


# Java

## POO - Introducción

 Conceptos clave explicados:

### ¿Qué es la POO?

- Es un paradigma de programación donde **todo gira alrededor de los objetos**.
- Cada objeto **representa algo del mundo real o abstracto**, y tiene:
  - **Estado** (atributos)
  - **Comportamiento** (métodos)
  - **Identidad** (es único)

Ejemplo: Una clase `Auto` puede tener como atributos `color`, `marca`, `velocidad`, y métodos como `acelerar()` y `frenar()`.

### Objeto vs Clase

- **Clase**: es el **molde**, la **plantilla**. Define cómo serán los objetos que creemos.
- **Objeto**: es la **instancia concreta** de una clase. Tiene valores reales.

Ejemplo:

```
class Auto {  
  
    String color;  
  
    void encender() { ... }  
  
}
```

```
Auto miAuto = new Auto();
```

```
miAuto.color = "Rojo";
```

### Mensajes

- En la POO no se llama a funciones, sino que se **envían mensajes** a los objetos.
- Cuando hacés `objeto.metodo()`, le estás diciendo al objeto que haga algo.
- **Diferencia importante**:
  - `f(o, x)` → estilo procedural: “ejecutá f con o y x”.
  - `o.f(x)` → estilo orientado a objetos: “objeto o, hacé f con x”.

### Consultas vs Comandos

- **Consulta**: pregunta por el estado del objeto, pero **no lo modifica**.

- Ej: `getNombre()`, `obtenerEdad()`
- **Comando:** cambia el estado del objeto.
  - Ej: `cambiarContraseña()`, `agregarElemento()`

Esto es importante porque los **objetos deben cuidar su estado interno**.

## 🧱 Clases como molde

- Al definir una clase, **no estás creando un objeto**, sino una definición.
- Cuando hacés `new`, recién ahí **creás una instancia concreta**.

```
Persona p = new Persona("Juan");
```

Ahora `p` es un objeto con un estado real.

## 👤 Identidad del objeto

- Aunque dos objetos tengan los mismos valores, son distintos.

```
Persona a = new Persona("Ana");
Persona b = new Persona("Ana");
System.out.println(a == b); // false
```

# Encapsulamiento

## 🧠 ¿Qué es el encapsulamiento?

Encapsular significa **agrupar datos y operaciones relacionadas en una misma unidad**, que en la POO es una clase. Es uno de los pilares fundamentales de la programación orientada a objetos.

💡 El documento arranca con una advertencia muy válida:

“Encapsulamiento NO es simplemente hacer atributos privados”

Es mucho más que eso. Es una forma de **organizar el código, proteger el estado interno** de los objetos y **asignar responsabilidades claras**.

## 🎯 Objetivo del encapsulamiento

- **Especialización:** cada objeto sabe cómo manejar sus propios datos.
- **Compleitud:** todo lo que el objeto necesita para cumplir su responsabilidad está dentro del mismo objeto.

📦 Una clase bien encapsulada es como una caja negra: sabés qué hace, pero no cómo lo hace por dentro.

## 🔒 Interfaz pública vs implementación interna

- **Interfaz pública:** conjunto de métodos que el objeto expone al exterior (lo que otros objetos pueden usar).
- **Implementación interna:** los atributos y detalles que **no se deben conocer desde afuera**.

Esto permite cambiar la implementación sin romper a los que usan la clase.

## ⚠ Encapsulamiento ≠ Ocultamiento de información (pero se relacionan)

- **Encapsulamiento:** organizar datos + métodos en una clase.
- **Ocultamiento de información:** evitar que otros módulos accedan directamente a los datos internos.

Ambos trabajan juntos: encapsulás para definir responsabilidades, ocultás para proteger la lógica interna.

## 🔧 Ejemplos prácticos del archivo

```
// MAL: no encapsula nada
public class NoEncapsulation {
    public ArrayList widths = new ArrayList();
}

// MEJOR: encapsula, pero no oculta bien
private ArrayList widths = new ArrayList();
public ArrayList getWidths() { return widths; }

// AÚN MEJOR: oculta detalles internos
private ArrayList widths = new ArrayList();
public List getWidths() { return widths; }
```

El último ejemplo devuelve `List` en lugar de `ArrayList`, para **ocultar la implementación concreta**.

## 🔧 Buenas prácticas

1. **No exponer atributos directamente** (`public int edad` → ❌).
2. Usar `getters` y `setters`, pero solo si son realmente necesarios.
3. No mostrar cómo están implementadas las cosas (ocultá la estructura interna).
4. Aislá los cambios dentro de la clase siempre que sea posible.

