

Resumen del tema 5

Desarrollo Web en Entorno Servidor

11 de enero de 2023

CIFP Carlos III - Cartagena

Santiago Francisco San Pablo Raposo

2º curso DAW

Contenido

# Índice de ilustraciones.

# Índice de tablas.

Resumen del tema 5. Programación orientada a objetos en PHP.

# 1.- Características de orientación a objetos en PHP (I).

La programación orientada a objetos (POO, o OOP en lenguaje inglés), es una metodología de programación basada en objetos. Un objeto es una estructura que contiene datos y el código que los maneja.

La estructura de los objetos se define en las clases. Entre los miembros de una clase puede haber:

* **Métodos**: son los que contienen el código de una clase. Un método es como una función. Puede recibir parámetros y devolver valores.
* **Atributos o propiedades**: almacenan información acerca del estado del objeto al que pertenecen. Su valor puede ser distinto para cada uno de los objetos de la misma clase.

A la creación de un objeto basado en una clase se le llama instanciar una clase y al objeto obtenido también se le conoce como **instancia de esa clase**.

Pilares fundamentales de la POO:

* **Herencia**: Es el proceso de crear una clase a partir de otra, heredando su comportamiento y características.
* **Abstracción**: cada clase oculta en su interior las peculiaridades de su implementación, y presenta al exterior una serie de métodos (interface) cuyo comportamiento está bien definido.
* **Polimorfismo**: Un mismo método puede tener comportamientos distintos en función del objeto con que se utilice.
* **Encapsulación**: En la POO se juntan en un mismo lugar los datos y el código que los manipula.

**Ventajas más importantes que aporta la programación orientada a objetos son**:

* **Modularidad**: La POO permite dividir los programas en partes o módulos más pequeños, que son independientes unos de otros pero pueden comunicarse entre ellos.
* **Extensibilidad**: Si se desean añadir nuevas características a una aplicación, la POO lo facilita de dos formas: añadiendo nuevos métodos al código, o creando nuevos objetos que extiendan el comportamiento de los ya existentes.
* **Mantenimiento**: Los programas desarrollados utilizando POO son más sencillos de mantener, debido a la modularidad antes comentada. También ayuda seguir ciertas convenciones al escribirlos, por ejemplo, escribir cada clase en un fichero propio.

**Para saber más**: [repasar sobre los conceptos de orientación a objetos](https://desarrolloweb.com/articulos/499.php).

## 1.1.- Características de orientación a objetos en PHP (II).

Vamos a ver las peculiaridades propias de PHP en lo que respecta a la POO.

Como ya has visto en las unidades anteriores, especialmente con las extensiones para utilizar BB.DD, con PHP puedes utilizar dos estilos de programación: estructurada y orientada a objetos.

// utilizando programación estructurada

$dwes = mysqli\_connect('localhost', 'dwes', 'abc123.', 'dwes');

// utilizando POO

$dwes = new mysqli();

$dwes->connect('localhost', 'dwes', 'abc123.', 'dwes');

Sin embargo, el lenguaje PHP original no se diseñó con características de orientación a objetos. Solo a partir de la versión 3, se empezaron a introducir algunos rasgos de POO en el lenguaje. Esto se potenció en la versión 4, aunque todavía de forma muy rudimentaria.

Por ejemplo, **en PHP4**:

* **Los objetos se pasan siempre por valor**, no por referencia.
* **No se puede definir el nivel de acceso para los miembros de la clase. Todos son públicos**.
* **No existen los interfaces**.
* No existen métodos destructores.

En la versión actual, PHP5, se ha reescrito y potenciado el soporte de orientación a objetos del lenguaje. Las características de POO que soporta PHP5 son:

* Métodos estáticos.
* Métodos constructores y destructores.
* Herencia.
* Interfaces.
* Clases abstractas.

