

```
mirror_mod = modifier_ob.modifiers.new("mirror_mod")
# add mirror object to mirror_ob
mirror_mod.mirror_object = mirror_ob

operation == "MIRROR_X":
    mirror_mod.use_x = True
    mirror_mod.use_y = False
    mirror_mod.use_z = False
operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

# selection at the end -add back the deselected
mirror_ob.select= 1
modifier_ob.select=1
key.context.scene.objects.active = modifier_ob
print("selected" + str(modifier_ob)) # modifier ob
mirror_ob.select = 0
key = key.context.selected_objects[0]
key.name.objects[one.name].select = 1

print("please select exactly two objects, ok")
```

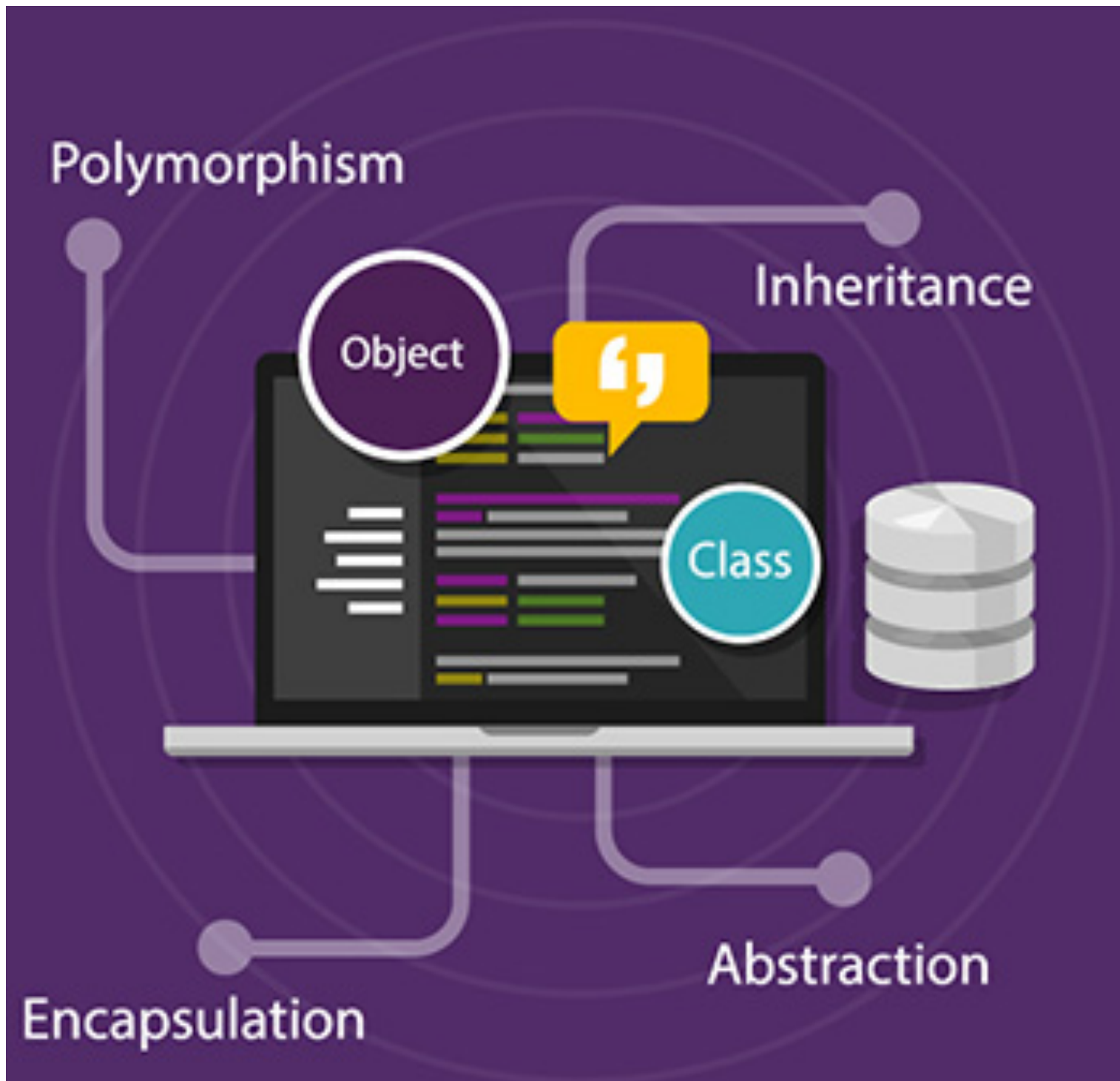
OPERATOR CLASSES -----

```
class MirrorOperator(operator.Operator):
    """Mirror object to the selected object"""
    def __init__(self, context, mirror_mirror_x):
        self.context = context
        self.mirror_x = mirror_mirror_x
```

```
def __call__(self, context):
    """Mirror object to the selected object"""
    if context.selected_active_object is not None:
```