## 🧠 Frase clave: “Tell, don’t ask”

En vez de preguntarle a un objeto por su estado para decidir algo...

```
if (monitor.getTemperatura() > 100) {
    monitor.sonarAlarmas();
}
```

...delegá la responsabilidad al objeto:

```
monitor.verificarSobrecalentamiento();
```

Esto respeta el encapsulamiento porque **le das la responsabilidad al objeto**, y vos no te metés en sus detalles.

## Conclusión

El encapsulamiento **no es esconder por esconder**, sino diseñar las clases de forma tal que **las responsabilidades estén claramente definidas y controladas desde adentro**. Y que los objetos **no tengan que saber detalles de otros para poder interactuar con ellos**.

# Herencia

## ¿Qué es la herencia?

La herencia permite **definir nuevas clases a partir de otras existentes**.

Es una forma de **organizar clases jerárquicamente**, reutilizando y extendiendo código.

 Ejemplo típico:

`Estudiante es un Persona`  
`class Estudiante extends Persona`

## Relación “es-un”

- **Generalización:** `Estudiante` es una `Persona`. (`Estudiante` hereda de `Persona`)
- **Especialización:** `Persona` es una **superclase** más general que se especializa con `Estudiante`.

## ¿Para qué sirve la herencia?

- **Reutilizar código:** los métodos y atributos de la superclase se heredan automáticamente.
- **Organizar clases por jerarquía lógica.**
- **Habilitar polimorfismo** (tema del próximo archivo).

## Herencia clásica

```
class Persona {  
    void saludar() {  
        System.out.println("Hola");  
    }  
}  
  
class Estudiante extends Persona {  
    void darPresente() {  
        System.out.println("¡Presente!");  
    }  
}
```

```
}  
Persona p = new Estudiante(); // válido (polimorfismo)  
p.saludar(); // "Hola"
```

## ⚠️ Cuándo usar (y cuándo NO usar) herencia

### ✅ Usar herencia cuando:

- Hay una **relación natural “es-un”**.
- Las clases **comparten comportamiento real**.

### ❌ Evitar herencia cuando:

- Solo querés **reutilizar código**.  
En ese caso, mejor usar **composición** (“tiene-un”).

## 💡 Composición vs Herencia

Mal ejemplo con herencia:

```
class Persona extends Boca { } // “una persona es una boca”? NO
```

Mejor con composición:

```
class Persona {  
    private Boca boca;  
    void comer(Comida c) {  
        boca.comer(c);  
    }  
}
```

## 🧰 Sobreescritura de métodos (@Override)

Podés redefinir métodos heredados:

```
class Camion extends Transporte {  
    @Override  
    double calcularDesgaste() {  
        return super.calcularDesgaste() + carga / maxCarga;  
    }  
}
```

- Usás `super.metodo()` si querés **reutilizar parte** del comportamiento heredado.
- Sin `super`, reemplazás completamente el método.

## 🧩 Clases abstractas

- **Clase abstracta:** no se puede instanciar. Sirve como base para subclases.
- Puede tener **métodos abstractos** (sin implementación) que las subclases deben implementar.

```
abstract class Transporte {
    abstract void cargarPaquete(Paquete p);
}
```

## Constructores y herencia

- Si la superclase tiene constructores **con parámetros**, la subclase debe invocarlos explícitamente con `super(...)`.

```
class Persona {
    Persona(String nombre) { ... }
}
class Estudiante extends Persona {
    Estudiante(String nombre) {
        super(nombre);
    }
}
```

## Smell: uso excesivo de **if**

Si tenés muchas condiciones en una clase general como:

```
if (tipo.equals(\"bicicleta\")) ...
else if (tipo.equals(\"automóvil\")) ...
```

... probablemente estés necesitando usar **herencia** para que cada tipo tenga su propio comportamiento.

# Polimorfismo

## ¿Qué es el polimorfismo?

“Misma interfaz, diferente implementación.”

Significa que un **mismo mensaje (método)** puede producir **diferentes comportamientos** dependiendo del tipo real del objeto que lo recibe.

## Requisitos previos

- Se basa en la **herencia**: una clase **Hija** extiende a una clase **Padre**.

- El objeto puede ser tratado como una instancia de su **superclase**, pero se comporta como su **tipo real**.

## **Asignaciones polimórficas**

```
Persona persona1 = new Persona();
```

```
Persona persona2 = new Estudiante(); // polimórfico
```

Aunque `persona2` es una `Persona`, **internamente** es un `Estudiante`.