Entre las características que no incluye PHP5, y que puedes conocer de otros lenguajes de programación, están:

* Herencia múltiple.
* [Sobrecarga de métodos](https://ead.murciaeduca.es/mod/glossary/showentry.php?displayformat=dictionary&concept=Sobrecarga%20de%20m%C3%A9todos%20%28DAW_DWES05%29) (incluidos los métodos constructores).
* [Sobrecarga de operadores](https://ead.murciaeduca.es/mod/glossary/showentry.php?displayformat=dictionary&concept=Sobrecarga%20de%20operadores%20%28DAW_DWES05%29).

## 1.2.- Creación de clases (I).

La declaración de una clase en PHP se hace utilizando la palabra **class**.

class Producto {

private $codigo;

public $nombre;

public $PVP;

public function muestra() {

print "<p>" . $this->codigo . "</p>";

}

}

Además, muchos programadores prefieren utilizar para las clases nombres que comiencen por letra mayúscula, para, de esta forma, distinguirlos de los objetos y otras variables.

Una vez definida la clase, podemos usar la palabra **new** para instanciar objetos de la siguiente forma:

$p = new Producto();

Para que la línea anterior se ejecute sin error, previamente debemos haber declarado la clase. Para ello, en ese mismo fichero tendrás que incluir la clase poniendo algo como:

require\_once('producto.php');

Para acceder desde un objeto a sus atributos o a los métodos de la clase, debes utilizar el **operador flecha** (fíjate que sólo se pone el símbolo **$** delante del nombre del objeto):

$p->nombre = 'Samsung Galaxy S';

$p->muestra();

Cuando se declara un atributo, se debe indicar su nivel de acceso. **Los principales niveles son**:

* **Public**: pueden utilizarse directamente por los objetos de la clase.
* **Private**: solo pueden ser accedidos y modificados por los métodos definidos en la clase, no directamente por los objetos de la misma.

En PHP4 no se podía definir nivel de acceso para los atributos de una clase, por lo que todos se precedían de la palabra var.

Hoy en día, aunque aún es aceptado por PHP5, no se recomienda su uso. Si encuentras algún código que lo utilice, ten en cuenta que tiene el mismo efecto que **public**.

### 1.2.1.- Creación de clases (II).

**Uno de los motivos para crear atributos privados es que su valor forma parte de la información interna del objeto y no debe formar parte de su interface**. Otro motivo es mantener cierto control sobre sus posibles valores.

Por ejemplo, no quieres que se pueda cambiar libremente el valor del código de un producto. En estos casos, se suelen definir esos atributos como privados y además se crean dentro de la clase métodos para permitirnos obtener y/o modificar los valores de esos atributos. Por ejemplo:

private $codigo;

public function setCodigo($nuevo\_codigo) {

if (noExisteCodigo($nuevo\_codigo)) {

$this->codigo = $nuevo\_codigo;

return true;

}

return false;

}

public function getCodigo() { return $this->codigo; }

El nombre del método que nos permite obtener el valor de un atributo suele empezar por get, y el que nos permite modificarlo por set.

class Producto {

private $atributos = array();

public function \_\_get($atributo) {

return $this->atributos[$atributo];

}

public function \_\_set($atributo, $valor) {

$this->atributos[$atributo] = $valor;

}

}

En la documentación de PHP tienes más información sobre los métodos mágicos. [PHP: Métodos mágicos - Manual](https://www.php.net/manual/es/language.oop5.magic.php)

Texto

Descripción generada automáticamente

### 1.2.2.- Creación de clases (III).

Desde los métodos de las clases, **podemos hacer referencia al objeto o instancia de esa clase, utilizando la variable $this**. Se usa para tener acceso a los atributos del objeto. Por ejemplo:

print "<p>" . $this->codigo . "</p>";

**Debes conocer**: una referencia es una forma de utilizar distintos nombres de variables para acceder al mismo contenido. En los puntos siguientes aprenderás a crearlas y a utilizarlas: [PHP: Explicación de las Referencias - Manual](https://www.php.net/manual/es/language.references.php)

En una clase, también se pueden definir **constantes**, utilizando la palabra **const**. Es importante que no confundas los atributos con las constantes. Son conceptos distintos: las constantes:

* no pueden cambiar su valor no usan el carácter $.
* además, su valor va siempre entre comillas.
* está asociado a la clase, es decir, no existe una copia del mismo en cada objeto.

