Qué es el polimorfismo en la Programación Orientada a Objetos, el motivo de su existencia y cómo implementar polimorfismo en clases y objetos.

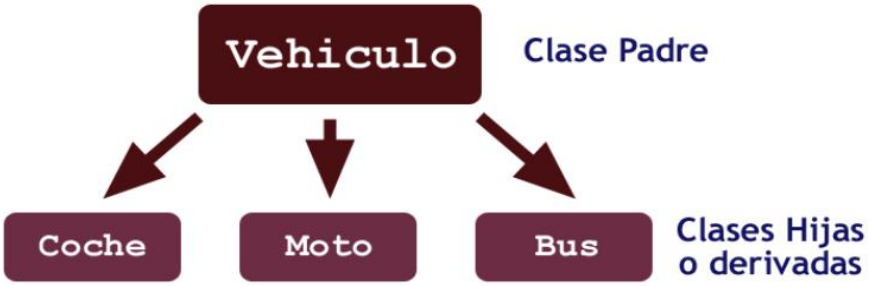
El concepto de polimorfismo es en realidad algo muy básico.

Definición: El polimorfismo es una relajación del sistema de tipos, de tal manera que una referencia a una clase (atributo, parámetro o declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas, …).

Herencia y las clasificaciones en Programación Orientada a Objetos

Para poder entender este concepto de OOP necesitamos entender otras cosas previas, como es el caso de la herencia.

Veremos que el polimorfismo y la herencia son dos conceptos estrechamente ligados. Conseguimos implementar polimorfismo en jerarquías de clasificación que se dan a través de la herencia. Por ejemplo, tenemos una clase vehículo y de ella dependen varias clases hijas como coche, moto, autobús, etc.



Pero antes de entender todo esto, queremos ir un poco más hacia atrás, entendiendo lo que es un sistema de tipos.

```
Largometraje miLargo = new Largometraje("Lo que el viento se llevó");
```

Esa variable "miLargo", por declaración tendrá una referencia a un objeto de la clase "Largometraje". Pues bien, durante toda su vida, deberá tener siempre una referencia a cualquier objeto de la misma clase.

Volviendo al ejemplo de los vehículos, si defino una variable que apunta a un objeto de clase "Coche", durante toda la vida de esa variable tendrá que contener un objeto de la clase Coche, no pudiendo más adelante apuntar a un objeto de la clase Moto o de la clase Bus. Esta rigidez no existe en los lenguajes débilmente tipados como es el caso de Javascript o PHP, sin embargo, es una característica habitual de lenguajes como Java, que son fuertemente tipados.

```
Coche miCoche = new Coche("Ford Focus 2.0");

//la variable miCoche apunta a un objeto de la clase coche
```

```
//si lo deseo, mañana podrá apuntar a otro objeto diferente, pero siempre
tendrá que ser de la clase Coche
miCoche = new Coche("Renault Megane 1.6");
```

Lo que nunca podré hacer es guardar en esa variable, declarada como tipo Coche, otra cosa que no sea un objeto de la clase Coche.

```
//si miCoche fue declarada como tipo Coche, no puedo guardar un objeto de la
clase Moto
miCoche = new Moto("Yamaha YBR");
//la línea anterior nos daría un error en tiempo de compilación
```

Entendida esa premisa, pensemos en el concepto de función y su uso en lenguajes de tipado estático.

Cuando en un lenguaje fuertemente tipado declaramos una función, siempre tenemos que informar el tipo de los parámetros que va a recibir. Por ejemplo, la función "sumaDosNumeros()" recibirá dos parámetros, que podrán ser de tipo entero.

```
function sumaDosNumeros(int num1, int num2)
```

A esta función, tal como está declarada, no le podremos pasar como parámetros otra cosa que no sean variables -o literales- con valores de número entero. En caso de pasar otros datos con otros tipos, el compilador te avisará.

Esto mismo de los parámetros en las funciones te ocurre también con los atributos de las clases, cuyos tipos también se declaran, con los datos que se insertan en un array, etc.

Polimorfismo en objetos

Tal como funcionan los lenguajes fuertemente tipados, una variable siempre deberá apuntar a un objeto de la clase que se indicó en el momento de su declaración. Una función cuyo parámetro se haya declarado de una clase, solo te aceptará recibir objetos de esa clase. Un array que se ha declarado que es de elementos de una clase determinada, solo aceptará que rellenemos sus casillas con objetos de esa clase declarada.

```
Vehiculo[] misVehiculos = new Vehiculo[3];
```

Esa variable misVehiculos es un array y en ella hemos declarado que el contenido de las casillas serán objetos de la clase "Vehiculo". Como se ha explicado, en lenguajes fuertemente tipados sólo podría contener objetos de la clase Vehiculo. Pues bien, polimorfismo es el mecanismo por el cual podemos "relajar el sistema de tipos", de modo que nos acepte también objetos de las clases hijas o derivadas.
Por tanto, la "relajación" del sistema de tipos no es total, sino que tiene que ver con las clasificaciones de herencia que tengas en tus sistemas de clases. Si defines un array con casillas de una determinada clase, el compilador también te aceptará que metas en esas casillas objetos de una clase hija de la que fue declarada. Si declaras que una función recibe como parámetros objetos de

una determinada clase, el compilador también te aceptará que le envíes en la invocación objetos de una clase derivada de aquella que fue declarada.

