PROGRAMACIÓN ORIENTADA A OBJETOS

Los lenguajes de programación antiguos, como C, Basic o COBOL, seguían un estilo procedimental a la hora de escribir código. Es decir, los programas escritos en estos lenguajes consistían en una serie de instrucciones, una detrás de otra, que se ejecutaban paso a paso. Para "encerrar" funcionalidad y poder reutilizarla definían procedimientos (también llamados subrutinas o funciones), pero se usaban datos globales y era muy complicado aislar los datos específicos unos de otros. Podríamos decir que este tipo de lenguajes se centraban más en la lógica que en los datos.

Sin embargo, los lenguajes modernos como C#, Java o... en realidad casi cualquiera, utilizan otros paradigmas para definir los programas. Entre éstos, el paradigma más popular es el que se refiere a la **Programación Orientada a Objetos** o **POO**.

En este paradigma, los programas se modelan en torno a **objetos** que aglutinan toda la funcionalidad relacionada con ellos. De este modo en lugar de crear una serie de funciones sin conexión alguna entre ellas, en POO se crean **clases**, que representan entidades que quieres manejar en tu programa. Por ejemplo, facturas, líneas de factura, clientes, coches... o cualquier entidad que necesites gestionar conceptualmente. En ese sentido, **la POO gira más en torno a los datos que en torno a la lógica**, que se delega a un segundo plano, ya que forma parte de dichos datos como veremos enseguida.

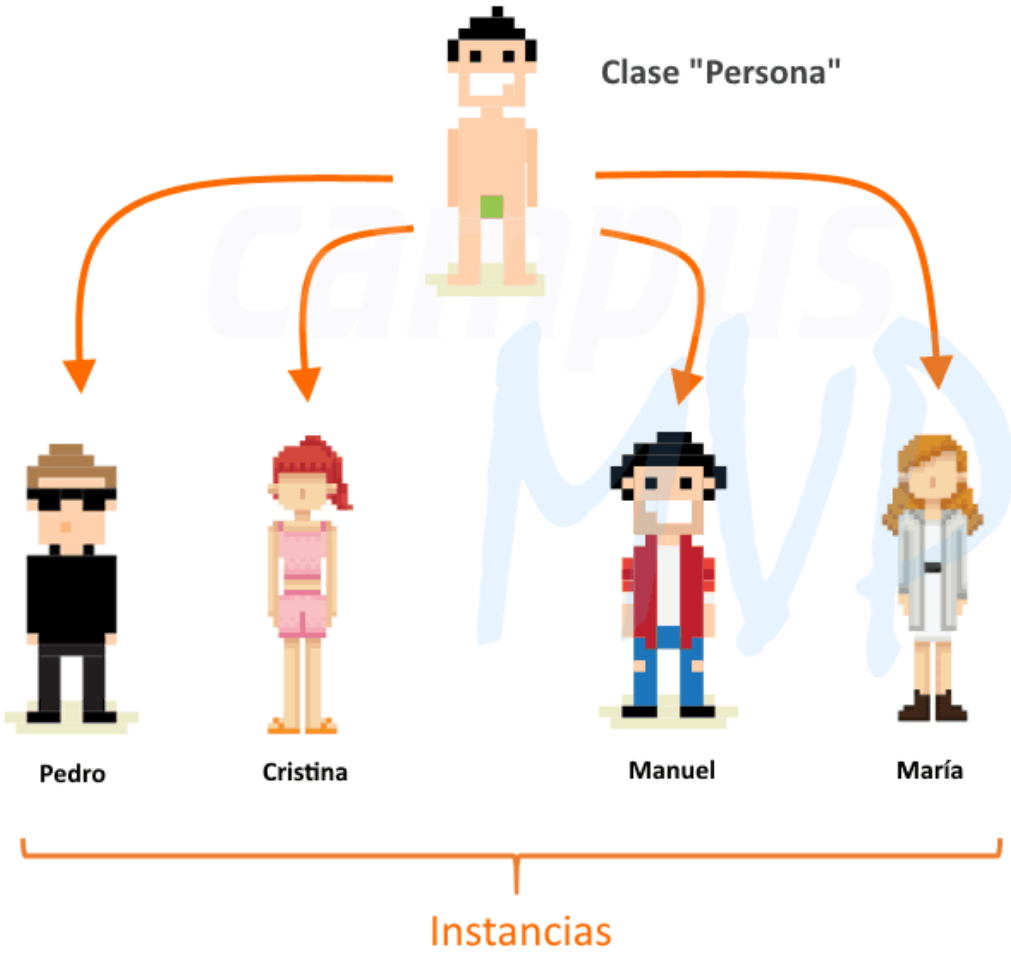
Clases, objetos e instancias

El primer concepto y más importante de la POO es **la distinción entre clase y objeto**.

