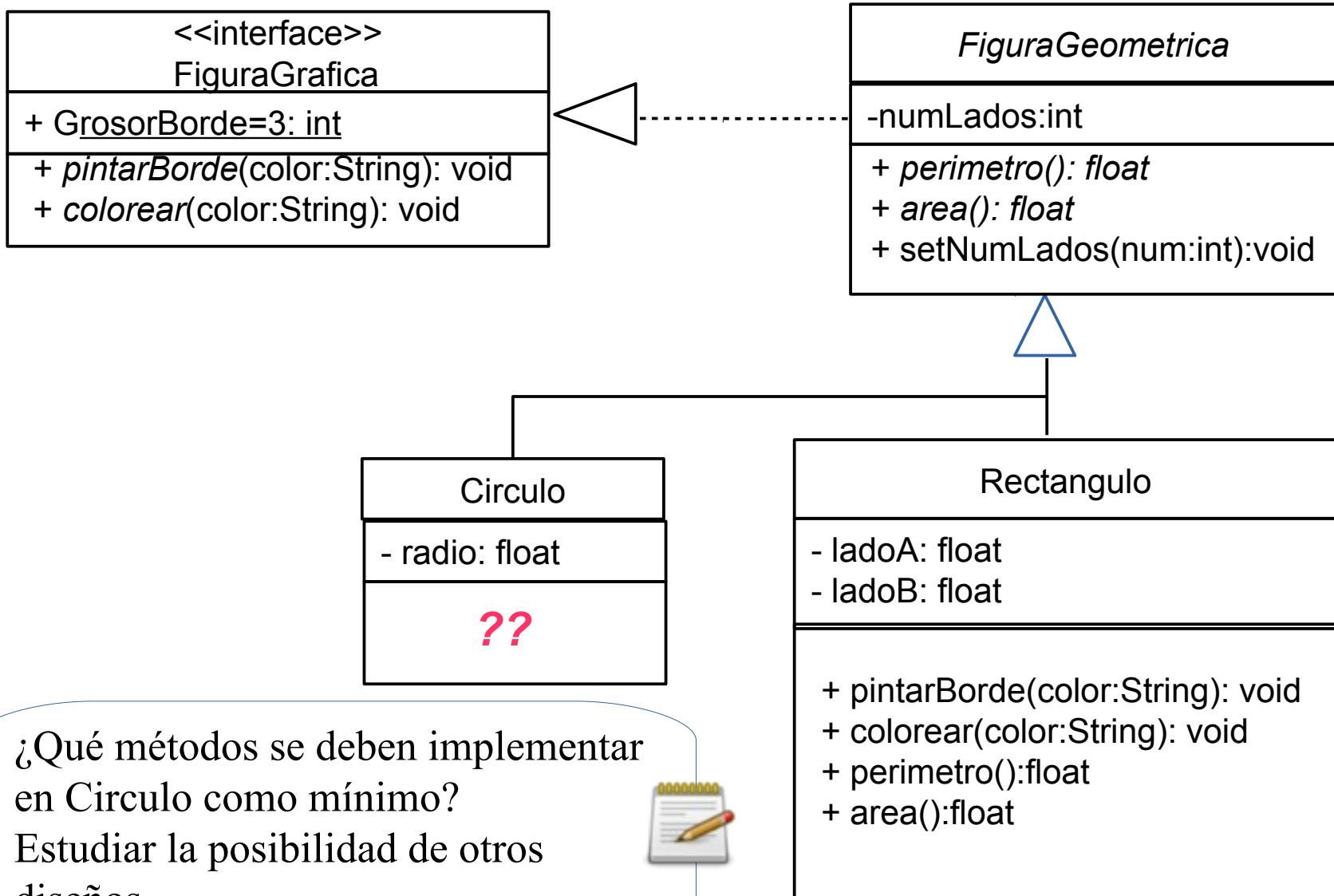
```
public class Rectangulo implements FiguraGrafica {
    private float ladoA;
    private float ladoB;
    public void colorear(String color){...}
    public void pintarBorde(String color){...}
}
```

## 4. Interfaz y sus relaciones: “herencia múltiple”

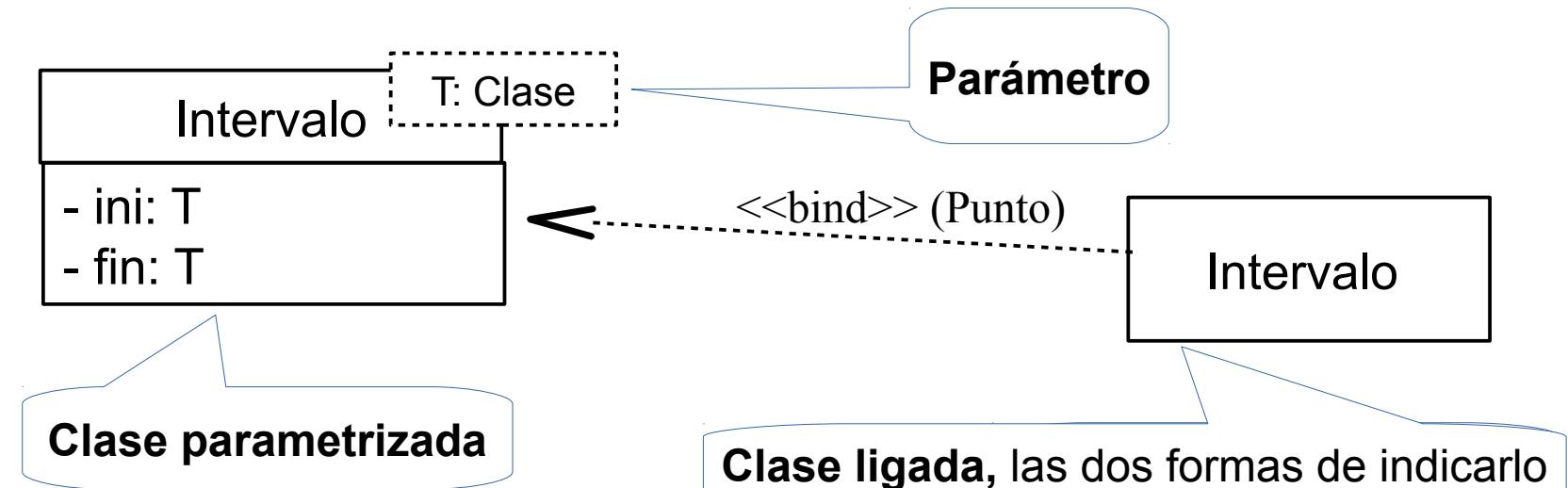


Implementa este  
diagrama de  
clases en Java

## 4. Interfaz y sus relaciones: Ejemplo



# 5. Clase parametrizada



```
class Intervalo<T> {  
    T inicio, fin;  
    public Intervalo (T ini, T fin)  
        { inicio=ini; fin=fin; }  
    public T getInicio() {return inicio;}  
    public void setInicio(T ini) {inicio = ini;}  
    public T getFin() {return fin;}  
    public void setFin(T f ) {fin = f; }  
}
```

```
Intervalo<Punto> segmento= new Intervalo(a,b);
```