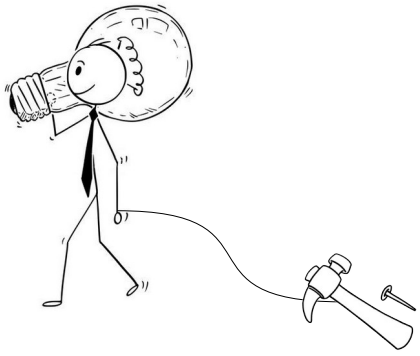
Paradigma Orientado a Objetos - Conceptos iniciales

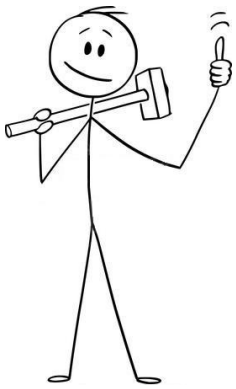
El paradigma orientado a objetos (POO) es uno de los paradigmas de programación más populares en el mundo del desarrollo de software y sistemas. Tiene su origen a fines de la década del '60 con el lenguaje Simula 67 diseñado por Ole-Johan Dahl y Kristen Nygaard, del Centro de Cómputo Noruego en Oslo, y perfeccionado en los laboratorios de Xerox-PARC con la implementación de Smalltalk.

Cuando hablamos de paradigma, hablamos de un modelo conceptual que permite guiar la resolución de problemas. Algo importante a tener en cuenta es que no todos los problemas pueden ser resueltos bajo el mismo paradigma, por lo que ante un problema es necesario elegir el paradigma apropiado para poder resolverlo de la manera más adecuada.

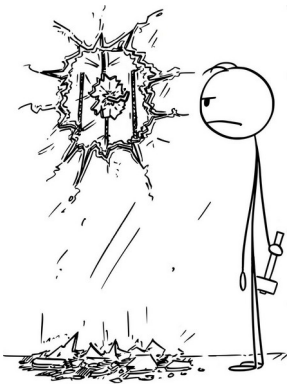
Por ejemplo, pensemos en la construcción de casas y edificios. Hasta hace unas décadas, el paradigma más utilizado para construir paredes era el de la construcción tradicional o húmeda. En ese contexto, si uno quería clavar un clavo para colgar un cuadro en una pared, sólo bastaba con agarrar un clavo y clavarlo en la pared con un martillo.



Si después queríamos hacer una ventana en otra pared, podríamos haber usado el mismo martillo, pero tardaríamos demasiado; mejor otra herramienta, como un mazo y una moladora, disponibles para el mismo método de construcción.



El problema se presenta cuando cambia el paradigma y se empieza a construir de acuerdo al método de construcción en seco. Dado el mismo problema del cuadro, si utilizamos las herramientas que ya conocíamos (clavo y martillo) terminaríamos obteniendo... ¡una hermosa ventana!



Al cambiar el método de construcción cambiaron entonces las reglas de construcción, los métodos y las herramientas adecuadas. Algo similar ocurre en el ámbito de la programación: distintos paradigmas ofrecen distintas reglas, metodologías y herramientas para resolver los problemas que se presentan.

Formalizando, un paradigma se puede definir como un modelo que especifica un marco lógico que contiene supuestos, verdades, metodologías y herramientas para la resolución de problemas. En el ámbito de la programación, un paradigma guía la manera en la que se encara el diseño y desarrollo de programas.

En el paradigma estructurado, un programa se define como la suma de:

- *algoritmos*, es decir los pasos a seguir para llegar a la solución del problema, más
- *estructuras de datos*, que sostienen la información necesaria para resolver el problema

En el POO la visión cambia, definiéndose un programa como un:

"conjunto de objetos que interactúan y colaboran entre sí mediante el intercambio de mensajes"

A partir de esta definición, se deben tener en cuenta los siguientes elementos:

- un *objeto* puede ser considerado, en principio, como una representación de un concepto de la realidad; puede ser visto como una unidad funcional, que posee *propiedades* (características) y *comportamientos* asociados. El *estado interno* de un objeto está definido por el valor de sus características en un momento dado
- un *mensaje* es aquello que un objeto sabe responder; representa un comportamiento que el objeto puede realizar
- un *método* define cómo el objeto responde a un mensaje, es decir, cómo lleva a cabo el mensaje recibido

Por poner un ejemplo, definamos un objeto perro, cuyo nombre es Spike y sabemos que puede ladrar diciendo ¡Guau!

