Qué es la Programación Orientada a Objetos

Por **Miguel Ángel Álvarez**

Introducimos para los más profanos las bases sobre las que se asienta la Programación Orientada a Objetos.
La programación Orientada a objetos (POO) es una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación.

Con la POO tenemos que aprender a pensar las cosas de una manera distinta, para escribir nuestros programas en términos de objetos, propiedades, métodos y otras cosas que veremos rápidamente para aclarar conceptos y dar una pequeña base que permita soltarnos un poco con este tipo de programación.

Motivación

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que los esfuerzos de los programadores puedan ser utilizados por otras personas se creó la POO. Que es una serie de normas de realizar las cosas de manera que otras personas puedan utilizarlas y adelantar su trabajo, de manera que consigamos que el código se pueda reutilizar.

La POO no es difícil, pero es una manera especial de pensar, a veces subjetiva de quien la programa, de manera que la forma de hacer las cosas puede ser diferente según el programador. Aunque podamos hacer los programas de formas distintas, no todas ellas son correctas, lo difícil no es programar orientado a objetos sino programar bien. Programar bien es importante porque así nos podemos aprovechar de todas las ventajas de la POO.

Cómo se piensa en objetos

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar.

Pues en un esquema POO el coche sería el objeto, las propiedades serían las características como el color o el modelo y los métodos serían las funcionalidades asociadas como ponerse en marcha o parar.

Por poner otro ejemplo vamos a ver cómo modelizaríamos en un esquema POO una fracción, es decir, esa estructura matemática que tiene un numerador y un denominador que divide al numerador, por ejemplo 3/2.
La fracción será el objeto y tendrá dos propiedades, el numerador y el denominador. Luego podría tener varios métodos como simplificarse, sumarse con otra fracción o número, restarse con otra fracción, etc.

Estos objetos se podrán utilizar en los programas, por ejemplo en un programa de matemáticas harás uso de objetos fracción y en un programa que gestione un taller de coches utilizarás objetos coche. Los programas

Orientados a objetos utilizan muchos objetos para realizar las acciones que se desean realizar y ellos mismos también son objetos. Es decir, el taller de coches será un objeto que utilizará objetos coche, herramienta, mecánico, recambios, etc.

Clases en POO

Las clases son declaraciones de objetos, también se podrían definir como abstracciones de objetos. Esto quiere decir que la definición de un objeto es la clase. Cuando programamos un objeto y definimos sus características y funcionalidades en realidad lo que estamos haciendo es programar una clase. En los ejemplos anteriores en realidad hablábamos de las clases coche o fracción porque sólo estuvimos definiendo, aunque por encima, sus formas.

Propiedades en clases

Las propiedades o atributos son las características de los objetos. Cuando definimos una propiedad normalmente especificamos su nombre y su tipo. Nos podemos hacer a la idea de que las propiedades son algo así como variables donde almacenamos datos relacionados con los objetos.

Métodos en las clases

Son las funcionalidades asociadas a los objetos. Cuando estamos programando las clases las llamamos métodos. Los métodos son como funciones que están asociadas a un objeto.

Objetos en POO

Los objetos son ejemplares de una clase cualquiera. Cuando creamos un ejemplar tenemos que especificar la clase a partir de la cual se creará. Esta acción de crear un objeto a partir de una clase se llama instanciar (que viene de una mala traducción de la palabra instace que en inglés significa ejemplar). Por ejemplo, un objeto de la clase fracción es por ejemplo 3/5. El concepto o definición de fracción sería la clase, pero cuando ya estamos hablando de una fracción en concreto 4/7, 8/1000 o cualquier otra, la llamamos objeto.

Para crear un objeto se tiene que escribir una instrucción especial que puede ser distinta dependiendo el lenguaje de programación que se emplee, pero será algo parecido a esto.

```
miCoche = new Coche()
```

Con la palabra new especificamos que se tiene que crear una instancia de la clase que sigue a continuación. Dentro de los paréntesis podríamos colocar parámetros con los que inicializar el objeto de la clase coche.

Estados en objetos

Cuando tenemos un objeto sus propiedades toman valores. Por ejemplo, cuando tenemos un coche la propiedad color tomará un valor en concreto, como por ejemplo rojo o gris metalizado. El valor concreto de una propiedad de un objeto se llama estado.

Para acceder a un estado de un objeto para ver su valor o cambiarlo se utiliza el operador punto.

miCoche.color = rojo

El objeto es miCoche, luego colocamos el operador punto y por último el nombre e la propiedad a la que deseamos acceder. En este ejemplo estamos cambiando el valor del estado de la propiedad del objeto a rojo con una simple asignación.

