**POO**

1. Clases: Las clases son la base de la POO. Representan un conjunto de objetos con propiedades y comportamientos similares. Una clase define la estructura y el comportamiento que los objetos creados a partir de ella tendrán.
2. Objetos: Los objetos son instancias de una clase. Cada objeto tiene su propio estado (valores de las propiedades) y comportamiento (métodos asociados). Puedes crear múltiples objetos a partir de una sola clase.
3. Propiedades: Las propiedades son las características o datos asociados a un objeto. Pueden ser variables que almacenan valores específicos para cada objeto. Por ejemplo, una clase "Persona" podría tener propiedades como "nombre" y "edad".
4. Métodos: Los métodos son funciones asociadas a una clase u objeto. Representan el comportamiento de un objeto y se utilizan para realizar operaciones o manipular los datos de un objeto. Por ejemplo, una clase "Persona" podría tener un método "saludar()" que muestra un saludo en la consola.
5. Encapsulación: La encapsulación es un concepto que se refiere a la agrupación de datos y métodos relacionados en una clase. Permite ocultar los detalles internos de una clase y proporcionar una interfaz pública para interactuar con los objetos. Esto ayuda a proteger los datos y a organizar el código de manera más estructurada.
6. Herencia: La herencia es un mecanismo que permite que una clase herede propiedades y métodos de otra clase. La clase que hereda se llama "clase derivada" o "subclase", y la clase de la cual se hereda se llama "clase base" o "superclase". La herencia permite la reutilización de código y la creación de jerarquías de clases.
7. Polimorfismo: El polimorfismo es la capacidad de objetos de diferentes clases para responder al mismo mensaje o invocar al mismo método. Esto significa que puedes tener diferentes clases con el mismo nombre de método, pero con implementaciones diferentes. El polimorfismo permite escribir código más genérico y flexible.

**Compiler 1**

****

En el ejemplo anterior, se define la clase **Persona** con un constructor que recibe el nombre y la edad como parámetros y los asigna a las propiedades **nombre** y **edad** del objeto. La clase también tiene un método llamado **saludar()** que muestra un mensaje de saludo utilizando las propiedades del objeto.

Luego, se crea un objeto llamado **persona1** utilizando la clase **Persona** y se le asigna el nombre "Juan" y la edad 25. Finalmente, se accede a las propiedades y al método del objeto utilizando la notación de punto (**objeto.propiedad** y **objeto.metodo()**).

Este es solo un ejemplo básico para ilustrar la creación de clases y objetos en JavaScript. En la programación orientada a objetos, las clases se utilizan para definir un tipo de objeto con sus propiedades y comportamientos, y los objetos se crean a partir de esas clases para representar entidades específicas.

**COMPILER 2**

****

En este ejemplo, tenemos una clase base llamada **Animal** con un método **hablar()**, que muestra un mensaje genérico de "El animal hace ruidos". Luego, tenemos dos clases derivadas, **Perro** y **Gato**, que heredan de la clase base **Animal** y sobrescriben el método **hablar()** con mensajes específicos para cada uno de ellos.

El polimorfismo se muestra al crear instancias de las clases **Perro** y **Gato** y llamar al método **hablar()**. A pesar de que el método **hablar()** está definido en la clase base **Animal**, cuando se llama desde las instancias de las clases derivadas **Perro** y **Gato**, se ejecuta la implementación específica de cada una de esas clases.

La encapsulación se logra mediante el uso de la palabra clave **this** dentro de las clases, que permite acceder y modificar las propiedades y métodos de cada instancia de objeto.

**EJERCICIO 2**

****

**EJERCICIO 3**

****

**EJERCICIO 4**

****

**SOLID:**

* Decir Solid ingles y español
* **Ventajas:** 
  + **Cohesión y Acoplamiento**

Los conceptos de cohesión y acoplamiento merecen un artículo a parte, pero a grandes rasgos lo que buscamos de un buen código es que sus clases puedan trabajar de forma independiente y que el cambio de uno afecte lo menos posible al resto.

Obviamente cuando dos clases se relacionan entre sí para trabajar juntas (y esto tiene que ocurrir sí o sí), va a existir un acoplamiento entre ellas.

Pero existen distintos niveles de acoplamiento, y gracias a algunos de los Principios SOLID, podemos relajar esas dependencias y hacerlas mucho más flexibles a cambios.

* + **Simplifican la creación de tests**

Todo esto está muy relacionado con los puntos anteriores: si tienes tu código desacoplado y una buena arquitectura, los tests van a ser mucho más sencillos.