Por tanto, **para acceder a las constantes de una clase**, se debe utilizar el nombre de la clase y el operador **::**, llamado **operador de resolución de ámbito** (que se utiliza para acceder a los elementos de una clase).

class DB {

const USUARIO = 'dwes';

…

}

echo DB::USUARIO;

No es necesario que exista ningún objeto de una clase para poder acceder al valor de las constantes que defina. Además, sus nombres suelen escribirse en mayúsculas.

Tampoco se deben confundir las constantes con los miembros estáticos de una clase. En PHP5, **una clase puede tener atributos o métodos estáticos**, también llamados a veces atributos o métodos de clase. Se definen utilizando la palabra clave **static**, y se usan con la palabra **self::**.

class Producto {

private static $num\_productos = 0;

public static function nuevoProducto() {

self::$num\_productos++;

}

…

}

**Los atributos y métodos estáticos** no pueden ser llamados desde un objeto de la clase utilizando el operador ->. Si el método o atributo es público, **deberá accederse utilizando el nombre de la clase y el operador de resolución de ámbito**.

Producto::nuevoProducto();

Si es privado, como el atributo **$num\_productos** en el ejemplo anterior, sólo se podrá acceder a él desde los métodos de la propia clase, utilizando la palabra **self**. De la misma forma que **$this** hace referencia al objeto actual, **self** hace referencia a la clase actual.

self::$num\_productos ++;

Los **atributos estáticos** de una clase (se almacenan a nivel de clase) se utilizan para guardar información general sobre la misma, como puede ser el número de objetos que se han instanciado.

Los **métodos estáticos** suelen realizar alguna tarea específica o devolver un objeto concreto. **Por ejemplo**: las clases matemáticas suelen tener métodos estáticos para realizar logaritmos o raíces cuadradas. No tiene sentido crear un objeto si lo único que queremos es realizar una operación matemática.

Los métodos estáticos se llaman desde la clase. No es posible llamarlos desde un objeto y por tanto, no podemos usar **$this** dentro de un método estático

### 1.2.3.- Creación de clases (IV).

Repaso de todo un poco:

|  |  |
| --- | --- |
| Acción | Cómo se hace |
| Instanciar objetos de una clase | $p = new Producto(); |
| Definición de constructores | class Producto {  private static $num\_productos = 0;  private $codigo;    public function \_\_construct() {  self::$num\_productos++;  }  …  } |
| El constructor de una clase puede llamar a otros métodos o tener parámetros. |
| Sin embargo, solo puede haber un método constructor en cada clase (no admite sobrecarga).  class Producto {  private static $num\_productos = 0;  private $codigo;    public function \_\_construct($codigo) {  $this->$codigo = $codigo;  self::$num\_productos++;  }  …  }  $p = new Producto('GALAXYS'); |
| Métodos mágicos: \_\_call | Sirve para capturar llamadas a métodos que no estén implementados en la clase. |
| Entonces, en función del nombre del método y del número de parámetros que se pasen, se podrían realizar unas acciones u otras. |
| [PHP: Sobrecarga - Manual](https://www.php.net/manual/es/language.oop5.overloading.php#language.oop5.overloading.methods) |
| Definición de destructores | Permite definir acciones que se ejecutarán cuando se elimine el objeto. |
| class Producto {  private static $num\_productos = 0;  private $codigo;    public function \_\_construct($codigo) {  $this->$codigo = $codigo;  self::$num\_productos++;  }    public function \_\_destruct() {  self::$num\_productos--;  }  …  }  $p = new Producto('GALAXYS'); |
| Datos adicionales | Los **métodos constructores** también existen en PHP4, pero en lugar de llamarse **\_\_construct**, se deben llamar del mismo modo que la clase.  Los **métodos destructores** son nuevos en PHP5; no existían en versiones anteriores del lenguaje. |

Texto

Descripción generada automáticamente

## 1.3.- Utilización de objetos (I).

Ya sabemos como instanciar un objeto utilizando **new**, y cómo acceder a sus métodos y atributos públicos con el operador flecha.