Mensajes en objetos

Un mensaje en un objeto es la acción de efectuar una llamada a un método. Por ejemplo, cuando le decimos a un objeto coche que se ponga en marcha estamos pasándole el mensaje ponte en marcha.
Para mandar mensajes a los objetos utilizamos el operador punto, seguido del método que deseamos invocar.

miCoche.ponerseEnMarcha()

En este ejemplo pasamos el mensaje ponerseEnMarcha(). Hay que colocar paréntesis igual que cualquier llamada a una función, dentro irían los parámetros.

Otras cosas

Hay mucho todavía que conocer de la POO ya que sólo hemos hecho referencia a las cosas más básicas. También existen mecanismos como la herencia y el polimorfismo que son unas de las posibilidades más potentes de la POO.

La herencia sirve para crear objetos que incorporen propiedades y métodos de otros objetos. Así podremos construir unos objetos a partir de otros sin tener que reescribirlo todo. Puedes encontrar en DesarrolloWeb.com un artículo completo dedicado a la Herencia.

El polimorfismo sirve para que no tengamos que preocuparnos sobre lo que estamos trabajando, y abstraernos para definir un código que sea compatible con objetos de varios tipos. Puedes acceder a otro artículo para saber más sobre Polimorfismo.

Son conceptos avanzados que cuesta explicar en las líneas de ese informe. No hay que olvidar que existen libros enteros dedicados a la POO y aquí solo pretendemos dar un repaso a algunas cosas para que os suenen cuando tengáis que poneros delante de ellas en los lenguajes de programación que debe conocer un desarrollador del web. Sin embargo, si quieres saber más también puedes continuar leyendo en DesarrolloWeb.com, en el manual de la teoría de la Programación orientada a objetos.

Ejemplo concreto de programación orientada a objetos

Para conseguir un ejemplo concreto de lo que es la programación orientada a objetos, podemos entrar en el Manual de PHP 5. Realmente este manual explica las características de orientación a objetos de PHP 5 y ofrece ejemplos concretos de creación de clases con características como herencia, polimorfismo, etc.

Herencia en Programación Orientada a Objetos

Concepto de herencia en la programación orientada a objetos: un mecanismo básico por el que las clases hijas heredan el código de las clases padre.

Si ya entendiste lo que son clases y objetos, atributos y estados, métodos y mensajes, ahora puedes ampliar la información en el presente texto para conocer acerca de la herencia. Pero antes de ello, centrémonos en entender algunas de las prácticas más útiles y deseables de la programación en general.

Jerarquización

Es un proceso por el cual se crean organizaciones de elementos en distintos niveles. No es un concepto específicamente de POO, sino que es algo que vemos en la vida real en muchos ámbitos, algo inherente a cualquier tipo de sistema. Puedo tener diversos tipos de jerarquías, como clasificación o composición.

Composición: Es cuando unos elementos podemos decir que están compuestos de otros, o que unos elementos están presentes en otros. Por ejemplo, el sistema respiratorio y los pulmones, la nariz, etc. Podemos decir que los pulmones están dentro del sistema respiratorio, así como dentro de los pulmones encontramos bronquios y alvéolos. En esta jerarquía de elementos tenemos composición porque donde unos forman parte de otros. En una factura también podemos decir que puede haber una jerarquía de composición. La factura tiene un cliente, varios conceptos facturables, un impuesto, etc.

Clasificación: Este tipo de jerarquización indica que unos elementos son una especialización de otros. Por ejemplo, los animales, donde tenemos vertebrados e invertebrados. Luego, dentro de los vertebrados encontramos aves, reptiles, mamíferos, etc. En los mamíferos encontramos perros, vacas, conejos... Éste es el tipo de jerarquización en que quiero que te fijes.

Los lenguajes de programación orientados a objetos son capaces de crear jerarquizaciones basadas en composición con lo que ya sabemos de clases y objetos. Eso es porque podemos tener como propiedades de objetos, otros objetos. Por ejemplo, en el caso de la factura, podríamos tener como propiedades el cliente, el impuesto, la lista de conceptos facturables, etc. Sin embargo, para hacer jerarquías de clasificación nos hace falta conocer la herencia.

Reutilización del código

Por otra parte, otro de los mecanismos que cualquier lenguaje de programación debe proveer es la posibilidad de reutilizar el código. En la programación estructurada tenemos las funciones, así que ya hemos podido reutilizar código de alguna manera. Así pues, el equivalente a las funciones, los métodos, ya nos da un grado de reutilización, pero no llegan al nivel de potencia de las que encontraremos en la herencia.