En un vídeo anterior ya comentaba los 7 errores que solemos cometer al escribir tests, y mucho del tema va por aquí.Al final piensa que todo es como una cadena: si aplicas bien los principios, organizas mejor tu código. Esto te permite definir una arquitectura que hará que los tests sean más sencillos.

Podría decirse por tanto que los Principios SOLID son parte de la base de un código de calidad.

* **Unos principios no pueden existir sin los otros**

Una pregunta muy recurrente en mi formación de Architect Coders cuando vemos estos Principios es que parece que al querer aplicar uno de ellos, hay otro que necesitan aplicar inevitablemente.

¡Claro! Ahí está la gracia. Aún no hemos hablado de los Principios a fondo, pero por poner un ejemplo:

Imagina que tienes una clase A que tiene un acoplamiento muy fuerte con una clase B, de tal forma que cada vez que cambia B inevitablemente tiene que cambiar A.

Esto es muy posible que esté incumpliendo el Principio de Responsabilidad Única. La forma de cumplirlo sería haciendo que cuando B cambie, A no lo haga. Y para esto, la solución puede ser aplicar el Principio de Inversión de Dependencias.

¡No pasa nada! Esto es muy normal y te pasará casi siempre.

Un mismo problema se puede resolver desde dos perspectivas distintas en función de en qué Principio nos enfoquemos cuando lo resolvamos. Pero el resultado será el mismo.

* **Al cumplir un Principio puede que estés incumpliendo otro**

Esto es lo más difícil de aceptar: muchas veces es imposible cumplir todos los Principios a la vez

Porque al aplicar un Principio, se puede estar dando la espalda a otro.

Pero no te obsesiones con esto: al final lo importante es entender la potencia de cada Principio.

Y ese conocimiento unido a la experiencia te irá diciendo poco a poco cuáles son las mejores decisiones a tomar.

**¿Qué es el Principio de Responsabilidad Única?**

Si lo has visto por ahí, seguramente hayas una frase similar a esta:

*El Principio de Responsabilidad Única nos dice que un módulo tiene una única razón para cambiar*

En estos artículos verás que uso de forma indiferente las palabras módulo, entidad o clase. En realidad, cuando hablamos de lenguajes orientados a objetos, esto siempre se refiere a una clase.

A mí esta definición no me gusta mucho, porque lo de “una única razón para cambiar” me suena muy etéreo, y es difícil bajarlo a tierra.

Prefiero decir que el Principio de Responsabilidad Única se cumple cuando nuestra clase solo hace una cosa.

Tampoco es fácil definir qué es “una cosa”, pero ya tenemos herramientas más sencillas para detectarlo.

Por ejemplo, si cuando tienes que explicar el funcionamiento de una clase, dices que “esta clase hace esta cosa **Y** esta otra”, entonces sospecha.

Pero te lo voy a poner más fácil, te voy a dar unos truquillos para mejorar la detección:

**¿Cómo detectar si estamos violando el Principio de Responsabilidad Única?**

La respuesta a esta pregunta es bastante subjetiva. Sin necesidad de obsesionarnos con ello, podemos detectar situaciones en las que una clase podría dividirse en varias:

* **En una misma clase están involucradas dos capas de la arquitectura**: esta puede ser difícil de ver sin experiencia previa. En toda arquitectura, por simple que sea, debería haber una capa de presentación, una de lógica de negocio y otra de persistencia. Si mezclamos responsabilidades de dos capas en una misma clase, será un buen indicio.
* **El número de métodos públicos**: Si una clase hace muchas cosas, lo más probable es que tenga muchos métodos públicos, y que tengan poco que ver entre ellos. Detecta cómo puedes agruparlos para separarlos en distintas clases. Algunos de los puntos siguientes te pueden ayudar.
* **Los métodos que usan cada uno de los campos de esa clase**: si tenemos dos campos, y uno de ellos se usa en unos cuantos métodos y otro en otros cuantos, esto puede estar indicando que cada campo con sus correspondientes métodos podrían formar una clase independiente. Normalmente esto estará más difuso y habrá métodos en común, porque seguramente esas dos nuevas clases tendrán que interactuar entre ellas.
* **Por el número de imports**: Si necesitamos importar demasiadas clases para hacer nuestro trabajo, es posible que estemos haciendo trabajo de más. También ayuda fijarse a qué paquetes pertenecen esos imports. Si vemos que se agrupan con facilidad, puede que nos esté avisando de que estamos haciendo cosas muy diferentes.
* **Nos cuesta testear la clase**: si no somos capaces de escribir tests unitarios sobre ella, o no conseguimos el grado de granularidad que nos gustaría, es momento de plantearse dividir la clase en dos.
* **Cada vez que escribes una nueva funcionalidad, esa clase se ve afectada**: si una clase se modifica a menudo, es porque está involucrada en demasiadas cosas.
* **Por el número de líneas**: a veces es tan sencillo como eso. Si una clase es demasiado grande, intenta dividirla en clases más manejables.