Pero solo podés usar métodos definidos en `Persona`, no en `Estudiante`, a menos que hagas un cast.

```
persona2.saludar(); // OK
```

```
persona2.darPresente(); // ❌ No compila
```

## **Reglas de polimorfismo**

- El tipo a la derecha (`new Estudiante()`) debe ser igual o **más específico** que el tipo a la izquierda (`Persona`).
- No funciona al revés:

```
Estudiante estudiante = new Persona(); // ❌ No compila
```

## **Dynamic binding (ligadura dinámica)**

Cuando llamás a un método sobre un objeto polimórfico, **se ejecuta la versión correspondiente al tipo real**, no al tipo declarado.

```
class Persona {  
    void saludar() { System.out.println("Hola"); }  
}  
  
class Estudiante extends Persona {  
    void saludar() { System.out.println("¡Presente!"); }  
}  
  
Persona p = new Estudiante();  
  
p.saludar(); // ¡Presente!
```

Aunque `p` es de tipo `Persona`, se comporta como un `Estudiante`. Esto se llama **binding dinámico**.

## Entidades polimórficas

Métodos que reciben parámetros del tipo general (**superclase**) y pueden trabajar con **subtipos**:

```
class Medico {  
  
    void curar(Persona persona) {  
  
        // puede recibir Persona, Estudiante, Medico...  
  
    }  
  
}
```

No importa el tipo exacto que reciba, mientras sea **Persona** o una subclase. Esto da **flexibilidad** al diseño.

## Estructuras polimórficas

Podés usar colecciones para guardar objetos de distintas subclases si comparten una superclase:

```
List<Persona> asistentes = new ArrayList<>();  
  
asistentes.add(new Estudiante());  
  
asistentes.add(new Medico());
```

Podés recorrer la lista y pedirles que “saluden”, aunque cada uno lo hará a su manera:

```
for (Persona p : asistentes) {  
  
    p.saludar(); // cada uno usa su propia implementación  
  
}
```

## ¡Cuidado! Polimorfismo no es conversión

```
Estudiante e = new Estudiante();  
  
Persona p = e;  
  
System.out.println(p instanceof Estudiante); // true
```

El **objeto no cambia**, solo la **referencia** cambia de tipo.

**No convierte un objeto en otro**, solo lo trata como su tipo base.



## Diferencias de polimorfismo

### Polimorfismo por subtipos

- Clases relacionadas por herencia.
- Ej: **Forma**, **Círculo**, **Cuadrado**, todos con `.dibujar()`.

### Polimorfismo por utilidad

- Clases no relacionadas pero con un método en común.
- Ej: todas las clases implementan `toString()`, aunque no tengan nada que ver entre sí.

## Las gotas de UML que hacen falta

### ¿Qué es UML?

UML (**Unified Modeling Language**) es un **lenguaje de modelado visual**.

No es un lenguaje de programación, sino una **forma de representar cómo está estructurado tu sistema** (clases, relaciones, comportamiento).

### ¿Para qué sirve?

- Para **planificar** antes de programar.
- Para **comunicar** tu diseño a otras personas (docentes, compañeros, evaluadores).  
Para **documentar** cómo está organizado tu código.

### Diagrama de clases – Lo esencial

Un **diagrama de clases** muestra:

1. **Clases** con:
  - **Nombre**
  - Atributos**
  - **Métodos**
2. **Relaciones** entre clases.

### Notación básica de clase

-----

|      **Persona**                      |   ← Nombre de la clase

-----

| - nombre: String                |   ← Atributos (visibilidad + tipo)

```
| - edad: int |
```

```
-----
```

```
| + getNombre():String| ← Métodos (visibilidad + tipo de retorno)
```

```
| + saludar():void |
```

```
-----
```

## Visibilidad

Símbol o	Significad o
-------------	-----------------

+	Público
---	---------

-	Privado
---	---------

#	Protegido
---	-----------

~	Paquete
---	---------

## Tipos de relaciones

Relación	Símbolo UML	Significado
Asociación	simple línea	Una clase <b>conoce</b> a otra
Agregación	◇ (rombo vacío)	“tiene-un”, relación débil
Composición	◆ (rombo lleno)	“tiene-un”, relación fuerte

Herencia      ▲ (flecha hueca)      “es-un”, especialización/generalización

Dependencia      línea discontinua con      Uso temporal o puntual  
a      flecha

## Asociación, Agregación, Composición

### Asociación

Auto ----- Persona

Significa que **Persona** conoce a **Auto**, puede tener una referencia.