No necesitamos decirte mucho más para entender las bondades de la reutilización: en inglés lo resume el término "DRY", Don't Repeat Yourself (no te repitas) y es uno de los enunciados que debes tener más presente cuando programas. "No es mejor programador quien más líneas de código hace, sino quien mejor las reutiliza".

Quizás está de más decirlo, porque seguro que ya sabes que debemos evitar escribir dos veces el mismo código, evitar los copia/pega y pensar que la reutilización nos ayuda seriamente en el mantenimiento del software. Enseguida verás cómo la herencia es un mecanismo fundamental para reutilizar código.

Herencia en la POO

Ahora que ya conoces dos beneficios que nos proporciona la herencia y por qué es algo tan deseable en la programación, creo que te sentirás motivado para profundizar en las bases de este mecanismo, herencia, clave de la Orientación a Objetos.

La herencia es la transmisión del código entre unas clases y otras. Para soportar un mecanismo de herencia tenemos dos clases: la clase padre y la/s clase/s hija/s. La clase padre es la que transmite su código a las clases hijas. En muchos lenguajes de programación se declara la herencia con la palabra "extends".

```
class Hija extends Padre{ }
```

Eso quiere decir que todo el código de la clase padre se transmite, tal cual, a la clase hija. Si lo quieres ver así, es como si tuvieras escrito, línea a línea, todo el código de la class "Padre" dentro de las llaves de la class "Hija". Por eso, la herencia es fundamental para reutilizar código, porque no necesitas volver a incorporar el código de Padre en Hija, sino que realmente al hacer el "extends" es como si ya estuviera ahí.

Ejemplo de herencia

Volvamos a los animales, pensemos en los mamíferos. Todos tienen una serie de características, como meses de gestación en la barriga de la madre, pechos en las hembras para amamantar y luego funcionalidades como dar a luz, mamar, etc. Eso quiere decir que cuando realices la clase perro vas a tener que implementar esos atributos y métodos, igual que la clase vaca, cerdo, humano, etc.

¿Te parecería bien reescribir todo ese código común en todos los tipos de mamíferos, o prefieres heredarlo? en este esquema tendríamos una clase mamífero que nos define atributos como numero_mamas, meses_gestacion y métodos como dar_a_luz(), mamar(). Luego tendrías la clase perro que extiende (hereda) el código del mamífero, así como las vacas, que también heredan de mamífero y cualquiera de los otros animales de esta clasificación.

Otro ejemplo, tenemos alumnos universitarios. Algunos son alumnos normales, otros Erasmus y otros becarios. Probablemente tendremos una clase Alumno con una serie de métodos como asistir_a_clase(), hacer_examen() etc., que son comunes a todos los alumnos, pero hay operaciones que son diferentes en cada tipo de alumno como pagar_mensualidad() (los becarios no pagan) o matricularse() (los Erasmus que son estudiantes de intercambio, se matriculan en su universidad de origen).

Lo que debes observar es que con la herencia siempre consigues clases hijas que son una especialización de la clase padre. Para saber si está correcto emplear herencia entre unas clases y otras, plantéate la pregunta ¿CLASE HIJA es un CLASE PADRE? (por ejemplo, ¿un perro es un mamífero? ¿Un becario es un alumno de universidad?)

Nota: Existen otros modos de decir clases hija, como clase heredada, clase derivada, etc.

Otras cosas que tienes que saber sobre herencia

En este artículo nos hemos limitado a hablar sobre el concepto de herencia, pero no sobre una serie de mecanismos asociados que resultan clave para entender todavía mejor las posibilidades de esta capacidad de la POO. Nos referimos a la visibilidad de propiedades y métodos entre clases padre e hija, la posibilidad de hacer clases abstractas, que son las que contienen métodos abstractos o incluso propiedades abstractas. Hemos dejado de lado asuntos como la herencia múltiple, que te proporciona la posibilidad de heredar de varias clases a la vez (los ejemplos mencionados son de herencia simple).

Todo eso es algo que tendrás que aprender en otros textos, futuros artículos o en la referencia de tu propio lenguaje de programación. Nosotros esperamos que el presente texto te haya aclarado el concepto de una forma amena, que es lo más fundamental para que a partir de aquí tengas la base suficiente para profundizar en las características de tu propio lenguaje de programación con orientación a objetos.

Polimorfismo en Programación Orientada a Objetos

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. Realmente, cuando estamos aprendiendo Programación Orientada a Objetos (también conocida por sus siglas POO / OOP) muchos estudiantes nos hacemos un embolado tremendo al tratar de entender el concepto, pero en su base es algo extremadamente sencillo.