Una vez creado un objeto, puedes utilizar el operador **instanceof** para comprobar si es o no una instancia de una clase determinada.

if ($p instanceof Producto) {

…

}

Además, en PHP5 se incluyen una serie de funciones útiles para el desarrollo de aplicaciones utilizando POO:

|  |  |  |
| --- | --- | --- |
| Función | Ejemplo | Signficado |
| get\_class | echo "La clase es: " . get\_class($p); | Devuelve el nombre de la clase del objeto. |
| class\_exists | if (class\_exists('Producto') {  $p = new Producto();  …  } | Devuelve **true** si la clase está definida o **false** en caso contrario. |
| get\_declared \_classes | print\_r(get\_declared\_classes()); | Devuelve un array con los nombres de las clases definidas. |
| class\_alias | class\_alias('Producto', 'Articulo');  $p = new Articulo(); | Crea un alias para una clase. |
| get\_class  \_methods | print\_r(get\_class\_methods('Producto')); | Devuelve un **array con los nombres de los métodos de una clase** que son accesibles desde dónde se hace la llamada. |
| method\_  exists | if (method\_exists('Producto', 'vende') {  …  } | Devuelve **true** si existe el método en el objeto o la clase que se indica, o **false** en caso contrario, independientemente de si es accesible o no. |
| get\_class  \_vars | print\_r(get\_class\_vars('Producto')); | Devuelve un array con los nombres de los atributos de una clase que son accesibles desde dónde se hace la llamada. |
| get\_object  \_vars | print\_r(get\_object\_vars($p)); | Devuelve un array con los nombres de los métodos de un objeto que son accesibles desde dónde se hace la llamada. |
| property\_  exists | if (property\_exists('Producto', 'codigo') {  …  } | Devuelve true si existe el atributo en el objeto o la clase que se indica, o false en caso contrario, independientemente de si es accesible o no. |

Desde PHP5, puedes indicar en las funciones y métodos de qué clase deben ser los objetos que se pasen como parámetros. Para ello, debes especificar el tipo antes del parámetro.

public function vendeProducto(Producto $p) {

…

}

**Si** cuando se realiza la llamada, **el parámetro no es del tipo adecuado**, **se produce un error que podrías capturar**.

Además, ten en cuenta que **sólo funciona con objetos** (**y** a partir de PHP 5.1 también **con arrays**).

### 1.3.1.- Utilización de objetos (II).

**Importante**: Qué sucede con los objetos cuando los pasas a una función, o simplemente cuando ejecutas un código como el siguiente:

$p = new Producto();

$p->nombre = 'Samsung Galaxy S';

$a = $p;

En PHP4, la última línea del código anterior crea un nuevo objeto con los mismos valores del original, copiándolos uno a uno. Si después de hacer la copia se modifica, por ejemplo, el atributo 'nombre' de uno de los objetos, el otro objeto no se vería modificado.

En PHP5, este comportamiento varía, y tiene un comportamiento más parecido a Java. El código anterior simplemente crearía un **nuevo identificador del mismo objeto**. En cuanto se utilice uno cualquiera de los identificadores para cambiar el valor de algún atributo, este cambio se vería también reflejado al acceder utilizando el otro identificador.