Desarrollo de Interfaces

Repaso de Java



POO

La programación orientada a objetos (POO) es un paradigma de programación que consiste en modelar la realidad como un conjunto de **objetos que interactúan entre sí** para resolver un problema.

Propiedades —> Estructuras de datos (Atributos)

Comportamiento —> Algoritmos (Métodos)

DOG

class

Breed
Size
Age
Color

Data
members

Eat()
Sleep()
Sit()
Run()

Methods

Clases y Objetos

Clase —> Entidad sintáctica que describe objetos que van a tener la misma estructura y el mismo comportamiento.

Objeto —> Es una instancia de una clase creada en tiempo de ejecución.

```
public class Dog {  
    private String breed;  
    private float size;  
    private int age;  
    private String color;  
  
    public void Sleep() {  
        System.out.println("Zzzzz...");  
    }  
    . . .  
}  
public static void main(String[] args) {  
    Dog bimbo = new Dog();  
    while(true) {  
        bimbo.Sleep();  
    }  
}
```



Genericidad

```
public class Caja<T> {  
    private T contenido;  
  
    public void setContenido(T c) {  
        this.contenido = c;  
    }  
  
    public T getContenido() {  
        return contenido;  
    }  
}  
  
public static void main(String[] args) {  
    Caja<Integer> cajaEntero = new Caja<Integer>();  
    Caja<String> cajaString = new Caja<String>();  
  
    cajaEntero.setContenido(new Integer(10));  
    cajaString.setContenido(new String("Hello World"));  
}
```



Polimorfismo

Sobrecarga

```
public class Polimorfismo
{
    void demo (int a)
    {
        System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
        System.out.println ("a y b: " + a +
", " + b);
    }
}

public static void main (String args [])
{
    Polimorfismo pol = new
Polimorfismo();
    pol.demo(10);
    pol.demo(10, 20);
}
```