Trataremos de explicarlo en este artículo con palabras sencillas, pero para los valientes, aquí va una primera definición que no es mía y que carece de la prometida sencillez. Pero no te preocupes, pues la entiendas o no, luego lo explicaré todo de manera más llana.

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, …).

Nota: Esta es la definición académica que nos ofrece el profesor de la UPM Luis Fernández, del que fui alumno en la universidad y en EscuelaIT.

Herencia y las clasificaciones en Programación Orientada a Objetos

Para poder entender este concepto de OOP necesitas entender otras cosas previas, como es el caso de la herencia. Esto lo hemos explicado en un artículo anterior en DesarrolloWeb.com: Herencia en la Programación Orientada a Objetos.

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.

Por qué el sistema de tipos es importante en Polimorfismo

Muchos de los lectores que asumo se introducen en el concepto de polimorfismo a través de este artículo han aprendido a programar en lenguajes débilmente tipados, como es el caso de PHP y Javascript. Por ello es conveniente entender cómo es un lenguaje fuertemente tipado, como es el caso de Java o C.

En estos lenguajes, cuando defino una variable, siempre tengo que decir el tipo de datos que va a contener esta variable. Por ejemplo:

```
int miNumero;
```

Así le indicamos que la variable declarada "miNumero" va a contener siempre un entero. Podrás asignarle diversos valores, pero siempre deben de ser números enteros. De lo contrario el compilador te lanzará un mensaje de error y no te permitirá compilar el programa que has realizado.

Esto incluso pasa con los objetos. Por ejemplo, si en Java defino la clase "Largometraje" (una cinta que se puede exhibir en la televisión o el cine), cuando creo objetos de la clase "Largometraje" debo declarar variables en las que indique el tipo de objeto que va a contener.

```
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. O sea, mañana no podremos guardar un entero en la variable, ni una cadena u otro objeto de otra 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, como decimos, 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
```

Fíjate que en este punto no te estoy hablando todavía de polimorfismo, sino de algo de la programación en general como es el sistema de tipos. Sin embargo, tienes que amoldar la cabeza a esta restricción de lenguajes fuertemente tipados para que luego puedas entender por qué el polimorfismo es importante y clave en la programación orientada a objetos. Y ojo, insisto que esto es algo relacionado con lenguajes fuertemente tipados (también llamados de tipado estático), en PHP no habría problema en cambiar el tipo de una variable, asignando cualquier otra cosa, dado que no se declaran los tipos al crear las variables.

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

Nota: A veces, a los lenguajes fuertemente tipados se les llama de "tipado estático" y a los débilmente tipados se les llama "tipado dinámico". Si quieres saber más sobre lenguajes tipados y no tipados, te recomiendo ver el #programadorIO tipados Vs no tipados.

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 alertará. Osea, si intentas invocar sumaDosNumeros("algo", "otro"), el compilador no te dejará compilar el programa porque no ha encontrado los tipos esperados en los parámetros de la función.

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. Como ves, en estos lenguajes como Java el tipado se lleva a todas partes.

Polimorfismo en objetos

Ahora párate a pensar en clases y objetos. Quédate con esto: 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, sólo 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 he 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 podrás contener en los elementos del array no solo vehículos genéricos, sino también todos los objetos de clases hijas o derivadas de la clase "Vehiculo", osea 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. Puedes, no obstante, tener varios tipos de largometrajes, como películas o documentales, etc. Quizás las películas y documentales tienen diferentes características, distintos horarios de audiencia, distintos precios para los espectadores y por ello has decidido que tu 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". Si has entendido el sistema de tipos, y sin entrar todavía en polimorfismo, debido a que los métodos declaran los tipos de los parámetros que recibes, tendrás 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? De acuerdo, igual no te supone tanto problema, ¿pero si mañana te mandan otro tipo de cinta a reproducir, como la grabación de la final del mundial de fútbol en 3D? ¿Tendrás que crear un nuevo método reproducir() sobre la clase "Cine" que te acepte ese tipo de emisión? ¿es posible ahorrarnos todo ese mantenimiento?

Aquí es donde el polimorfismo nos ayuda. Podrías crear perfectamente un método "reproducir()" que recibe un largometraje y donde podrás 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 lo bonito de la historia es que todos estos objetos son aceptados por el método "reproducir()", gracias a la relajación del sistema de tipos. Incluso, si mañana quieres reproducir otro tipo de cinta, no tendrás que tocar la clase "Cine" y el método "reproducir()". Siempre que aquello que quieras reproducir sea de la clase "Largometraje" o una clase hija, el método te lo aceptará.

Pongamos otro ejemplo por si acaso no ha quedado claro con lo visto hasta el momento, 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 me gustaría que se entienda.