### Agregación

Equipo ◇----- Jugador

Un equipo **tiene jugadores**, pero los jugadores pueden existir fuera del equipo.

### Composición

Casa ◆----- Habitación

Las habitaciones **no existen sin la casa**. Si se destruye la casa, desaparecen.

## Herencia

Persona ▲← Estudiante

Indica que **Estudiante** hereda de **Persona**. Es una relación **jerárquica**.

## Notación de cardinalidad

Define **cuántas instancias** de una clase se relacionan con otra.

Ejemplos:

- 1 : una sola
- 0..1 : cero o una
- \* : muchas
- 1..\* : al menos una

Curso 1 ↔ \* Alumno

Un curso tiene muchos alumnos, y un alumno pertenece a un curso.

## ✓ Reglas de diseño sugeridas en el archivo

- No pongas TODO en una clase. Distribuí responsabilidades.
- Usá **nombres significativos** para clases, atributos y métodos.
- Mostrá sólo lo necesario en el diagrama (lo esencial).
- No pongas código en el diagrama, solo la **firma** de métodos.

## 🔧 ¿Cómo armar uno?

1. **Identificá entidades clave** de tu problema (clases).
2. Pensá: ¿qué atributos y métodos tiene cada una?
3. Definí relaciones entre ellas: ¿hay herencia? ¿quién contiene a quién?
4. Dibujalo con las notaciones de UML.

# Entrada y Salida en Java

## 🧠 ¿Qué es E/S?

La **entrada (input)** y la **salida (output)** son las formas en que un programa **interactúa con el exterior**:

- Entrada: lo que recibe (teclado, archivos, red, etc.).
- Salida: lo que genera (consola, archivos, etc.).

En Java, todo esto se maneja con objetos llamados **flujos** (streams).

## 🔄 Flujos de datos (Streams)

Un **stream** es una secuencia de datos que va en una dirección:

Dirección	Ejemplo en Java
-----------	-----------------

Entrada	<code>System.in</code> , <code>Scanner</code>
---------	---

Salida	<code>System.out</code> , <code>PrintWriter</code>
--------	---

Los flujos pueden ser:

- **De bytes** → para datos binarios (`InputStream`, `OutputStream`)
- **De caracteres** → para texto (`Reader`, `Writer`)



## Clases comunes para Entrada

### ✓ **Scanner** (la más usada para consola)

```
Scanner sc = new Scanner(System.in);
```

```
int edad = sc.nextInt();
```

```
String nombre = sc.nextLine();
```

- Simple, intuitiva.
- Ideal para ejercicios en consola.

### ✓ **BufferedReader** (más eficiente, usa buffers)

```
BufferedReader br = new BufferedReader(new  
InputStreamReader(System.in));
```

```
String linea = br.readLine();
```

- Lee líneas completas de texto.
- Necesita manejo de excepciones (lanza `IOException`).



## Clases comunes para Salida

### ✓ **System.out**

```
System.out.println("Hola mundo");
```

- Salida directa a consola.
- Muy simple, ideal para debug.

### ✓ **PrintWriter / FileWriter**

Para escribir en archivos:

```
PrintWriter pw = new PrintWriter(new FileWriter("salida.txt"));
```

```
pw.println("Hola archivo");
```

```
pw.close();
```

- Podés usar `BufferedWriter` para mayor eficiencia.
- Siempre cerrá el archivo al terminar (`close()`).



## Leer archivos línea por línea

```
BufferedReader br = new BufferedReader(new
FileReader("archivo.txt"));

String linea;

while ((linea = br.readLine()) != null) {

    System.out.println(linea);

}

br.close();
```

### ¿Qué es un buffer?

Un **buffer** es una memoria intermedia que **acelera el acceso a archivos o teclado**. Por eso muchas clases en Java son `BufferedReader`, `BufferedWriter`, etc.

### Manejo de errores

Trabajar con archivos puede fallar → **se debe usar excepciones**:

```
try {

    BufferedReader br = new BufferedReader(new
FileReader("datos.txt"));

    // ...

} catch (IOException e) {

    System.out.println("No se pudo leer el archivo");

}
```

## Colecciones en Java (Java Collections Framework)

### ¿Qué son?

Son estructuras de datos listas para usar que permiten **almacenar, organizar y procesar grupos de objetos** (listas, conjuntos, mapas, etc.).