Aquí tenemos:

un objeto	⇨	el perro
una característica del objeto	⇨	el nombre, cuyo valor es "Spike"
un mensaje	⇨	ladrar
un método	⇨	decir ¡Guau!

Entre las características fundamentales del POO se encuentran:

- la *abstracción*, que permite enfocarse en lo relevante al dominio del problema, dejando de lado factores que no lo son
- el *encapsulamiento*, que permite definir bajo una única unidad (cápsula) un concepto en particular y todo lo relacionado a él
- el *ocultamiento*, que le da la posibilidad al objeto de definir cómo se mostrará e interactuará con el resto de los objetos, ocultando su estado interno
- el *polimorfismo*, que define la capacidad que tienen distintos objetos de responder al mismo mensaje de manera distinta
- la *herencia*, que permite que un concepto adquiera o herede características y comportamientos de otro de orden superior

Para ejemplificar estos conceptos:

Abstracción	Si estamos desarrollando un programa que permita tomar asistencia en una universidad, bastará con registrar de cada alumno su nombre, su apellido y su número de legajo. Ahora bien, si lo que necesitamos es un listado de alumnos que realizan deportes, probablemente se requiera también el talle de cada uno, a fin de saber cuántos equipos de cada talle se deben comprar; el dato del talle sólo es relevante en el segundo caso, en el primero no por lo que no tiene sentido tenerlo en cuenta
Encapsulamiento	Conociendo las características que definen a una cuenta bancaria (titular, saldo) y su comportamiento (realizar extracción/depósito, consultar saldo), se justifica definir el objeto “Cuenta” y asociar a dicho objeto las características y comportamientos identificados
Ocultamiento	Una persona tiene como característica un nombre y el comportamiento de

	saber responder cuando otro objeto le pregunta cómo se llama. De esta manera, uno puede preguntarle a la persona cuál es su nombre, lo que uno no puede hacer es modificarselo. ¿Por qué? Porque la persona expone el mensaje “¿cuál es tu nombre?”, pero no el mensaje “modificar nombre”.
Polimorfismo	En un restaurant, el sueldo de un mozo se calcula teniendo en cuenta las propinas recibidas sumadas a su sueldo básico, mientras que el del cajero es un monto fijo mensual. Ambos objetos, mozo y cajero, saben responder al mensaje "¿cuál es tu sueldo?", la diferencia estará en la respuesta al mismo, es decir, en el método asociado al mensaje: el primero a su sueldo básico le sumará las propinas obtenidas, mientras que el segundo sólo responderá informando su monto fijo mensual.
Herencia	Sabemos una heladera y un aire acondicionado son conceptos distintos que tienen características y comportamientos que los distinguen uno del otro. Pero si los analizamos podemos encontrar que tienen cosas en común: ambos pueden ser considerados electrodomésticos. Y no sólo eso, sino que además hasta pueden compartir características como ser el nivel de consumo eléctrico. Siendo así podemos entonces definir la característica "consumo" dentro del concepto electrodoméstico, y definir las características específicas de heladera y aire acondicionado en cada uno de ellos. De esta manera, heladera y aire acondicionado *heredan* la característica consumo de este concepto "electrodoméstico" que los *generaliza*.