Tenemos la clase Parking. Dentro de ésta 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 todos estaremos de acuerdo que estacionar un coche, una moto o un bus es bastante similar: "entrar en el parking, recoger el ticket de entrara, buscar una plaza, situar el vehículo dentro de esa plaza...".

Lo ideal sería que nuestro método me permita 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 una van, todoterreno hibrido, o una nave espacial, mi 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, osea, coches, motos, buses, etc. Esa relajación del sistema de tipos para aceptar una gama de objetos diferente es lo que llamamos polimorfismo.

En fin, esto es lo que significa polimorfismo. A partir de aquí puede haber otra serie de consideraciones y recomendaciones, así como características implementadas en otros lenguajes, pero explicar todo eso no es el objetivo de este artículo. Esperamos que con lo que has aprendido puedas orientar mejor tus estudios de Programación Orientada a Objetos. Si quieres más información sobre el tema lee el artículo Qué es Programación Orientada a Objetos, que seguro te será de gran utilidad.

Abstracción en Programación Orientada a Objetos

Concepto de abstracción en el paradigma de la Programación Orientada a Objetos y situaciones en las que se puede y se debe aplicar.

Abstracción es un término del mundo real que podemos aplicar tal cual lo entendemos en el mundo de la Programación Orientada a Objetos. Algo abstracto es algo que está en el universo de las ideas, los pensamientos, pero que no se puede concretar en algo material, que se pueda tocar.

Pues bien, una clase abstracta es aquella sobre la que no podemos crear especímenes concretos, en la jerga de POO es aquella sobre la que no podemos instanciar objetos. Ahora bien, ¿cuál es el motivo de la existencia de

clases abstractas? o dicho de otra manera, ¿por qué voy a necesitar alguna vez declarar una clase como abstracta?, ¿en qué casos debemos aplicarlas? Esto es todo lo que pretendemos explicar en este artículo.

Abstracción en el mundo real

La programación orientada a objetos sabemos que, de alguna manera, trata de "modelizar" los elementos del mundo real. En el mundo en el que vivimos existe un universo de objetos que colaboran entre sí para realizar tareas de los sistemas. Llevado al entorno de la programación, también debemos programar una serie de clases a partir de las cuales se puedan instanciar objetos que colaboran entre sí para la resolución de problemas. Si asumimos esto, a la vista de las situaciones que ocurren en el mundo real, podremos entender la abstracción.

Cuando estudiamos en el concepto de Herencia en Programación Orientada a Objetos vimos que con ella se podían definir jerarquías de clasificación: los animales y dependiendo de éstos tenemos mamíferos, vertebrados, invertebrados. Dentro de los mamíferos tenemos vacas, perros…

Animal puede ser desde una hormiga a un delfín o un humano. En nuestro cerebro el concepto de animal es algo genérico que abarca a todos los animales: "seres vivos de un "reino" de la existencia". Si defines animal tienes que usar palabras muy genéricas, que abarquen a todos los animales posibles que puedan existir en el mundo. Por ello no puedes decir que animales son aquellos que nacen de huevos, o después de un periodo de gestación en la placenta.

Adonde quiero llegar es que el animal te implica una abstracción de ciertos aspectos. Si lo definimos con palabras no podemos llegar a mucho detalle, porque hay muchos animales distintos con características muy diferentes. Hay características que se quedan en el aire y no se pueden definir por completo cuando pensamos en el concepto de animal "genérico".

Para acabar ¿en el mundo real hay un "animal" como tal? No, ni tan siquiera hay un "mamífero". Lo que tenemos son especímenes de "perro" o "vaca", "hormiga", "cocodrilo", "gorrión" pero no "animal" en plan general. Es cierto que un perro es un animal, pero el concepto final, el ejemplar, es de perro y no animal.

Por tanto "animal", en términos del lenguaje común, podemos decir que es un concepto genérico, pero no una concreción. En términos de POO decimos que es un concepto abstracto, que implementaremos por medio de una clase abstracta. No instanciaremos animales como tal en el mundo, sino que instanciaremos especímenes de un tipo de animal concreto.

En los animales existen propiedades y métodos que pueden ser comunes a todos los animales en general. Los animales podrán tener un nombre o una edad, determinadas dimensiones o podrán desempeñar acciones como morir. Lo que nos debe quedar claro es que no deberíamos poder instanciar un animal como tal. ¿Cómo nace un animal en concreto?, ¿cómo se alimenta? Para responder a esas preguntas necesitamos tener especímenes más concretos. Sí que sé cómo nace o cómo se alimenta una hormiga, o un gorrión, pero no lo puedo saber de un animal genérico, porque puede hacerlo de muchas maneras distintas.