Cómo crear identificadores en PHP5 a un objeto ya existente

En PHP puedes crear referencias a variables (como números enteros o cadenas de texto), utilizando el operador **&**:

$a = 'Samsung Galaxy S';

$b = & $a;

Las referencias se pueden utilizar para pasarlas como parámetros a las funciones. Si utilizamos el operador & junto al parámetro, en lugar de pasar una copia de la variable, se pasa una referencia a la misma.

function suma(&$v) {

$v ++;

}

$a = 3;

suma ($a);

echo $a; // Muestra 4

[PHP: Explicación de las Referencias - Manual](https://www.php.net/manual/es/language.references.php)

Por tanto, a partir de PHP5 no puedes copiar un objeto utilizando el operador **=**. Si necesitas copiar un objeto, debes utilizar **clone**. Al utilizar **clone** sobre un objeto existente, se crea una copia de todos los atributos, como ocurría en versiones de PHP anteriores simplemente con el operador =:

$p = new Producto();

$p->nombre = 'Samsung Galaxy S';

$a = clone($p);

Además, **existe una forma sencilla de personalizar la copia para cada clase particular**. Por ejemplo, puede suceder que quieras copiar todos los atributos menos alguno. En nuestro ejemplo, al menos el código de cada producto debe ser distinto.

Este método se llamará automáticamente después de copiar todos los atributos en el nuevo objeto.

class Producto {

…

public function \_\_clone($atributo) {

$this->codigo = nuevo\_codigo();

}

…

}

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

### 1.3.2.- Utilización de objetos (III).

Hay dos formas de comparar objetos.

* **Puedes comparar objetos con ==**: compara los valores de los atributos de los objetos.

$p = new Producto();

$p->nombre = 'Samsung Galaxy S';

$a = clone($p);

// El resultado de comparar $a == $p da verdadero

// pues $a y $p son dos copias idénticas

* **Puedes comparar objetos con ===**: el resultado de la comparación será true sólo cuando las dos variables sean referencias al mismo objeto.

$p = new Producto();

$p->nombre = 'Samsung Galaxy S';

$a = clone($p);

// El resultado de comparar $a === $p da falso

// pues $a y $p no hacen referencia al mismo objeto

$a = & $p;

// Ahora el resultado de comparar $a === $p da verdadero

// pues $a y $p son referencias al mismo objeto.

**Recomendación**: Los operadores de comparación == y === no funcionan de la misma manera que acabas de ver en PHP4. Si utilizas objetos en tus programas, es recomendable que te asegures de la versión del intérprete que los va a ejecutar, utilizando por ejemplo la función phpversion.

## 1.4.- Mecanismos de mantenimiento del estado.

En la unidad anterior aprendiste a usar la sesión del usuario para almacenar el estado de las variables, y poder recuperarlo cuando sea necesario (mediante el array superglobal $\_SESSION.

Todas las variables almacenan su información en memoria de una forma u otra según su tipo. Los objetos, no tienen un único tipo, debido a sus atributos, que pueden ser de diferente tipo. Por tanto, para almacenar los objetos en la sesión del usuario, hace falta convertirlos a un formato estándar. Este proceso se llama **serialización**.

|  |  |
| --- | --- |
| Procesos | Cómo se hace |
| Serializar un objeto | $p = new Producto();  $a = serialize($p); |
| Des serializar un objeto (reconstruirlo) | $p = unserialize($a); |

La serialización genera una cadena que contiene un flujo de bytes. Esta se puede almacenar en cualquier parte (sesión del usuario, BB.DD, etc.). A partir de ella, es posible reconstruir el objeto original utilizando la función **unserialize**.

**Debes conocer**: Las funciones **serialize** y **unserialize** se utilizan mucho con objetos, pero sirven para convertir en una cadena cualquier tipo de dato, excepto el tipo **resource**.

**Cuando se aplican a un objeto**, convierten y recuperan toda la información del mismo, incluyendo sus atributos privados. **La única información que no se puede mantener** utilizando estas funciones es la que contienen **los atributos estáticos** de las clases.

Si simplemente queremos **almacenar un objeto en la sesión del usuario**:

session\_start();

$\_SESSION['producto'] = serialize($p);

Pero en PHP esto aún es más fácil. Los objetos que se añadan a la sesión del usuario son serializados automáticamente. Por tanto, no es necesario usar **serialize** ni **unserialize**. Eso sí, para poder deserializar un objeto, debe estar definida su clase.