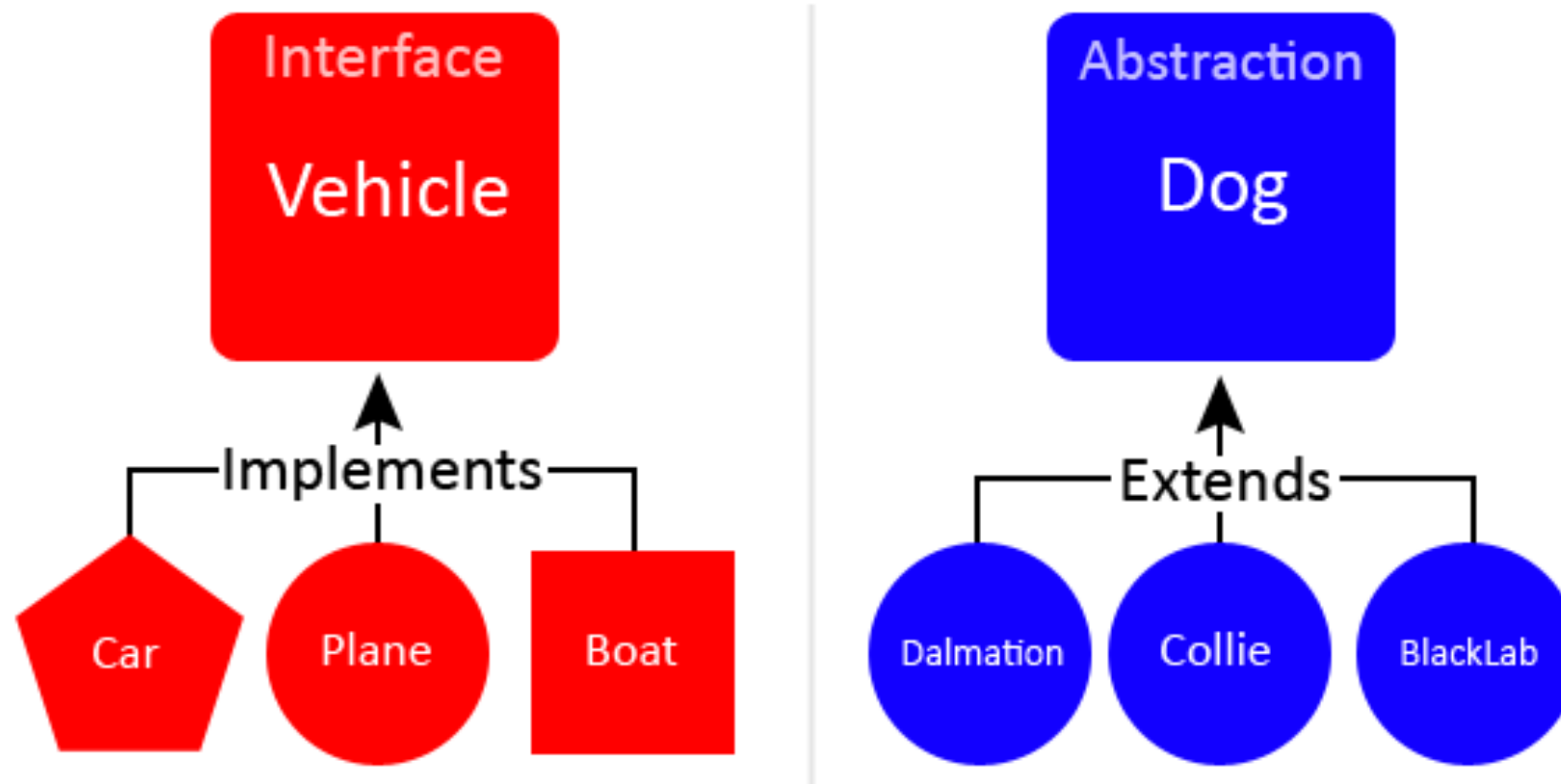
Polimorfismo

Sobreescritura



```
public class Animal
{
    public void sonido()
    {
        System.out.println("Un animal
        haciendo un sonido");
    }
}

public class Gato extends Animal
{
    @Override
    public void sonido()
    {
        System.out.println("Miau");
    }
}
```