En concreto, en nuestro array de vehículos, gracias al polimorfismo podremos contener en los elementos del array no solo vehículos genéricos, sino también todos los objetos de clases hijas o derivadas de la clase "Vehiculo", es decir objetos de la clase "Coche", "Moto", "Bus" o cualquier hija que se haya definido.

Para qué nos sirve en la práctica el polimorfismo

Volvamos a la clase "Largometraje" y ahora pensemos en la clase "Cine". En un cine se reproducen largometrajes. Podemos tener varios tipos de largometrajes, como películas o documentales, etc. Laas películas y documentales tienen diferentes características, distintos horarios de audiencia, distintos precios para los espectadores y por ello decidimos que la clase "Largometraje" tenga clases hijas o derivadas como "Película" y "Documental".

Imagina que en tu clase "Cine" creas un método que se llama "reproducir()". Este método podrá recibir como parámetro aquello que quieres emitir en una sala de cine y podrán llegarte a veces objetos de la clase "Película" y otras veces objetos de la clase "Documental". Debido a que los métodos declaran los tipos de los parámetros que reciben, habrá que hacer algo como esto:

```
reproducir(Pelicula peliculaParaReproducir)
```

Pero si luego tienes que reproducir documentales, tendrás que declarar:

```
reproducir(Documental documentaParaReproducir)
```

Probablemente el código de ambos métodos sea exactamente el mismo. Poner la película en el proyector, darle al play, crear un registro con el número de entradas vendidas, parar la cinta cuando llega al final, etc. ¿Realmente es necesario hacer dos métodos? Y si más adelante nos surge una tercera opción para reproducir ¿Tendremos que crear un nuevo método reproducir() sobre la clase "Cine" que nos acepte ese tipo de emisión? ¿es posible ahorrarnos todo ese mantenimiento?

Aquí es donde el polimorfismo nos ayuda. Podremos crear perfectamente un método "reproducir()" que recibe un largometraje y donde podremos recibir todo tipo de elementos, películas, documentales y cualquier otra cosa similar que sea creada en el futuro.

Entonces lo que te permiten hacer los lenguajes es declarar el método "reproducir()" indicando que el parámetro que vas a recibir es un objeto de la clase padre "Largometraje", pero donde realmente el lenguaje y compilador te aceptan cualquier objeto de la clase hija o derivada, "Película", "Documental", etc.

```
reproducir(Largometraje elementoParaReproducir)
```

Podremos crear películas y reproducirlas, también crear documentales para luego reproducir y todos estos objetos son aceptados por el método "reproducir()", gracias a la relajación del sistema de tipos. Incluso, si queremos reproducir otro tipo de cinta, no habrá que tocar la clase "Cine" y el método "reproducir()". Siempre que aquello que queramos reproducir sea de la clase "Largometraje" o una clase hija, el método lo aceptará.

Otro ejemplo: volviendo de nuevo a la clase Vehiculo. Además, nos centramos en la utilidad del polimorfismo y sus posibilidades para reducir el mantenimiento de los programas informáticos, que es lo que realmente interesante.

Tenemos la clase Parking. Dentro de esta tenemos un método estacionar(). Puede que en un parking tenga que estacionar coches, motos o autobuses. Sin polimorfismo tendría que crear un método que permitiese estacionar objetos de la clase "Coche", otro método que acepte objetos de la clase "Moto" para estacionarlos, etc. Pero estacionar un coche, una moto o un bus es bastante similar: "entrar en el parking, recoger el ticket de entrada, buscar una plaza, situar el vehículo dentro de esa plaza...".

Lo ideal sería que nuestro método permita recibir todo tipo de vehículos para estacionarlos, primero por reutilización del código, ya que es muy parecido estacionar uno u otro vehículo, pero además porque así, si mañana el mercado trae otro tipo de vehículos, como un todoterreno hibrido, o una furgoneta, nuestro software sea capaz de aceptarlos sin tener que modificar la clase Parking.

Gracias al polimorfismo, cuando declaro la función estacionar() puedo decir que recibe como parámetro un objeto de la clase "Vehiculo" y el compilador me aceptará no solamente vehículos genéricos, sino todos aquellos objetos que hayamos creado que hereden de la clase Vehículo, es decir, coches, motos, buses, etc. Esa relajación del sistema de tipos para aceptar una gama de objetos diferente es lo que llamamos **polimorfismo**.