En general no hay reglas de oro para estar 100% seguros. La práctica te irá haciendo ver cuándo es recomendable que cierto código se mueva a otra clase, pero estos indicios te ayudarán a detectar algunos casos donde tengas dudas.

**COMPILER 1 mal:**

****

En este ejemplo, la clase **Pedido** tiene varias responsabilidades diferentes, violando el principio de responsabilidad única. La clase se encarga de calcular el total del pedido, generar una factura y enviar un correo electrónico de confirmación. Esto hace que la clase sea difícil de mantener y modificar, ya que cualquier cambio en una de estas responsabilidades puede afectar a las demás.

**Compiler 2 bien:**

****

En este nuevo ejemplo, hemos dividido las responsabilidades en diferentes clases. La clase **Pedido** ahora solo se encarga de calcular el total del pedido. La generación de la factura se ha movido a la clase **Facturador**, y el envío de correos electrónicos se ha encapsulado en la clase **EmailService**. Cada clase tiene una única responsabilidad y puede ser modificada de forma independiente sin afectar a las demás.

Este enfoque facilita la comprensión, el mantenimiento y la extensión del código, ya que cada clase se centra en una tarea específica y no tiene que preocuparse por las demás responsabilidades.

**Principio Open/Closed**

El principio Open/Closed fue nombrado por primera vez por **Bertrand Mayer**, un programador francés, quien lo incluyó en su libro [Object Oriented Software Construction](http://devexperto.com/object-oriented-software-construction) allá por 1988.

Este principio nos dice que **una entidad de software debería estar abierta a extensión pero cerrada a modificación**

¿Qué quiere decir esto? Que tenemos que ser capaces de extender el comportamiento de nuestras clases sin necesidad de modificar su código.

Esto nos ayuda a seguir añadiendo funcionalidad con la seguridad de que no afectará al código existente. Nuevas funcionalidades implicarán añadir nuevas clases y métodos, pero en general no debería suponer modificar lo que ya ha sido escrito.

La forma de llegar a ello está muy relacionada con el punto anterior. Si las clases sólo tienen una responsabilidad, podremos añadir nuevas características que no les afectarán. **Esto no quiere decir que cumpliendo el primer principio se cumpla automáticamente el segundo**, ni viceversa. Luego verás un caso claro en el ejemplo.

El principio Open/Closed **se suele resolver utilizando**[**polimorfismo**](https://es.wikipedia.org/wiki/Polimorfismo_(inform%C3%A1tica)). En vez de obligar a la clase principal a saber cómo realizar una operación, delega esta a los objetos que utiliza, de tal forma que no necesita saber explícitamente cómo llevarla a cabo. Estos objetos tendrán una interfaz común que implementarán de forma específica según sus requerimientos.

**¿Cómo detectar que estamos violando el principio Open/Closed?**

Una de las formas más sencillas para detectarlo es darnos cuenta de qué clases modificamos más a menudo. Si cada vez que hay un nuevo requisito o una modificación de los existentes, las mismas clases se ven afectadas, podemos empezar a entender que estamos violando este principio.

Compiler 3:



En este ejemplo, se aplica el principio Open-Closed al permitir la extensión de las formas geométricas sin modificar el código existente. La clase base **FormaGeometrica** define un método abstracto **calcularArea** que debe ser implementado por las subclases. Las subclases **Rectangulo** y **Circulo** extienden la clase base y proporcionan su propia implementación del método **calcularArea**.

La función **calcularAreaTotal** puede aceptar un conjunto de formas geométricas y calcular su área total utilizando el polimorfismo. De esta manera, si se agregan nuevas formas geométricas en el futuro, no será necesario modificar la función **calcularAreaTotal**, ya que simplemente trabajará con las formas proporcionadas y utilizará el método **calcularArea** correspondiente de cada forma.

**Principio de Sustitución de Liskov**

El principio de sustitución de Liskov nos dice que **si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido**.

Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la padre.

Este principio viene a desmentir la idea preconcebida de que las clases son una forma directa de modelar la realidad, y que hay que tener cuidado con esa modelización.

La primera en hablar de él fue [Bárbara Liskov](https://en.wikipedia.org/wiki/Barbara_Liskov) (de ahí el nombre), una reconocida ingeniera de software americana.