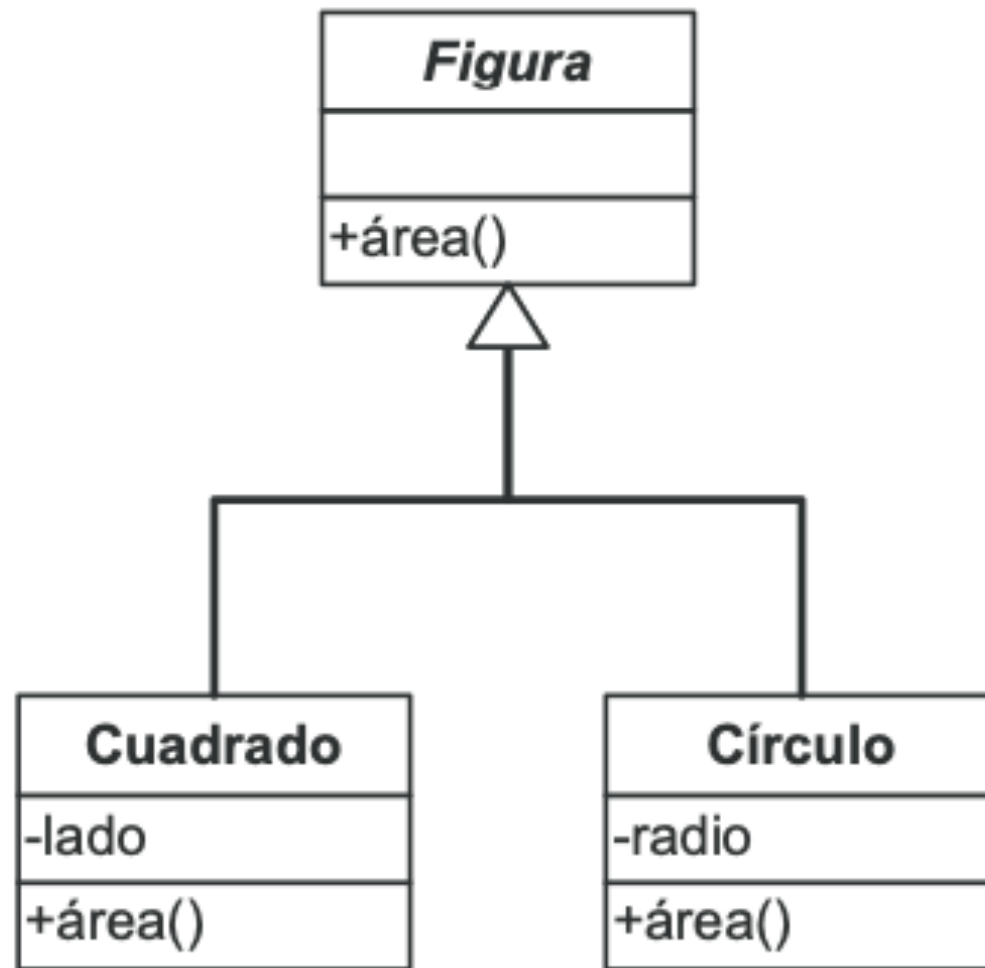


Clases Abstractas

Clase base que **no se puede instanciar** y que únicamente se emplea para definir otras subclases. Su función por tanto es la de **agrupar parámetros y métodos comunes** de las clases que se derivan de ella, simplificando el código y mejorando su jerarquía y comprensión.

Los métodos de una clase abstracta carecen de implementación, ya que deben ser implementados en sus clases derivadas. A su vez, las clases que hereden de una clase abstracta y no implemente todos los métodos declarados en la clase base, también será abstracta.

Clases Abstractas



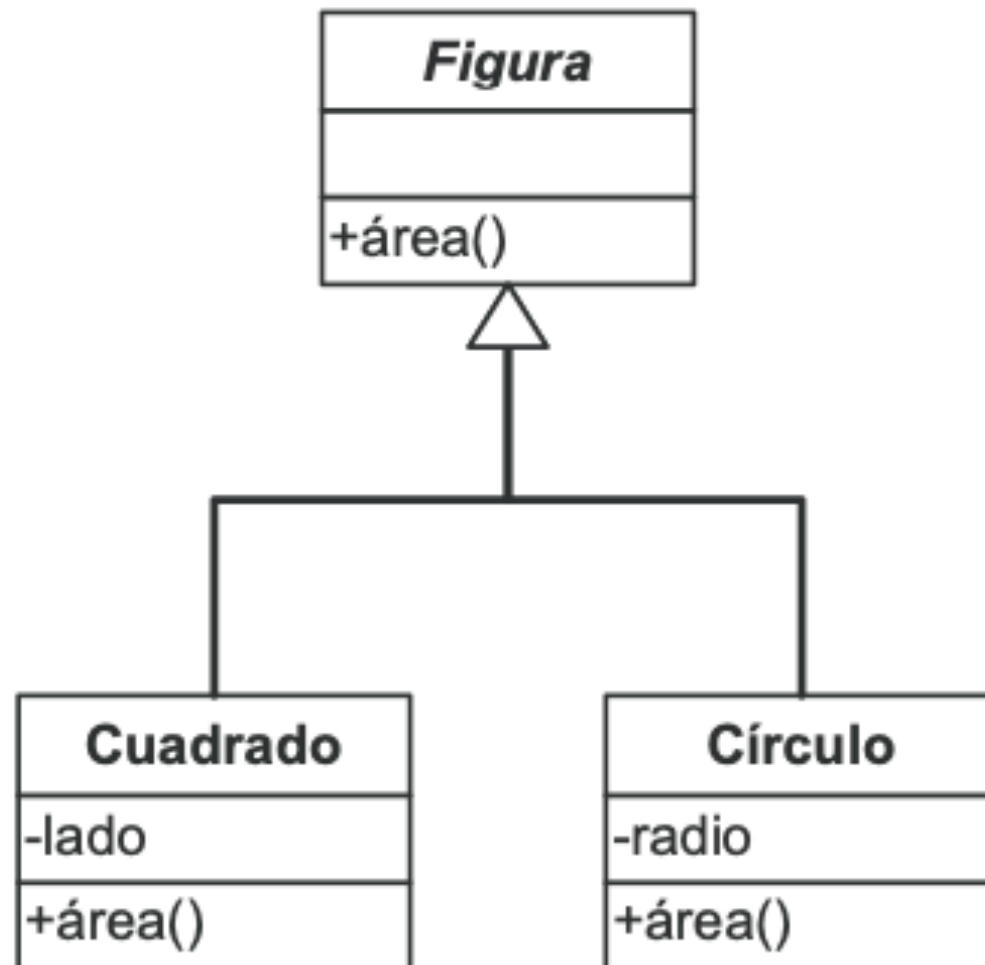
```
public abstract class Figura {
    protected double x;
    protected double y;
    public Figura(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public abstract double area();
}

public class Circulo extends Figura {
    private double radio;
    public Circulo(double x, double y, double radio) {
        super(x, y);
        this.radio = radio;
    }
    public double area() {
        return Math.PI * radio * radio;
    }
}
```

```
public class Cuadrado extends Figura {
    private double lado;
    public Cuadrado(double x, double y, double lado) {
        super(x, y);
        this.lado = lado;
    }
    public double area() {
        return lado * lado;
    }
}
```