Están en el paquete `java.util` y forman un sistema bien pensado de **interfaces e implementaciones**.

## Interfaces principales

Interfaz	¿Qué representa?	Permite duplicados	Mantiene orden
List	Lista ordenada	✓ Sí	✓ Sí
Set	Conjunto (sin duplicados)	✗ No	✗/✓ Según tipo
Map	Diccionario clave→valor	Claves: ✗ / Valores: ✓	🔄 Según tipo

### 1. List

#### ¿Qué es?

Una **secuencia ordenada** de elementos. Puede contener duplicados.

#### Implementaciones comunes:

- **ArrayList**: más rápido para acceder por índice.
- **LinkedList**: más rápido para insertar/borrar en el medio.

#### Métodos útiles:

```
List<String> nombres = new ArrayList<>();
nombres.add("Ana");
nombres.add("Juan");
nombres.get(0); // "Ana"
nombres.remove(1);
nombres.contains("Ana"); // true
```

### 2. Set

#### ¿Qué es?

Un conjunto de elementos **únicos** (no duplicados). Muy útil para filtrar.

#### Implementaciones comunes:

- **HashSet**: sin orden.
- **LinkedHashSet**: mantiene orden de inserción.
- **TreeSet**: mantiene orden **natural** o por **Comparator**.

#### Ejemplo:

```
Set<Integer> numeros = new HashSet<>();
```

```
numeros.add(5);
numeros.add(5); // Ignorado
System.out.println(numeros.size()); // 1
```

## 3. Map

### ¿Qué es?

Una colección de **pares clave→valor**. Como una agenda telefónica.

### Implementaciones comunes:

- `HashMap`: sin orden.
- `TreeMap`: ordena por clave.
- `LinkedHashMap`: mantiene orden de inserción.

### Métodos útiles:

```
Map<String, Integer> edades = new HashMap<>();
edades.put("Ana", 30);
edades.get("Ana"); // 30
edades.containsKey("Juan"); // false
edades.remove("Ana");
```

## Métodos comunes para todas las colecciones

- `.size()`: cantidad de elementos
- `.isEmpty()`: está vacía
- `.clear()`: borra todo
- `.contains(elem)`: lo contiene

## Iterar una colección

```
for (String nombre : nombres) {
    System.out.println(nombre);
}
```

Con Map:

```
for (Map.Entry<String, Integer> entry : edades.entrySet()) {
    System.out.println(entry.getKey() + " → " + entry.getValue());
}
```

 Cuándo usar cada una



Necesito...	Uso recomendado
Elementos ordenados y duplicables	List
Elementos únicos, sin importar orden	HashSet
Elementos únicos, ordenados	TreeSet
Asociar claves con valores	HashMap
Mantener orden de inserción en un Map	LinkedHashMap

## Excepciones en Java

### ¿Qué es una excepción?

Una **excepción** es un evento que ocurre durante la ejecución del programa y **interrumpe el flujo normal del código**.

Java te da herramientas para manejar esos errores **de forma controlada**.

Ejemplo clásico:

```
int x = 10 / 0; // ArithmeticException
```

## Tipos de excepciones

### Chequeadas (checked)

- **Obligatorias de manejar** (en `try-catch` o con `throws`)
- Son excepciones previsibles (ej: acceso a archivos, conexión a red)

```
FileReader fr = new FileReader("archivo.txt"); // IOException
```

### No chequeadas (unchecked)

- **Errores de lógica**, como `NullPointerException`, `IndexOutOfBoundsException`
- No estás obligado a manejarlas (pero podés hacerlo)

## Estructura básica de manejo

```
try {
```

```

        // Código que puede fallar
    } catch (TipoDeExcepcion e) {
        // Qué hacer si ocurre
    } finally {
        // Código que SIEMPRE se ejecuta (opcional)
    }

```

## Ejemplo práctico

```

Scanner sc = new Scanner(System.in);
try {
    int n = sc.nextInt();
    int resultado = 10 / n;
} catch (ArithmeticException e) {
    System.out.println("No se puede dividir por cero");
} catch (InputMismatchException e) {
    System.out.println("No ingresaste un número válido");
} finally {
    System.out.println("Gracias por usar el programa");
}

```

## Lanzar excepciones manualmente

Si vos querés forzar un error:

```

if (edad < 0) {
    throw new IllegalArgumentException("La edad no puede ser negativa");
}

```

## Declarar que un método puede lanzar excepciones

```

public void leerArchivo() throws IOException {
    BufferedReader br = new BufferedReader(new
    FileReader("datos.txt"));
}