**Debes conocer**: Si fuera necesario, es posible almacenar esta información en una base de datos. Para ello tendrás que usar las funciones **serialize** y **unserialize**, pues **en este caso PHP ya no realiza la serialización automática**.

También es posible personalizar el proceso de serialización y deserialización, igual que con el método clone. Para ello, utilizamos los métodos mágicos **\_\_sleep**, y **\_\_wakeup**.

* Si está definido **\_\_sleep** en la clase, se ejecutará antes de serializar un objeto.
* De la misma forma, si está definido **\_\_wakeup**, se ejecutará con cualquier llamada a la función **unserialize**.

Captura de pantalla de un celular

Descripción generada automáticamente

## 1.5.- Herencia (I).

La herencia es un mecanismo de la POO que nos permite definir nuevas clases en base a otra ya existente.

* Nuevas clases que heredan = **subclases**.
* Clase de la que heredan = **clase base o superclase**.

Por ejemplo, en nuestra tienda web vamos a tener productos de distintos tipos. Para ello definimos una clase Producto:

class Producto {

public $codigo;

public $nombre;

public $nombre\_corto;

public $PVP;

public function muestra() {

print "<p>" . $this->codigo . "</p>";

}

}

Si necesitas personalizar la información que vas a tratar de cada tipo de producto (por ejemplo, para los televisores, almacenar las pulgadas, o su tecnología de fabricación), puedes crear clases que hereden de **Producto**:

class TV **extends** Producto {

public $pulgadas;

public $tecnologia;

}

Aquí la palabra clave es **extends**: es la que permite que una clase herede de otra. **Los nuevos objetos que se instancien a partir de la subclase son también objetos de la clase base**; se puede comprobar utilizando el operador **instanceof**.

$t = new TV();

if ($t instanceof Producto) {

// Este código se ejecuta pues la condición es cierta

…

}

Funciones relacionadas con la herencia:

|  |  |  |
| --- | --- | --- |
| Función | Ejemplo | Significado |
| get\_parent\_class | echo "La clase padre es: " . get\_parent\_class($t); | Devuelve el nombre de la clase padre del objeto o la clase que se indica. |
| is\_subclass\_of | if (is\_subclass\_of($t, 'Producto') {  … | Devuelve **true** si el objeto o la clase del primer parámetro, tiene como clase base a la que se indica en el segundo parámetro, o **false** en caso contrario. |

::parent para llamar al constructor de la clase padre desde el constructor de la clase hija.

### 1.5.1.- Herencia (II).

* La nueva clase hereda todos los atributos y métodos públicos de la clase base, pero no los privados. Si quieres **crear atributos privados, pero que se puedan heredar a las subclases**, debes usar la **palabra protected**.
* Además, puedes **redefinir el comportamiento de los métodos existentes en la clase base**, simplemente creando en la subclase un nuevo método con el mismo nombre.

class TV extends Producto {

public $pulgadas;

public $tecnologia;

public function muestra() {

print "<p>" . $this->pulgadas . " pulgadas</p>";

}

}

* Existe una forma de evitar que las clases heredadas puedan redefinir el comportamiento de los métodos existentes en la superclase: utilizar la palabra **final**.

class Producto {

public $codigo;

public $nombre;

public $nombre\_corto;

public $PVP;

public final function muestra() {

print "<p>" . $this->codigo . "</p>";

}

}

En este caso, el método **muestra** no podría redefinirse en la clase TV.

* Incluso se puede declarar una clase utilizando **final**. En este caso **no se podrían crear clases heredadas utilizándola como base**.

final class Producto {

…

}

* Existe también **abstract**. Se utiliza de la misma forma, tanto con métodos como con clases completas, pero en lugar de prohibir la herencia, obliga a que se herede.
  + **Una clase con el modificador** **abstract** no puede tener objetos que la instancien, pero sí podrá utilizarse a través de sus subclases, instanciando objetos de estas.

abstract class Producto {

…

}

* + **Un método con el modificador abstract**, debe ser redefinido obligatoriamente por las subclases, y no podrá contener código.

class Producto {

…

abstract public function muestra();

}

**Nota**: obviamente, no se puede declarar una clase como **abstract** y **final** simultáneamente. **Abstract** obliga a que se herede para que se pueda utilizar, mientras que **final** indica que no se podrá heredar.