Seguiremos trabajando para explicar estos conceptos, pero de momento entendemos que "animal" es una clase abstracta, pero "hormiga", "perro" o "gorrión" no serían clases abstractas, que sí podríamos instanciar.

Herencia y abstracción

Si entendemos el concepto de herencia podremos entender mejor la abstracción y cómo se implementa.

Recuerda nuestro ejemplo: Tengo animales. Hemos acordado que no puedo tener un animal concreto instanciado en un sistema. Si acaso tendré instancias de perros, saltamontes o lagartijas. Pues bien, en los esquemas de herencia este caso nos puede surgir muy habitualmente.

En la clase "animal" puedo tener determinadas propiedades y acciones implementadas. Por ejemplo, todos los animales pueden tener un nombre, o una edad (ya sean segundos, días o años de edad). También es posible que pueda definir diversas acciones de una vez para todos los animales de una jerarquía de herencia, por ejemplo, la acción de morir, pues todos morimos igual (simplemente dejamos de existir aunque aquí dependiendo de las creencias de cada uno esto pueda ser discutible).

Aunque mi sistema no pueda crear animales como tal, tener definidas esas cuestiones comunes a todos los animales me resulta útil para no tener que programarlas de nuevo en todos los tipos de animales que puedan existir. Simplemente las heredaré en las clases hijas, de modo que estarán presentes sin tener que volver a programar todas esas cosas comunes.

Sin embargo hay cosas de los animales que no podré implementar todavía. Atributos como el número de patas, el alcance de la visión, se implementarán a futuro en los tipos de animales que las necesiten, pero fijémonos en las acciones o métodos. Por ejemplo nacer, alimentarse, etc. No sé cómo va a nacer un animal, pero sé que todos los animales del mundo nacen de algún modo (unos nacen de huevos, otros estaban en la barriga de las hembras y nacen a consecuencia de un parto, etc.)

En estos casos nos puede ser útil definir como métodos abstractos en la clase "animal" esos métodos que van a estar presentes en todos los animales, aunque no seamos capaces de implementarlos todavía.

```
public abstract function nacer();
```

Esto quiere decir que todos los animales del mundo heredarán un método abstracto llamado nacer. En las clases concretas que hereden de animal y donde ya sepamos cómo nace tal animal, por ejemplo, la gallina, podemos implementar ese método, para que deje de ser abstracto.

```
public function nacer(){
    //se rompe el huevo y nace el pollito que más adelante será una hermosa gallina
}
```

Hasta ahora sabemos que hay clases que tienen métodos abstractos, que no somos capaces de implementar todavía y clases en las que se heredan métodos abstractos y en las que seremos capaces de implementarlos.
La utilidad de esto la entenderemos mejor en unos instantes, al tratar el polimorfismo, pero de momento debemos ser capaces de asimilar estas definiciones más formales:

"Una clase abstracta es aquella en la que hay definidos métodos abstractos, sobre la que no podremos instanciar objetos" Además, en un esquema de herencia, **"Si heredamos de una clase abstracta métodos abstractos, tampoco se podrán instanciar objetos de las clases hijas y tendrán que definirse como abstractas, a no ser que implementemos todos y cada uno de los métodos que se habían declarado como abstractos en la clase padre"**.

Polimorfismo y abstracción

Creo que si no se examina de cerca la abstracción bajo el prisma del polimorfismo no se puede entender bien la verdadera utilidad de hacer clases abstractas. Pero antes de seguir, recuerda qué es el Polimorfismo en Programación Orientada a Objetos.

Cuando hablamos de polimorfismo explicamos que es una relajación del sistema de tipos por la cual éramos capaces de aceptar objetos de un tipo y de todas las clases hijas. Por ejemplo, tengo la clase "PoligonoRegular". Sé que los polígonos regulares voy a querer conocer su área, pero para saber su área necesito conocer el número de lados que tiene. Entonces la clase "PoligonoRegular" tendrá un método abstracto "dameArea()". Luego, al definir la clase "cuadrado", o el "pentágono", etc. podremos implementar tal método, con lo que dejará de ser abstracto. Tenemos "Alumnos de una Universidad", los alumnos los vas a querer matricular en las universidades, pero dependiendo del tipo de alumno la matrícula se hace diferente, pues no es lo mismo matricular un alumno becario, o de familia numerosa, que uno normal. Entonces, en la clase "alumno" tendré un método abstracto que sea "matriculate()" que podré definir del todo cuando implemente las clases hijas.

Ahora piensa en esto. Gracias a que fueron definidos los métodos abstractos "dameArea()" y "matriculate()" en las clases padres, tengo clara una cosa: cuando trabajo con elementos de la clase "poligonoRegular", sé que a todos los polígonos regulares que pueda recibir les puedo pedir que me devuelvan su área. También sé que a todos los alumnos les puedo pedir que se matriculen en una universidad.