```

## Excepciones comunes

**Excepción**

**Cuándo ocurre**

<code>ArithmeticException</code>	División por cero
<code>NullPointerException</code>	Acceso a métodos/atributos de <code>null</code>
<code>ArrayIndexOutOfBoundsException</code>	Índice inválido en arreglo
<code>InputMismatchException</code>	Entrada incorrecta con <code>Scanner</code>
<code>IOException</code>	Problemas al leer/escribir archivos
<code>FileNotFoundException</code>	Archivo no encontrado

## Buenas prácticas

- No atrapes `Exception` genérico a menos que tengas una buena razón.
- Usá **varios `catch` específicos** si sabés qué puede fallar.
- **No ignores las excepciones** (nunca hagas `catch (Exception e) {}` vacío).
- Cerrá recursos en el bloque `finally` o usá `try-with-resources`.

## Patrón de diseño: Composite

### ¿Qué es?

`Composite` permite **tratar objetos individuales y grupos de objetos de la misma forma**.

Sirve para representar **estructuras jerárquicas (como árboles)**: por ejemplo, un archivo puede ser una carpeta o un documento, y ambos deben responder a las mismas operaciones.

💬 “Componer objetos en estructuras de árbol para representar jerarquías parte-todo.”

### Estructura

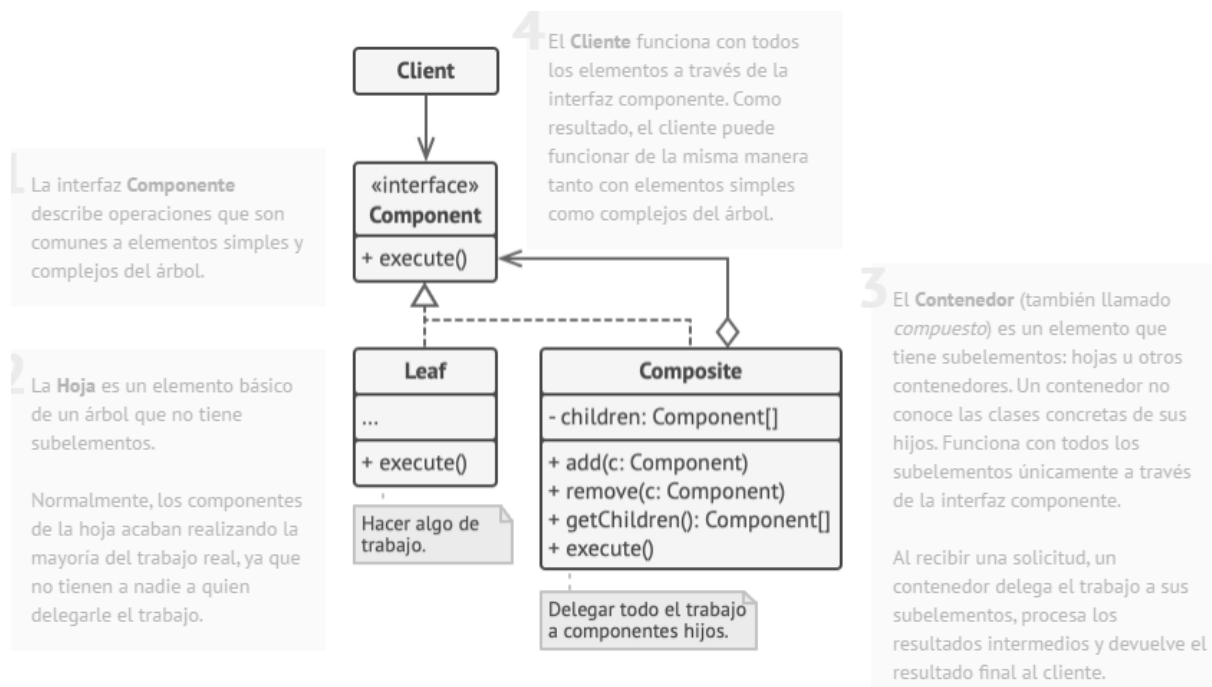
plaintext

CopiarEditar

`Component`

├─ Leaf

└─ Composite (que también contiene `Component`)



