



**CICLO 2**

[FORMACIÓN POR CICLOS]


# Interfaces y **CLASES ABSTRACTAS**



**Ingeni@**  
Soluciones TIC



**UNIVERSIDAD  
DE ANTIOQUIA**  
Facultad de Ingeniería

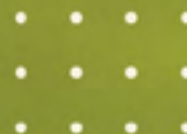


Como ya hemos visto, la herencia es un mecanismo muy útil para la reutilización de código en el lenguaje Java. Sin embargo, esta no es su única función. La abstracción y el modelado de conceptos y situaciones del mundo real se enriquecen enormemente con la herencia. Aun así, podemos ir más allá en estos últimos propósitos. Dos mecanismos útiles en esto son las interfaces y las clases abstractas.

## Clases abstractas

Existen conceptos en el mundo real que podemos calificar como abstractos en el sentido de que no siempre es fácil relacionarlos con un individuo o un grupo de individuos u objetos concreto. Es decir, entendemos a qué se refieren dichos conceptos, pero no siempre es posible pensar en un ejemplo concreto y exacto de los mismos. Podemos pensar en varios ejemplos:

- El concepto de vehículo tiene diferentes acepciones. Sin embargo, la más común dice que un vehículo es algo que las personas utilizamos para desplazarnos de un lugar a otro. Si nos ponemos a pensar, no es fácil dar un ejemplo exacto de lo que es un vehículo, dado que hay vehículos tan dispares como lo son carros, motos, botes y hasta helicópteros o aviones. Un vehículo puede ser cualquiera de esos y muchos otros que cumplan con el criterio del desplazamiento.
- Otro concepto similar es el de ave. De manera intuitiva, podríamos decir que todas las aves tienen alas y plumas, ponen huevos y cuentan con un pico. Y aunque todas pueden agitar sus alas, algunas ni siquiera pueden volar, otras vuelan poco, y otras pueden volar grandes distancias y a gran altura. Por eso podemos decir que, aunque todas las aves tienen características comunes y pueden hacer casi lo mismo, todas tienen aspectos muy diferentes y hacen ciertas cosas de forma distinta.
- Y otro concepto, entre muchos más, es el de figura geométrica. Todas las figuras geométricas tienen unas dimensiones, y para todas podemos calcular el área y el perímetro. Pero para todas, las dimensiones son diferentes, y el área y el perímetro se calculan diferente para cada tipo de figura geométrica.



Para modelar en el lenguaje Java conceptos como estos, contamos con las clases abstractas.

Una **clase abstracta** es una clase que cuenta con uno o varios métodos abstractos. Un método abstracto es aquel en el cual se declara el nombre, su tipo de retorno, y sus parámetros, pero se especifica como abstracto y como tal no tiene cuerpo (no cuenta como implementación). La idea es que son las subclases (clases hijas) de las clases abstractas las encargadas de dar cuerpo (implementar) el o los métodos abstractos. Además, una clase abstracta no se puede instanciar. Las que sí se podrían instanciar son sus clases hijas. En todo lo demás, una clase abstracta funciona como una clase padre cualquiera.

Para declarar una clase o un método como abstractos se usa la palabra clave `abstract`, tal como podemos observar a continuación.

```
package co.edu.udea.udea_ruta2_ciclo2.poo;

public abstract class Figura {
    public abstract double getArea();
    public abstract double getPerimetro();
}
```

Podemos notar que la clase abstracta `Figura` cuenta con dos métodos abstractos: `getArea` y `getPerimetro`. Pero al ser abstractos no cuentan con un cuerpo, el cual debe ser provisto por sus clases hijas. Después de todo, no sabemos cómo se halla el área de una “figura” geométrica en general, pero sí de una en particular (como un círculo o un triángulo). Por lo anterior, una subclase de una clase abstracta, deberá implementar todos los métodos abstractos de dicha clase, y si no lo hace, la subclase deberá ser también abstracta. A continuación, podemos observar como dos subclases implementarían los métodos abstractos de la clase `Figura` de forma diferente.

```
package co.edu.udea.udea_ruta2_ciclo2.poo;
```

```
public class Cuadrado extends Figura {
```

```
    private double lado;
```

```
    public Cuadrado() {  
        this.lado = 0;  
    }
```

```
    public Cuadrado(double lado) {  
        this.lado = lado;  
    }
```

```
    public double getLado() {  
        return lado;  
    }
```

```
    public void setLado(double lado) {  
        this.lado = lado;  
    }
```

```
    @Override
```

```
    public double getArea() {  
        return this.lado * this.lado;  
    }
```

```
    @Override
```

```
    public double getPerimetro() {  
        return this.lado * 4;  
    }  
}
```



```

package co.edu.udea.udea_ruta2_ciclo2.poo;

public class Circulo extends Figura {
    private double radio;

    public Circulo() {
        this.radio = 0;
    }

    public Circulo(double radio) {
        this.radio = radio;
    }

    public double getRadio() {
        return radio;
    }

    public void setRadio(double radio) {
        this.radio = radio;
    }

    @Override
    public double getArea() {
        return Math.PI * this.radio * this.radio;
    }

    @Override
    public double getPerimetro() {
        return 2 * Math.PI * this.radio;
    }
}

```

Las clases abstractas tienen un papel importante en algunas librerías usadas en el lenguaje Java, así como en el polimorfismo, que exploraremos posteriormente.