Ahí está la potencia del polimorfismo, recibir un objeto que pertenece a una jerarquía de clasificación y saber que puedo pedirle determinadas cosas. Quizás en la clase padre no pudieron implementarse esos comportamientos, porque no sabíamos el código necesario para ello, pero al menos se declararon que iban a poder realizarse en el futuro en clases hijas. Eso me permite, en un esquema de polimorfismo, que pueda estar seguro que todos los objetos que reciba puedan responder a acciones determinadas, pues en las clases hijas habrán sido definidas necesariamente (si no se definen deberían declararse las clases como abstractas y en ese caso es imposible que me manden objetos de esa clase).

Es cierto que el concepto se puede quedar un poco en "la abstracción" pero cuando practiques un poco te darás cuenta de la esencia de las clases abstractas y entenderás lo útil y necesario que es declarar métodos abstractos para poder implementar el polimorfismo. De momento es todo en cuanto a este concepto, esperamos que este texto te haya servido de algo. Por supuesto, se agradecen los comentarios.

Métodos y atributos static en Programación Orientada a Objetos

Definición y ejemplos de elementos o miembros static, los métodos y atributos de clase o estáticos en la programación orientada a objetos.

En el Manual de la Teoría de la programación orientada a objetos estamos revisando diversos conceptos y prácticas comúnmente usadas en este paradigma de la programación. Ahora le toca el turno a "static".

Seguramente lo has visto en alguna ocasión en algún código, pues se trata de una práctica bastante común. Con "static", un modificador que podemos aplicar en la definición de métodos y atributos de las clases, podemos definir estos elementos como pertenecientes a la clase, en lugar de pertenecientes a la instancia. A lo largo de este artículo queremos explicar bien qué es esto de los elementos estáticos "static" o también llamados elementos "de clase".

Qué es static

La definición formal de los elementos estáticos (o miembros de clase) nos dice que son aquellos que pertenecen a la clase, en lugar de pertenecer a un objeto en particular. Recuperando concetos básicos de orientación a objetos, sabemos que tenemos:

Clases: definiciones de elementos de un tipo homogéneo.

Objetos: concreción de un ejemplar de una clase.

En las clases defines que tal objeto tendrá tales atributos y tales métodos, sin embargo, para acceder a ellos o darles valores necesitas construir objetos de esa clase. Una casa tendrá un número de puertas para entrar, en la clase tendrás definida que una de las características de la casa es el número de puertas, pero solo concretarás ese número cuando construyas objetos de la clase casa. Un coche tiene un color, pero en la clase solo dices que existirá un color y hasta que no construyas coches no les asignarás un color en concreto. En la clase cuadrado definirás que el cálculo del área es el "lado elevado a dos", pero para calcular el área de un cuadrado necesitas tener un objeto de esa clase y pedirle que te devuelva su área.

Ese es el comportamiento normal de los miembros de clase. Sin embargo, los elementos estáticos o miembros de clase son un poco distintos. Son elementos que existen dentro de la propia clase y para acceder los cuales no necesitamos haber creado ningún objeto de esa clase. O sea, en vez de acceder a través de un objeto, accedemos a través del nombre de la clase.

Ejemplos de situaciones de la vida real donde tendríamos miembros estáticos

De momento así dicho queda un tanto abstracto. Pero antes de ponerse con ejemplos concretos de programación donde hablemos de la utilidad práctica de los miembros estáticos sea bueno tratar de explicar estos conceptos un con situaciones de la vida real.

Por ejemplo, pensemos en los autobuses de tu ciudad. No sé si es el caso, pero generalmente en España todos los autobuses metropolitanos tienen la misma tarifa. Yo podría definir como un atributo de la clase AutobúsMetropolitano su precio. En condiciones normales, para acceder al precio de un autobús necesitaría instanciar un objeto autobús y luego consultar su precio. ¿Es esto práctico? quizás solo quiero saber su precio para salir de casa con dinero suficiente para pagarlo, pero en el caso de un atributo normal necesariamente debería tener instanciado un autobús para preguntar su precio.

Pensemos en el número "Pi". Sabemos que necesitamos ese número para realizar cálculos con circunferencias. Podría tener la clase Circunferencia y definir como atributo el número Pi. Sin embargo, igual necesito ese número para otra cosa, como pasar ángulos de valores de grados a radianes. En ese caso, en condiciones

normales sin atributos de clase, necesitaría instanciar cualquier círculo para luego preguntarle por el valor de "Pi". De nuevo, no parece muy práctico.