**¿Cómo detectar que estamos violando el principio de sustitución de Liskov?**

Seguro que te has encontrado con esta situación muchas veces: creas una clase que extiende de otra, pero de repente uno de los métodos te sobra, y no sabes que hacer con él.

Las opciones más rápidas son bien dejarlo vacío, bien lanzar una excepción cuando se use, asegurándote de que nadie llama incorrectamente a un método que no se puede utilizar.

**Si un método sobrescrito no hace nada o lanza una excepción, es muy probable que estés violando el principio de sustitución de Liskov**.

Si tu código estaba usando un método que para algunas concreciones ahora lanza una excepción, ¿cómo puedes estar seguro de que todo sigue funcionando?

Otra herramienta que te avisará fácilmente son los tests. Si los tests de la clase padre no funcionan para la hija, también estarás violando este principio.

Compiler 4:



En este ejemplo, tenemos una clase base llamada **Animal** que tiene un método **makeSound()** para hacer un sonido genérico. Luego, tenemos una clase derivada llamada **Dog** que hereda de **Animal** y redefine el método **makeSound()** para hacer el sonido "Guau guau!". Además, la clase **Dog** tiene un método adicional **fetch()** para recuperar una pelota.

Luego, tenemos una función **animalMakeSound()** que toma un objeto de tipo **Animal** y llama al método **makeSound()**. Esta función demuestra el principio de sustitución de Liskov, ya que acepta tanto un objeto de tipo **Animal** como un objeto de tipo **Dog**. A pesar de que **Dog** es una clase derivada, puede ser utilizado en lugar de **Animal** sin alterar el comportamiento esperado.

En resumen, el principio de sustitución de Liskov nos ayuda a diseñar jerarquías de clases que sean flexibles y compatibles. Las clases derivadas deben poder reemplazar a las clases base sin causar problemas en el funcionamiento del programa.

## Principio de segregación de interfaces

El principio de segregación de interfaces viene a decir que **ninguna clase debería depender de métodos que no usa**. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. **En caso contrario, es mejor tener varias interfaces más pequeñas**.

**Las interfaces nos ayudan a desacoplar módulos entre sí**. Esto es así porque si tenemos una interfaz que explica el comportamiento que el módulo espera para comunicarse con otros módulos, nosotros siempre podremos crear una clase que lo implemente de modo que cumpla las condiciones.

El módulo que describe la interfaz no tiene que saber nada sobre nuestro código y, sin embargo, nosotros podemos trabajar con él sin problemas.

### El problema

La problemática surge cuando esas interfaces intentan definir más cosas de las debidas, lo que se denominan fat interfaces.

Probablemente ocurrirá que **las clases hijas acabarán por no usar muchos de esos métodos**, y habrá que darles una implementación.

Muy habitual es lanzar una excepción, o simplemente no hacer nada.

Pero, al igual que vimos en algún ejemplo en el principio de sustitución de Liskov, **esto es peligroso**. Si lanzamos una excepción, es más que probable que el módulo que define esa interfaz use el método en algún momento, y esto hará fallar nuestro programa.

El resto de implementaciones “por defecto” que podamos dar, pueden generar efectos secundarios que no esperemos, y a los que sólo podemos responder conociendo el código fuente del módulo en cuestión, cosa que no nos interesa.

### ¿Cómo detectar que estamos violando el Principio de segregación de interfaces?

Como comentaba en los párrafos anteriores, si **al implementar una interfaz ves que uno o varios de los métodos no tienen sentido y te hace falta dejarlos vacíos o lanzar excepciones**, es muy probable que estés violando este principio.

Si la interfaz forma parte de tu código, divídela en varias interfaces que definan comportamientos más específicos.

Recuerda que no pasa nada porque una clase ahora necesite implementar varias interfaces. El punto importante es que use todos los métodos definidos por esas interfaces.

Compiler 5:



En el ejemplo, se muestra una clase **Ave** que tiene métodos para volar y nadar. El cliente **Pinguino** inicialmente depende de la interfaz general **Ave**, pero solo necesita el método **nadar**. Esto viola el principio de segregación de interfaces porque el cliente se ve obligado a depender de métodos que no utiliza.

En el buen ejemplo, separamos la interfaz en dos interfaces distintas: **Volador** y **Nadador**. El cliente **Pinguino** ahora depende únicamente de la interfaz **Nadador** y no tiene acceso al método **volar**, ya que no lo necesita. De esta manera, se cumple el principio de segregación de interfaces y se evita la dependencia innecesaria.