Por último, cabe aclarar que un objeto es la abstracción que el paradigma requiere. Por otro lado, el concepto de *clase* permite definir un molde para la creación o instanciación de objetos. En nuestro ejemplo del perro:

- la clase sería *Perro*, y contendría un atributo (el nombre) y un método (ladrar)
- Spike es un objeto de clase Perro, cuyo nombre es Spike y cuando recibe el mensaje ladrar dice ¡Guau!

Clases concretas vs abstractas

Hasta ahora hablamos de clases en general, con la herencia como mecanismo para definir conceptos generales en superclases y especificarlos en subclases.

Existen dos tipos de clases: las concretas y las abstractas. Las clases concretas son aquellas que representan conceptos concretos, a partir de los cuales se pueden crear o instanciar objetos. Por el contrario, las clases abstractas no son clases instanciables, es decir, no pueden tener instancias directas, ya sea porque su descripción es incompleta (le falta el método de una o más operaciones) o porque, aunque completa, originalmente no fue pensada para ser instanciada.

Definamos por ejemplo una superclase `Animal` y tres subclases `Perro`, `Gato` y `Ave`: la primera contiene comportamientos referidos a todos los animales, mientras que las últimas tres definen especificidades de cada animal en particular. Al tener este modelo y existir estas clases específicas (`Perro`, `Gato` y `Ave`), pierde sentido poder instanciar objetos de la clase `Animal`, dado que es una clase que de alguna manera se queda a mitad de camino, definiendo cosas de todos los animales pero que no describe de manera completa a ninguno.

La razón de ser de las clases abstractas es la especialización, es decir, ser heredada o extendida por subclases más específicas. Esto se debe justamente al hecho de que la clase abstracta no tiene implementadas todas sus operaciones. Definir operaciones abstractas en una clase abstracta es como si la clase dijese: *"dejo la implementación de estos comportamientos a mis subclases"*.

Cabe aclarar que una clase concreta no puede tener operaciones abstractas, mientras que una clase abstracta sí puede definir operaciones concretas además de las abstractas.

Interfaces

Las interfaces definen un conjunto de operaciones abstractas para ser implementadas por las clases que así lo deseen. Son utilizadas cuando uno necesita declarar las operaciones que las clases concretas deben implementar, pero no se lo quiere hacer a través de una clase abstracta. Recordemos que la relación de herencia, en muchos lenguajes, se limita a una única superclase, por lo que la implementación de interfaces permite que a una clase se le indique qué operaciones debe implementar sin ocupar el único lugar disponible para una herencia. Una clase puede implementar tantas interfaces como desee.

Una interfaz se puede pensar como un contrato que declara: *"estas son las operaciones que deben ser implementadas por las clases que deseen cumplir con este contrato"*.

Una interfaz no puede tener operaciones concretas, pero sí atributos, aunque estos suelen ser constantes y definidos a nivel de clase.

Cuando una clase implementa una interfaz, se dice que la realiza.

Las interfaces son muy buenas para separar el comportamiento requerido de la manera en que una clase lo implementará, es decir, el *qué* del *cómo*.

Cuando una clase implementa una interfaz, los objetos de dicha clase pueden ser referenciados tanto usando la clase como la interfaz. En otras palabras, cuando se define un atributo, su tipo podrá ser tanto del tipo específico de la clase como de cualquiera de las interfaces que ésta implementa.

Definamos como ejemplo una superclase `Felino` y dos subclases, `Leon` y `Gato`. De esta manera, estaríamos diciendo que un gato *"es un"* felino. Hasta aquí no hay problemas. Pero ¿qué pasaría si, además, necesitamos que la clase `Gato` implemente comportamientos propios de una mascota? La herencia ya la tiene cubierta, heredando de `Felino`, pero sabemos que no tiene límite en cuanto a cuántas interfaces puede implementar. Entonces, definiendo `Mascota` como una interfaz y haciendo que `Gato` la implemente o realice, podríamos decir que un gato *"es un"* felino, que *"se comporta como"* una mascota. De esta manera `Gato` hereda todas las operaciones propias de `Felino`, y además implementa todas las relacionadas con una `Mascota`.