## Interfaces

Por otro lado, tenemos las interfaces. Aunque las interfaces tienen ciertas similitudes con las clases abstractas, su motivación y funcionamiento son bastante diferentes.

En la actualidad, el desarrollo de software se basa, no solo en la codificación de algoritmos, sino en el ensamblaje de diferentes componentes preexistentes. Esto ha traído la necesidad de que librerías, *frameworks* y clases de diferentes desarrolladores interactúen entre sí de una manera natural, pero que no genere dependencia entre ellos. Para lograr lo anterior, la interacción entre componentes de software se da mediante una especie de *contratos* que nos especifican qué podemos hacer con un componente de software específico, sin necesidad de saber cómo funciona dicho componente por dentro. Dichos *contratos* se denominan interfaces.

En Java, una interfaz es un tipo similar a una clase (no confundir con interfaces gráficas de usuario), que sólo puede contener:

- Constantes (ya que los atributos que les agreguemos serán `public static final` de manera implícita y por defecto).
- Declaraciones de métodos públicos (es decir, sin cuerpo), que por defecto y de manera implícita serán `abstract final`.
- Métodos estáticos.
- Métodos default.

En la mayoría de los casos, las interfaces cuentan con las dos primeras opciones, es decir, constantes y declaraciones de métodos. Además, de manera similar a como pasa con las clases abstractas, una interfaz no puede ser instanciada, sino que puede ser implementada por una clase o bien ser heredada por otra interfaz.

Para declarar una interfaz, se usa la palabra clave `interface`.

```
package co.edu.udea.udea_ruta2_ciclo2.poo;  
  
public interface VolumenGraduable {  
  
}
```

Y como mencionamos arriba, suelen tener declaraciones de métodos (sin cuerpo).

```
package co.edu.udea.udea_ruta2_ciclo2.poo;

public interface VolumenGraduable {
    public void subirVolumen();
    public void bajarVolumen();
}
```

Ahora cabe preguntarnos. Una vez declarada, ¿cómo se utiliza una interfaz? La respuesta es que las interfaces no se instancian, ni las clases heredan de ellas; una interfaz se implementa. Es decir, una clase puede implementar una o varias interfaces, usando la palabra clave implements. Y si una clase implementa una interfaz, deberá implementar, es decir, darle cuerpo, a todos los métodos de dicha interfaz. A continuación, se muestra cómo dos clases implementan la interfaz anterior. Al hacerlo, deben implementar, como mínimo, los métodos de dicha interfaz.

```
package co.edu.udea.udea_ruta2_ciclo2.poo;

public class Parlante implements VolumenGraduable {
    //Código de atributos propios del parlante
    //Código de constructores propios del parlante
    //Código de métodos propios del parlante

    @Override
    public void subirVolumen() {
        //Sube el volumen del parlante
    }

    @Override
    public void bajarVolumen() {
        //Sube el volumen del parlante
    }
}
```

```
package co.edu.udea.udea_ruta2_ciclo2.poo;

public class Televisor implements VolumenGraduable {
    //Código de atributos propios del televisor
    //Código de constructores propios del televisor
    //Código de métodos propios del televisor

    @Override
    public void subirVolumen() {
        //Sube el volumen del televisor
    }

    @Override
    public void bajarVolumen() {
        //Sube el volumen del televisor
    }
}
```

En este punto cabe preguntarse ¿Para qué pueden servir las interfaces? ¿De qué sirve un mecanismo que nos obliga a implementar en nuestras clases determinados métodos? La respuesta la veremos al explorar el polimorfismo, en el cual las interfaces juegan un papel esencial<sup>1</sup>.

---

<sup>1</sup>Los ejemplos anteriores se pueden ver acá: [https://github.com/leonjaramillo/udea\\_ruta2\\_ciclo2/tree/main/main/java/co/edu/udea/udea\\_ruta2\\_ciclo2/poo](https://github.com/leonjaramillo/udea_ruta2_ciclo2/tree/main/main/java/co/edu/udea/udea_ruta2_ciclo2/poo)