## Ejemplo clásico: sistema de archivos

```

interface Componente {
    void mostrar();
}

class Archivo implements Componente {
    private String nombre;
    public Archivo(String nombre) { this.nombre = nombre; }
    public void mostrar() {
        System.out.println("Archivo: " + nombre);
    }
}

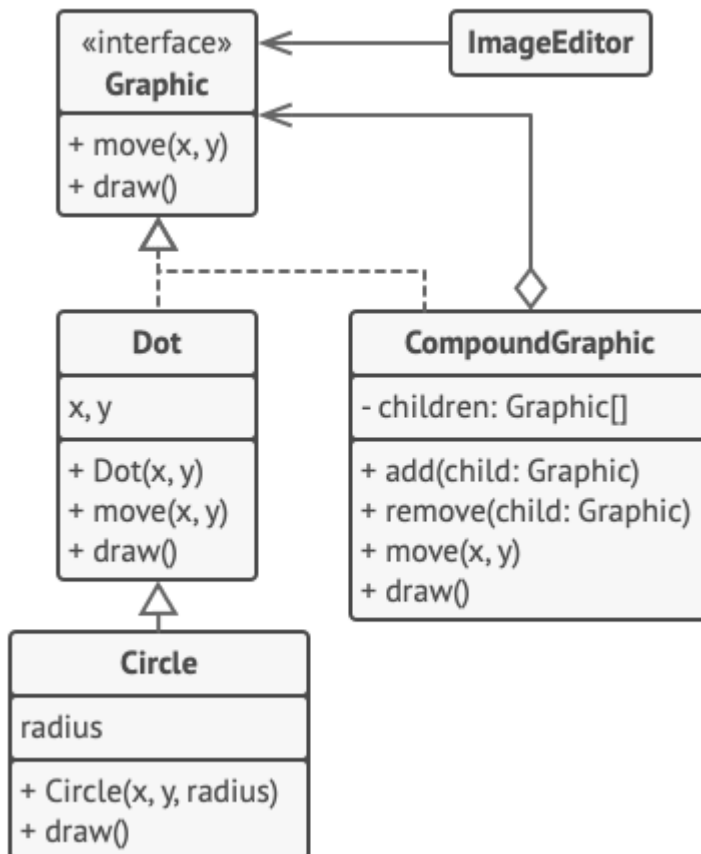
class Carpeta implements Componente {
    private String nombre;
    private List<Componente> hijos = new ArrayList<>();
    public Carpeta(String nombre) { this.nombre = nombre; }
    public void agregar(Componente c) {
        hijos.add(c);
    }
    public void mostrar() {
        System.out.println("Carpeta: " + nombre);
        for (Componente c : hijos) {
            c.mostrar();
        }
    }
}

```

## Uso:

```
Carpeta raiz = new Carpeta("Documentos");
raiz.agregar(new Archivo("cv.pdf"));
Carpeta sub = new Carpeta("Fotos");
sub.agregar(new Archivo("verano.jpg"));
raiz.agregar(sub);

raiz.mostrar();
```



## 🧠 Clave del Composite

- **Composite** implementa la misma interfaz que sus hijos.
- Así, no importa si tenés un **Archivo** o una **Carpeta**: ambos responden a `.mostrar()`.

## ✅ ¿Cuándo usar Composite?

- Cuando necesitás representar jerarquías (organización de empresa, carpetas, UI con widgets).
- Cuando querés que **objetos y colecciones se comporten igual**.

# Patrón de diseño: State

## ¿Qué es?

**State** permite que un objeto **cambie su comportamiento dependiendo de su estado interno, sin usar `if` o `switch` gigantes**.

💬 “Permite a un objeto alterar su comportamiento cuando cambia su estado.”

## Problema clásico:

```
if (estado == "encendido") {  
    // hacer A  
} else if (estado == "apagado") {  
    // hacer B  
} else if (estado == "en pausa") {  
    // hacer C  
}
```

Esto **viola el principio de abierto/cerrado** y es difícil de mantener.

## Estructura del patrón

Context

tiene una referencia a  $\rightarrow$  Estado (interfaz)