Interfaces

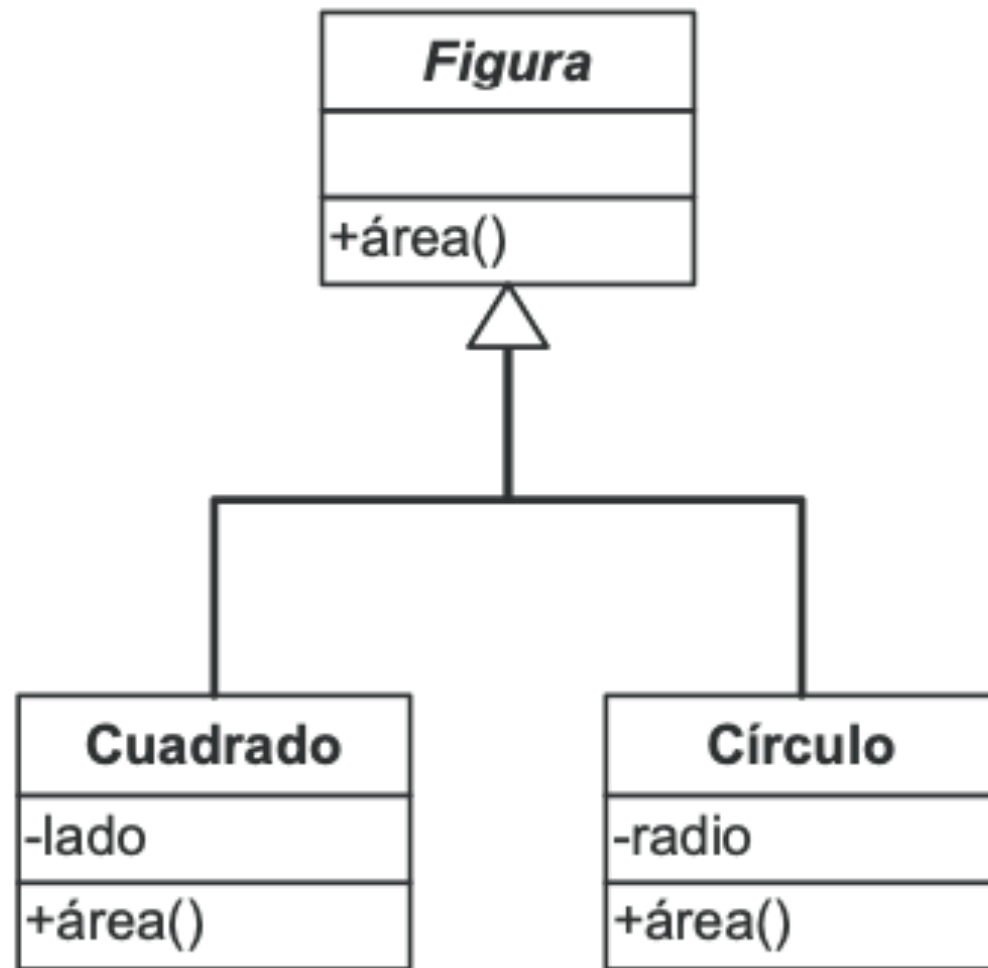
Si en el ejemplo anterior no estuviésemos interesados en conocer la posición de una Figura, podríamos eliminar por completo su implementación, y pasaríamos a tener lo que se conoce como una **interfaz**.