Nota: Ojo, porque en el caso del número Pi, su valor será siempre constante. Podríamos en esa caso usar constantes si nuestro lenguaje las tiene, pero los atributos estáticos no tienen por qué ser siempre un valor invariable, como es el caso del precio de los AutobusesMetropolitanos, que sube cada año.

Con esos tenemos dos ejemplos de situaciones en las que me pueden venir bien tener atributos "static", o de clase, porque me permitirían consultar esos datos sin necesidad de tener una instancia de un objeto de esa clase.

En cuanto a métodos, pensemos por ejemplo en la clase Fecha. Puedo intentar construir fechas con un día, un mes y un año, pero puede que no necesite una fecha en un momento dado y solo quiera saber si una fecha podría ser válida. En situaciones normales debería intentar construir esa fecha y esperar a ver si el constructor me arroja un error o si la fecha que construye es válida. Quizás sería más cómodo tener un método vinculado a la clase, en el que podría pasarle un mes, un día y un año y que me diga si son válidos o no.

Cómo definir y usar miembros estáticos

Esto ya depende de tu lenguaje de programación. Generalmente se usa el modificador "static" para definir los miembros de clase, al menos así se hace en Java o PHP, C#, pero podrian existir otros métodos en otros lenguajes.

Nota: El código que verás a continuación no es de uno u otro lenguaje. No es mi objetivo explicar Java, PHP, C#, etc. sino ver en pseudo-codigo rápidamente una posible implementación de el uso de static. Es parecido a Java, pero le he quitado algunas cosas que dificultan el entendimiento de los ejemplos. Tampoco es PHP, porque no he metido los $ de las variables y el uso del acceso a variables estáticas se hace con el operador :: que me parece muy feo y que complica la lectura y comprensión de los ejemplos. Tendrás que consultar la documentación de tu propio lenguaje para ver cómo se hace en él, aquí lo que quiero que queden claros son los conceptos nada más.

```
class AutobusMetropolitano {
        public static precio = 1.20;
}
```

Podrás observar que en la definición de la clase hemos asignado un valor al precio a la hora de declarar ese atributo de clase. Esto es condición indispensable en muchos lenguajes, puesto que si existe ese atributo debería tener un valor antes de instanciar el primer objeto. O sea, no es como los atributos normales, que les podemos asignar valores a través del constructor.

Luego podríamos acceder a esos elementos a través del nombre de la clase, como se puede ver a continuación:

```
if(1.5 >= AutobusMetropolitado.precio){
        //todo bien
}
```

Ese código sería en el caso que estés accediendo al atributo estático desde fuera de la clase, pero en muchos lenguajes puedes acceder a esos atributos (o métodos) desde la vista privada de tus clases (código de implantación de la clase) a través de la palabra "self".

```
class AutobusMetropolitano {
        static precio = 1.20;
        //...

        public function aceptarPago(dinero){
                if (dinero < self.precio){
                        return false;
                }
                // ...
        }
}
```

En el caso de los métodos estáticos la forma de crearlos o usarlos no varía mucho, utilizando el mismo modificador "static" al declarar el método.

```
class Fecha{
        //...
public static function valida(ano, mes, dia){
                if(dia >31)
                        return false;
                // ...
        }
        //...
}
```

Ahora para saber si un conjunto de año mes y día es válido podrías invocar al método estático a través del nombre de la clase.

```
if(Fecha.valida(2014, 2, 29)){
        // es valida
}else{
        // no es valida
}
```

Como en el caso de los atributos de clase, también podrías acceder a métodos de clase con la palabra "self" si estás dentro del código de tu clase.

Miembros estáticos son susceptibles de hacer "marranadas"

No quiero dejar este artículo sin aprovechar la oportunidad de explicar que los miembros estáticos deben estar sujetos a un cuidado adicional para no cometer errores de diseño.

En ocasiones se tiende a usar los métodos estáticos para agregar datos a modo de variables globales, que estén presentes en cualquier lugar de la aplicación a partir del nombre de una clase. Esto nos puede acarrear exactamente los mismos problemas que conocemos por el uso de variables globales en general.

En cuanto a métodos estáticos a veces se usan las clases como simples contenedores de unión de diferentes métodos estáticos. Por ejemplo la clase Math en Javascript, o incluso Java, que tiene simplemente una serie de métodos de clase. Usarlos así, creando métodos estáticos en la clase, sin agregar más valor a la clase a través de métodos o atributos "normales" es caer en prácticas similares a las que se vienen usando en la programación estructurada. Dicho de otra manera, es usar la Programación Orientada a Objetos sin aprovechar los beneficios que nos aporta.