**Principio de inversión de dependencias**

Este principio es una técnica básica, y será **el que más presente tengas en tu día a día si quieres hacer que tu código sea testable y mantenible**.

Gracias al principio de inversión de dependencias, podemos hacer que el código que es el núcleo de nuestra aplicación no dependa de los detalles de implementación, como pueden ser el framework que utilices, la base de datos, cómo te conectes a tu servidor…

Todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

La [definición](https://en.wikipedia.org/wiki/Dependency_inversion_principle) que se suele dar es:

*A. Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.*

*B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.*

Pero entiendo que sólo con esto no te quede muy claro de qué estamos hablando, así que voy a ir desgranando un poco el problema, cómo detectarlo y un ejemplo.

**El problema**

En la programación vista desde el modo tradicional, cuando un módulo depende de otro módulo, se crea una nueva instancia y la utiliza sin más complicaciones.

Esta forma de hacer las cosas, que a primera vista parece la más sencilla y natural, nos va a traer bastantes problemas posteriormente, entre ellos:

* **Las parte más genérica de nuestro código (lo que llamaríamos el dominio o lógica de negocio) dependerá por todas partes de detalles de implementación**. Esto no es bueno, porque no podremos reutilizarlo, ya que estará acoplado al framework de turno que usemos, a la forma que tengamos de persistir los datos, etc. Si cambiamos algo de eso, tendremos que rehacer también la parte más importante de nuestro programa.
* **No quedan claras las dependencias**: si las instancias se crean dentro del módulo que las usa, es mucho más difícil detectar de qué depende nuestro módulo y, por tanto, es más difícil predecir los efectos de un cambio en uno de esos módulos. También nos costará más tener claro si estamos violando algunos otros principios, como el de [Responsabilidad Única](https://devexperto.com/principio-responsabilidad-unica).
* **Es muy complicado hacer tests**: Si tu clase depende de otras y no tienes forma de sustituir el comportamiento de esas otras clases, no puedes testarla de forma aislada. Si algo en los tests falla, no tendrías forma de saber de un primer vistazo qué clase es la culpable.

**¿Cómo detectar que estamos violando el Principio de inversión de dependencias?**

Este es muy fácil: **cualquier instanciación de clases complejas o módulos es una violación de este principio**.

Además, si escribes tests te darás cuenta muy rápido, en cuanto no puedas probar esa clase con facilidad porque dependa del código de otra clase.

Te estarás preguntando entonces cómo vas a hacer para darle a tu módulo todo lo que necesita para trabajar. Tendrás que utilizar alguna de las alternativas que existen para suministrarle esas dependencias.

Aunque hay varias, las que más se suelen utilizar son **mediante constructor y mediante setters** (funciones que lo único que hacen es asignar un valor).

¿Y entonces auién se encarga de proveer las dependencias? Lo más habitual es utilizar un **inyector de dependencias**: un módulo que se encarga de instanciar los objetos que se necesiten y pasárselos a las nuevas instancias de otros objetos.

Se puede hacer una inyección muy sencilla a mano, o usar alguna de las muchas librerías que existen si necesitamos algo más complejo.

En cualquier caso esto se escapa un poco del objeto de este artículo.

Si quieres ver un caso particular y algo más sobre inyección, puedes leer [este artículo sobre inyección de dependencias en Android con Hilt](https://devexperto.com/dagger-hilt/).

Compiler 6:



En este ejemplo, tenemos dos módulos: **DBService** y **UserService**. **DBService** es un módulo de bajo nivel que se encarga de interactuar con la base de datos y proporciona un método **saveData** para guardar datos.

El módulo de alto nivel es **UserService**, que depende de **DBService**. Sin embargo, en lugar de depender directamente de **DBService**, **UserService** depende de una abstracción proporcionada por **DBService**. Esto significa que **UserService** no está acoplado directamente a **DBService** y puede funcionar con cualquier implementación que cumpla con la abstracción requerida.

En el archivo **main.js**, creamos una instancia de **DBService** y luego la pasamos como dependencia al construir la instancia de **UserService**. Luego, podemos utilizar **userService** para registrar un usuario, y este servicio internamente utiliza el método **saveData** de **dbService** para guardar los datos en la base de datos.

El principio de inversión de dependencias se cumple en este ejemplo porque **UserService** depende de una abstracción (**dbService**) en lugar de depender directamente de la implementación concreta (**DBService**). Esto facilita el intercambio de implementaciones y mejora la flexibilidad y modularidad de nuestro código.

Ejercicio de lectura de código, deben poner comentarios y ver si tiene solid y sustentar