Es decir, una interfaz no es más que **una clase completamente abstracta** (una clase sin implementación), y por tanto únicamente constará de una serie de declaraciones de métodos (formados por su nombre y firma, sin implementación).

Por lo tanto, una interfaz no encapsula datos, **solo define cuáles son los métodos que han de implementar los objetos de aquellas clases que implementen la interfaz.**

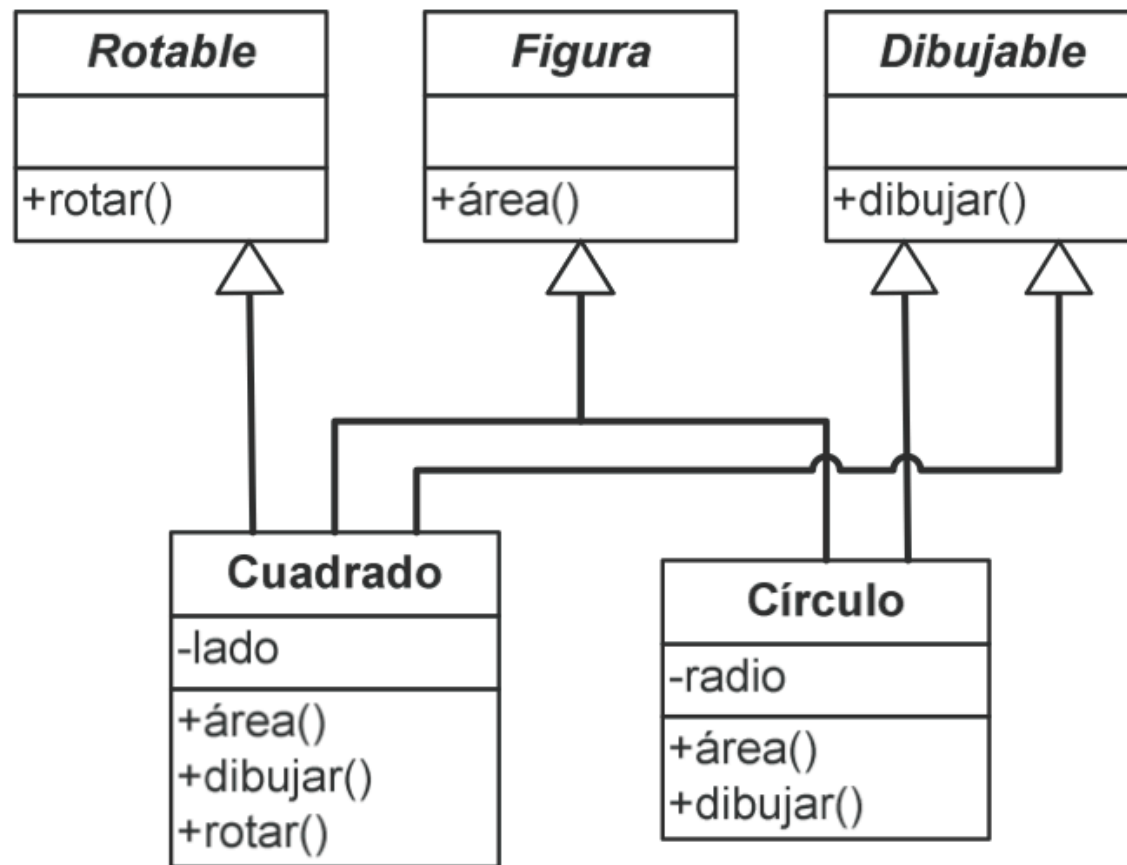
Interfaces



```
public class Circulo implements Figura {
    private double radio;
    public Circulo(double radio) {
        this.radio = radio;
    }
    public double area() {
        return Math.PI * radio * radio;
    }
}
```

```
public class Cuadrado implements Figura {
    private double lado;
    public Cuadrado(double lado) {
        this.lado = lado;
    }
    public double area() {
        return lado * lado;
    }
}
```

Interfaces



```
public abstract class Figura {
    public abstract double area();
}

public interface Dibujable {
    public void dibujar();
}

public interface Rotable {
    public void rotar(double grados);
}

public class Circulo extends Figura implements Dibujable
{
    ...
}

public class Cuadrado extends Figura implements
Dibujable, Rotable
{
    ...
}
```