Programación orientada a objetos con PHP

La Programación Orientada a Objetos es una forma de programar relativamente reciente (en PHP existe desde su versión 5), si la comparamos con la forma convencional de programar que hemos visto hasta ahora en los apartados previos, llamada programación estructurada y modular, que utiliza simplemente estructuras de control de flujo (*if*, *while*, *for*) y funciones para estructurar y dividir el código de un programa.

Mediante la Programación Orientada a Objetos, se pretende identificar cada uno de los tipos de objeto que componen un programa. A cada uno de esos tipos se le llama **clase**, y está compuesto por una serie de propiedades o **atributos**, y una serie de operaciones que puede realizar (**métodos**). Cada variable que creemos de ese tipo será un **objeto** de esa clase, con sus propios valores para sus propiedades o atributos.

1. Ejemplo sencillo

Veámoslo con un ejemplo. Imaginemos que tenemos una base de datos donde queremos almacenar el software que tenemos en casa. De cada software queremos almacenar un código identificativo, su título y su versión. También queremos que, para cualquier software, podamos mostrar sus datos por pantalla.

Para representar cualquier software que queramos gestionar, podemos crearnos una clase llamada `Software`, y definirle los atributos que va a tener (en este caso, su código, título y versión) y las operaciones que va a poder realizar, que definiremos en forma de funciones (por ejemplo, mostrar sus datos). Para hacer esto en PHP, usaremos la palabra `class` con el nombre de la clase que queramos (en este caso, *Software*), y dentro definimos sus atributos (con algún valor predeterminado, si queremos) y sus funciones o métodos.

```
class Software
{
    private $codigo = 0;
    private $titulo = "";
    private $version = "1";

    public function MostrarDatos()
    {
        echo "<p>$this->titulo ($this->version)</p>";
    }
}
```

Una diferencia importante entre los atributos de la clase y las variables normales es que, para poderlos referenciar en cualquier método de la clase, tenemos que anteponerle el objeto `$this`, seguido del operador flecha `->` y el nombre del atributo (sin el dólar). Esto es así para evitar problemas en el caso de que alguna otra variable en la función se llame igual que el atributo.

A diferencia del resto de código PHP, no podemos interrumpir la definición de una clase para intercalar código HTML en medio, toda la clase debe estar comprendida entre el mismo par de etiquetas `<?php` y `?>`. Las palabras `private` y `public` las explicaremos más adelante.

2. Creación de objetos

Ya tenemos la clase creada. ¿Cómo usamos variables de tipo *Software*? Para poder crear variables u objetos de cualquier clase, necesitamos definir una función especial llamada **constructor**. Estas funciones pueden recibir una serie de parámetros, que normalmente son los valores que queremos asignarles a los distintos atributos. Así, por ejemplo, para nuestra clase anterior, podemos definir un constructor que reciba tres parámetros (uno para el código, otro para el título y otro para la versión) y los asigne a los correspondientes atributos:

```
class Software
{
    private $codigo = 0;
    private $titulo = "";
    private $version = "1";

    public function __construct($c, $t, $v)
    {
        $this->codigo = $c;
        $this->titulo = $t;
        $this->version = $v;
    }

    ...
}
```

Con esto, ya podemos crear variables de este tipo, usando el operador `new` para crear cada objeto. Por ejemplo, así crearíamos dos variables de tipo Software, `$s1` y `$s2`, con diferentes datos cada una (desde fuera de la clase, si queremos):

```
$s1 = new Software(1, "LibreOffice", "6.0");
$s2 = new Software(2, "GIMP", "3.8");
```

Y para llamar a las funciones de la clase y poder, por ejemplo, mostrar los datos de cada software por pantalla, haríamos algo como esto.

```
echo "<p>Datos del primer programa:</p>";
$s1->MostrarDatos();
echo "<p>Datos del segundo programa:</p>";
$s2->MostrarDatos();
```

Observa que usamos el operador `->` para llamar a la función de un objeto. Ya hemos visto algo parecido anteriormente para acceder al error de una excepción.

3. Modificadores de acceso

Cuando hemos creado la clase *Software*, delante de los atributos o propiedades de la clase hemos puesto la palabra *private*, que es un modificador de acceso que quiere decir que no se puede acceder a esos atributos desde fuera de la clase (para preservar que su valor no se modifique sin control). En cambio, las funciones tienen el modificador *public*, para poderlas llamar desde fuera.

En general, podemos utilizar tres modificadores diferentes en las clases:

- **private**: el elemento no puede ser visto desde fuera de la clase
- **public**: el elemento puede verse y utilizarse fuera de la clase. Es el modificador por defecto si no indicamos uno nosotros
- **protected**: para herencia, el elemento es visible desde la propia clase o las clases heredadas; veremos el tema de la herencia a continuación.

3.1. Getters y setters

Hemos dicho que los atributos de una clase normalmente son privados. Esto es así para no poder acceder a ellos directamente desde fuera, y cambiar su valor erróneamente. Por ejemplo, en el caso anterior, si los atributos fueran públicos, alguien podría intentar acceder al código de un programa y darle un valor negativo:

```
$s1 = new Software(...);
...
$s1->codigo = -1;
```