Una clase es una plantilla. Define de manera genérica cómo van a ser los objetos de determinado tipo. Por ejemplo, en un juego, una clase para representar a personas puede llamarse `Persona` y tener una serie de **atributos** como `Nombre`, `Apellidos` o `Edad` (que normalmente son propiedades), y una serie de **comportamientos** que pueden tener, como `Hablar()`, `Caminar()` o `Comer()` y que se implementan como **métodos de la clase** (funciones).

Una clase por sí sola no sirve de nada, pues no es más que un concepto, sin entidad real. Para poder utilizar una clase en un programa lo que hay que hacer es **instanciarla**. Instanciar una clase consiste en **crear un nuevo objeto concreto de**

la misma. Es decir, **un objeto es ya una entidad concreta** que se crea a partir de la plantilla que es la clase. Este nuevo objeto tiene ya "existencia" real, puesto que ocupa memoria y se puede utilizar en el programa. Así un objeto puede ser una persona que se llama `Cristina López`, de 37 años y que en nuestro programa podría hablar, caminar o comer, que son los comportamientos que están definidos en la clase.



De este modo, si tenemos que manejar personas podemos ir creándolas a medida que las necesitemos, y actuar sobre ellas individualmente. Cada una tiene sus propios datos y sus propias acciones.

La clase define de forma genérica cómo son las personas, y los objetos son personas concretas.

La POO es muy potente porque nos permite modelar de manera sencilla datos y comportamientos complejos del mundo real. Al poder manejar los datos y los

comportamientos de cada objeto de manera independiente nos evita tener que mantener datos globales y coordinar todo eso. En su momento fue una verdadera revolución.

Los cuatro pilares de la POO

Lo anterior por si solo es estupendo. Sin embargo, no es suficiente. Para poder manejar de manera eficiente las clases y los objetos que se generan con la Programación Orientada a Objetos son necesarios algunos principios que nos ayudarán a reducir la complejidad, ser más eficientes y evitar problemas. **Son los 4 pilares de la POO**. Todos los lenguajes orientados a objetos los implementan de una u otra manera, y es indispensable conocerlos bien.

Vamos a verlos

Pilar 1: Encapsulación

El concepto de encapsulación es el más evidente de todos. Pero, precisamente por su sencillez, a veces pasa inadvertido.

La encapsulación es la característica de un lenguaje POO que **permite que todo lo referente a un objeto quede aislado dentro de éste**. Es decir, que todos los datos referentes a un objeto queden "encerrados" dentro de éste y sólo se puede acceder a ellos a través de los miembros que la clase proporcione (propiedades y métodos).

Por ejemplo, en el caso de las personas que estábamos viendo, toda la información sobre éstas (nombre, apellidos, edad... y cualquier otro dato interno que se utilice y que no necesariamente se ve desde el exterior del objeto) está circunscrito al ámbito de dicha persona.

Así, internamente tenemos un dato que es el nombre de la persona y accedemos a él a través de la propiedad pública `Nombre` que define la clase que representa a las personas. De este modo damos acceso sólo a lo que nos interese y del modo que nos interese. En un lenguaje tradicional tendríamos que montar alguna estructura global para almacenar esa información y luego acceder ordenadamente a ella, sin mezclar los datos de una persona con los de otra.

> Gracias a la encapsulación, toda la información de un objeto está contenida dentro del propio objeto.

Pilar 2: Abstracción

Este concepto está muy relacionado con el anterior.

Como la propia palabra indica, el principio de abstracción lo que implica es que la clase debe **representar las características de la entidad hacia el mundo exterior, pero ocultando la complejidad** que llevan aparejada. O sea, nos abstrae de la complejidad que haya dentro dándonos una serie de atributos y comportamientos (propiedades y funciones) que podemos usar sin preocuparnos de qué pasa por dentro cuando lo hagamos.

Así, una clase (y por lo tanto todos los objetos que se crean a partir de ella) debe exponer para su uso solo lo que sea necesario. Cómo se haga "por dentro" es irrelevante para los programas que hagan uso de los objetos de esa clase.

En nuestro ejemplo de las personas en un juego, puede ser que tengamos un dato interno que llamamos `energía` y que nunca es accesible directamente desde fuera. Sin embargo, cada vez que la persona anda (o corre, si tuviésemos un método para ello) gasta energía y el valor de este dato disminuye. Y cuando la persona come, el valor sube en función de lo que haya comido.

Otro ejemplo incluso más claro podría ser la acción `Hablar()`. Ésta puede suponer que se genere una voz sintética a partir de un texto que se le indica como parámetro de la acción para lo cual quizá ocurran un montón de cosas: se llama a un componente para síntesis de voz que está en la nube, se lanza la síntesis de voz en el dispositivo local de audio y se anota en una base de datos la frase que se ha pronunciado para guardar un histórico entre otras cosas. Pero todo esto es indiferente para el programa que hace uso de esta funcionalidad. El programa simplemente tiene acceso a un objeto `Cristina` y llama a la función `Hablar()`. No tiene ni idea de toda la complejidad interna que puede suponer. Si mañana cambiamos el modo de sintetizar la voz o cualquier otra acción interna, es indiferente para el programa que usa nuestros objetos de tipo `Persona`.