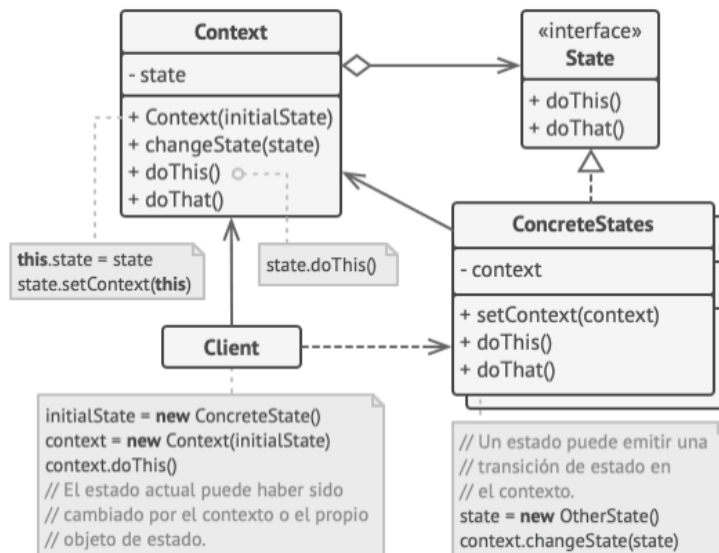
↑  
ConcretoEstadoA, ConcretoEstadoB...

La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasarle un nuevo objeto de estado.

La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.

3 Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.



4 Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

## Ejemplo práctico: Reproductor de música

```

interface Estado {
    void reproducir();
}

class Reproduciendo implements Estado {
    public void reproducir() {
        System.out.println("Ya está reproduciendo");
    }
}

class Pausado implements Estado {
    public void reproducir() {
        System.out.println("Reanudando...");
    }
}

class Reproductor {
    private Estado estado;
    public void setEstado(Estado estado) {
        this.estado = estado;
    }
    public void reproducir() {
        estado.reproducir();
    }
}
    
```

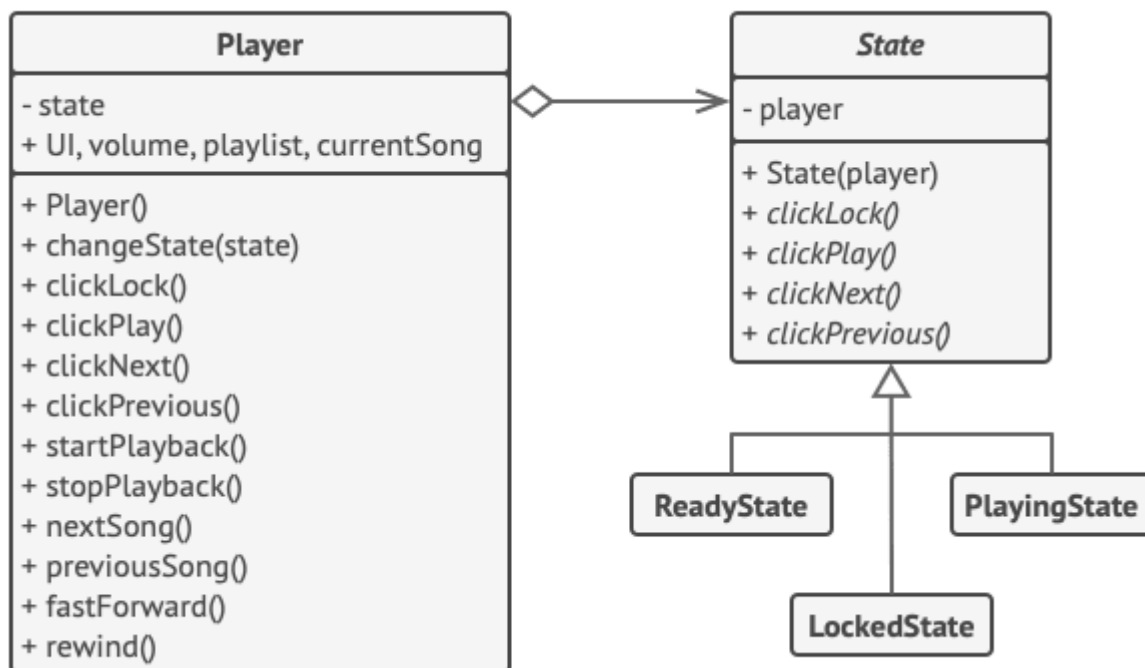
```
}
```

## Uso:

```
Reproductor mp3 = new Reproductor();
```

```
mp3.setEstado(new Pausado());  
mp3.reproducir(); // Reanudando...
```

```
mp3.setEstado(new Reproduciendo());  
mp3.reproducir(); // Ya está reproduciendo
```



## 🧠 Clave del patrón State

- Cada estado es **una clase diferente**.
- Se cambia de comportamiento **dinámicamente** cambiando el objeto que representa el estado.
- Evita condicionales y mejora la mantenibilidad.

## ✅ ¿Cuándo usar State?

- Cuando un objeto tiene **muchos comportamientos posibles según su estado**.
- Cuando esos comportamientos **cambian en tiempo de ejecución**.