Para evitar esto, se suelen dejar privados, y para acceder a su valor o modificarlo, se añaden unas funciones o métodos especiales, llamadas comúnmente **getters y setters**. Los primeros (*getters*) se llaman así porque el nombre de la función suele empezar por `__get` y sirven para obtener el valor del atributo al que representan. Los segundos (*setters*) se llaman así porque la función suele empezar por `__set` y se emplean para modificar el valor del atributo. Dentro de esta segunda función, podemos hacer alguna comprobación previa antes de cambiar el valor del atributo (por ejemplo, en el caso del código, comprobar que no tenga un valor negativo).

```php
class Software
{
    private $codigo;
    private $titulo;
    private $version;
    ...

    public function __get($nombre)
    {
        if ($nombre == 'Cod')
            return $this->codigo;
        else if ($nombre == 'Titulo')
            return $this->titulo;
        else if ($nombre == 'Version')
            return $this->version;
    }

    public function __set($nombre, $valor)
    {
        if ($nombre == 'Cod' && $valor > 0)
            $this->codigo = $valor;
        else if ($nombre == 'Titulo')
            $this->titulo = $valor;
        else if ($nombre == 'Version')
            $this->version = $valor;
    }
    ...
}
```

Observa que el método `__get` recibe como parámetro un nombre (el que queramos), y luego dentro de la función emparejamos cada nombre con el atributo al que queramos enlazarlo. Si ponemos *Cod*, lo asociaremos al atributo `$codigo`, y así sucesivamente.

Con esto, si queremos cambiar el código del objeto anterior, podríamos hacerlo con su setter correspondiente, y asegurarnos de que se cambiará a un valor correcto.

```php
$s1->Cod = -1; // Esto no hará nada
$s1->Cod = 10; // Esto sí funcionará
```

La llamada al *setter* o al *getter* no es como en el resto de funciones. Tenemos que poner el nombre del objeto, el operador `->` y un nombre (de entre los que vaya a aceptar el *getter* o *setter*), y en el caso de querer asignarle un valor, se lo asignamos como si fuera una variable normal. Ese nombre, como podemos apreciar, no tiene por qué coincidir con el nombre del atributo. Dentro de la función nos encargamos de emparejar cada nombre con el atributo, así que pueden ser diferentes, ya que la asociación la hacemos nosotros a mano.

4. Herencia

La herencia es una de las herramientas más potentes de la programación orientada a objetos. Consiste en definir una clase partiendo de otra, y suponiendo que la nueva clase es un subtipo de la anterior. Por ejemplo, si tuviéramos una clase *Animal* con una serie de atributos comunes de cualquier animal (por ejemplo, color y número de patas), podríamos definir una clase *Perro* que heredara de *Animal*, con lo que ya tendría implícitamente todo lo que tuviera *Animal* (en este caso, los atributos del color y número de patas), y además, podríamos añadir a esta nueva clase las características propias de un perro (por ejemplo, el tipo de ladrido).

Volviendo a nuestro ejemplo del *Software*, imaginemos que, entre el software que tenemos en casa, tenemos varios videojuegos, y que de ellos nos interesa guardar, además del título y la versión, la plataforma para la que están hechos (PC, consola, etc.). En este caso, como los videojuegos son un subtipo de software, podemos aplicar herencia, y crear una nueva clase `Videojuego` que herede de *Software*, así:

```
class Videojuego extends Software
{
    private $plataforma;

    public function __construct($c, $t, $v, $p)
    {
        $this->codigo = $c;
        $this->titulo = $t;
        $this->version = $v;
        $this->plataforma = $p;
    }

    ...
}
```

A la clase heredada (en este caso, *Software*), se le suele llamar clase base, superclase o clase padre. A la clase que hereda, se le llama subclase, clase derivada o clase hija.

Así, la clase *Videojuego* heredará todo lo que tenemos hecho en *Software*. Recuerda que los modificadores *private* impiden que el elemento que lo tiene sea visible desde fuera. Eso quiere decir que, ahora mismo, en *Videojuego*, no podríamos acceder directamente a los atributos `$codigo`, `$titulo` o `$version`, pero sí podemos acceder a los *getters* y *setters* de *Software* para darles valor o consultarlo. De todas formas, si quisiéramos acceder directamente a esos atributos, sin pasar por los *getters* y *setters*, tendríamos que definirlos como *protected* en la clase *Software*.

```
class Software
{
    protected $codigo;
    protected $titulo;
    protected $version;

    ...
}
```

Así, el constructor que hemos definido antes en *Videojuego* sí funcionaría (antes no, porque los atributos eran private).

4.1. El elemento parent

Hemos visto que el objeto `$this` nos sirve para referenciar a los atributos o elementos de una clase, y poderlos distinguir de otros externos que se llamen igual. Del mismo modo, podemos referenciar a los atributos o métodos de la clase padre mediante el objeto `parent`, con la sintaxis `parent::metodo(...)`. Así, por ejemplo, en el caso anterior, no sería necesario duplicar el código en el constructor de la clase hija para los atributos que asigna la clase padre. Podríamos poner:

```
public function __construct($c, $t, $v, $p)
{
    parent::__construct($c, $t, $v);
    $this->plataforma = $p;
}
```

> **Ejercicio 1:**
>
> Crea una carpeta **ejercicios7** para los ejercicios de esta sesión. Dentro crea una página llamada **clases.php** con una clase llamada **Persona** que tenga como atributos un DNI, un nombre y un email. Crea un constructor que permita rellenar esos tres atributos, y los *getters* y *setters* correspondientes. Define también un método *Mostrar* para sacar por la página los datos de la persona (en un párrafo, separados por guiones). Define adecuadamente la visibilidad (pública o privada) de cada atributo o método.
>
> Crea una segunda clase llamada **Estudiante** que herede de *Persona*, y añada un atributo llamado *numExpediente*. Crea su constructor, sus *getters* y *setters* y su correspondiente método *Mostrar*.
>
> Fuera de las clases, entre el código HTML de la página, crea un objeto de cada tipo (una *Persona* y un *Estudiante*), con los valores que quieras, llama después a algún setter de cada una para cambiar el valor de algún atributo, y finalmente llama a sus métodos *Mostrar* para que saquen la información de cada uno.