La abstracción está muy relacionada con la encapsulación, pero va un paso más allá pues no sólo controla el acceso a la información, sino también oculta la complejidad de los procesos que estemos implementando.

Pilar 3: Herencia

Desde el punto de vista de la genética, cuando una persona obtiene de sus padres ciertos rasgos (el color de los ojos o de la piel, una enfermedad genética, etc...) se

dice que los hereda. Del mismo modo **en POO cuando una clase hereda de otra obtiene todos los rasgos que tuviese la primera**.

Dado que una clase es un patrón que define cómo es y cómo se comporta una cierta entidad, una clase que hereda de otra obtiene todos los rasgos de la primera y **añade otros nuevos** y además también **puede modificar algunos de los que ha heredado**.

A la clase de la que se hereda se le llama **clase base**, y a la clase que hereda de ésta se le llama **clase derivada**.

Así, en nuestro juego que involucra personas, podemos tener clases de personas más especializadas para representar personajes especiales del juego. Por ejemplo, podríamos definir clases como `Pirata`, `Piloto` o `Estratega` que heredan de la clase `Persona`. Todos los objetos de estas clases heredan las propiedades y los métodos de `Persona`, pero pueden particularizar algunos de ellos y además añadir cosas propias.

Por ejemplo, los objetos de la clase `Pirata` tienen un método nuevo que es `Abordar()` que en el juego sirve para asaltar un barco enemigo. Pero además presentan una propiedad que solo tienen los piratas llamada `Sobrenombre`, que es el nombre por el que se les conoce (un pirata puede ser de nombre `Hızır` y de apellido `bin Yakup` pero su sobrenombre es `Barbaroja`).

No solo eso. Lo bueno de la herencia es que podemos reutilizar todo lo que tuviésemos en la clase base. Supongamos que en nuestro juego, los piratas hablan de forma diferente a los demás. El método `Hablar()` se modifica para que le añada un `¡Arrrr!` o un `¡Por todos los demonios!` aleatoriamente a la frase y que así parezca más un pirata . Para que el pirata hable no tendríamos que volver a hacer todo el código relacionado con hablar. Eso ya sabe cómo hacerlo por el mero hecho de ser una persona (por heredar de la clase `Persona`). Lo único que tendríamos que hacer es añadir esas interjecciones de pirata a la frase y luego delegar la síntesis de voz y todo lo demás a la clase base. Sería facilísimo y conseguiríamos consistencia entre todas las clases a la hora de particularizar la forma de hablar.

La herencia es una de las características más potentes de la POO ya que fomenta la reutilización del código permitiendo al mismo tiempo la particularización o especialización del mismo.

Pilar 4: Polimorfismo

La palabra polimorfismo viene del griego "polys" (muchos) y "morfo" (forma), y quiere decir "cualidad de tener muchas formas".

En POO, el concepto de polimorfismo se refiere al hecho de que **varios objetos de diferentes clases, pero con una base común, se pueden usar de manera indistinta**, sin tener que saber de qué clase exacta son para poder hacerlo.

Supongamos que en nuestro juego tenemos un montón de personajes que están juntos en un mismo escenario. Hay varios piratas, algunos estrategas y un montón de otros tipos de personas. En un momento dado necesitamos que todos se pongan a hablar. Cada uno lo hace de una forma diferente, ya que son tipos de personajes distintos. Sería algo bastante tedioso tener que localizar primero a los de un tipo y hacerlos hablar, lo luego a los de otro y así sucesivamente. La idea es que puedas tratarlos a todos como personas, independientemente del tipo específico de persona que sean y simplemente decirles que hablen.

Al derivar todos de la clase `Persona` todos pueden hablar, y al llamar al método `Hablar()` de cada uno de ellos se utilizará el proceso adecuado según el tipo (los piratas meterán sus expresiones adicionales que hemos visto, los pilotos dirán "Entrando en pista" o lo que sea, y los estrategas añadirán a todo "Déjame que lo piense bien"). Todo esto de manera transparente para el programador. Esto es el polimorfismo.

De hecho, el polimorfismo puede ser más complicado que eso ya que se puede dar también mediante la sobrecarga de métodos y, sobre todo, a través del uso de interfaces, pero el concepto es el que acabo de explicar.

El polimorfismo nos permite utilizar a los objetos de manera genérica, aunque internamente se comporten según su variedad específica.

Gracias a estos cuatro principios que cumplen todos los lenguajes orientados a objetos se facilita mucho la programación de ciertos tipos de problemas, se minimizan errores, se escribe código más rápido y se puede mantener más fácilmente cuando haya modificaciones en el futuro.

Cada lenguaje tiene su sintaxis específica para crear objetos y expresar los cuatro pilares, pero estos conocimientos genéricos te valdrán para cualquiera de ellos. La próxima vez que te los pregunten en una entrevista de trabajo seguro que ya no tienes problema para explicarlos