Texto

Descripción generada automáticamente con confianza baja

### 1.5.2.- Herencia (III).

**Sigamos con nuestro ejemplo**. Para facilitar la creación de nuevos objetos, crearemos un constructor al que se le pasará un array con los valores de los atributos del nuevo producto.

class Producto {

public $codigo;

public $nombre;

public $nombre\_corto;

public $PVP;

public function muestra() {

print "<p>" . $this->codigo . "</p>";

}

public function \_\_construct($row) {

$this->codigo = $row['cod'];

$this->nombre = $row['nombre'];

$this->nombre\_corto = $row['nombre\_corto'];

$this->PVP = $row['PVP'];

}

}

* Cuando crees un objeto de la clase TV, **¿se llamará al constructor de** **Producto**?
  + Obviamente puedes definir un nuevo constructor para las clases heredadas que redefinan el comportamiento del que existe en la clase base, tal y como harías con cualquier otro método
* **¿Puedes crear un nuevo constructor específico para** **TV** que redefina el comportamiento de la clase base?
  + **Si programas el constructor en la clase heredada**, no se llamará automáticamente. Deberás llamarlo tú, utilizando para ello la palabra **parent, y el operador de resolución de ámbito.**
  + **En cambio, si no lo programas**, sí que se llamará automáticamente.

class TV extends Producto {

public $pulgadas;

public $tecnologia;

public function muestra() {

print "<p>" . $this->pulgadas . " pulgadas</p>";

}

public function \_\_construct($row) {

parent::\_\_construct($row);

$this->pulgadas = $row['pulgadas'];

$this->tecnologia = $row['tecnologia'];

}

}

Igual que la palabra clave **self** servía para tener acceso a la clase actual. La clase **parent**, sirve para hacer referencia a la clase base de la actual.

Texto

Descripción generada automáticamente

## 1.6.- Interfaces (I).

Un interface es como una clase vacía que solamente contiene declaraciones de métodos. Se definen utilizando la palabra **interface**.

Por ejemplo, si quieres asegurarte de que todos los tipos de productos tengan un método **muestra**, puedes crear un interface como el siguiente:

interface iMuestra {

public function muestra();

}

Cuando crees las subclases, deberás indicar con la palabra **implements** que tienen que implementar los métodos declarados en este interface.

class TV extends Producto implements iMuestra {

…

public function muestra() {

print "<p>" . $this->pulgadas . " pulgadas</p>";

}

…

}

**Todos los métodos que se declaren en un interface deben ser públicos**. Además de métodos, **los interfaces podrán contener constantes pero no atributos**.

Al implementar todos los métodos declarados en el interface se asegura la interoperabilidad entre clases. Si sabes que una clase implementa un interface determinado, sabes qué nombre tienen sus métodos, qué parámetros les debes pasar y, probablemente, podrás averiguar fácilmente con qué objetivo han sido escritos.

**Por ejemplo**, en la librería de PHP está definido el interface **Countable**:

Countable {

abstract public int count ( void )

}

**Si creas una clase para la cesta de la compra en la tienda web**, podrías implementar este interface **para contar los productos** que figuran en la misma.

Por otra parte, recordar que **en PHP 5 no existe la herencia múltiple. Sin embargo, sí es posible crear clases que implementen varios interfaces**, simplemente separando la lista de interfaces por comas después de la palabra **implements**.

class TV extends Producto implements iMuestra, Countable {

…

}

**La única restricción es que los nombres de los métodos que se deban implementar en los distintos interfaces no coincidan**. Es decir, en nuestro ejemplo, el interface **iMuestra** no podría contener un método **count**, pues éste ya está declarado en **Countable**.

**Debes conocer**: también se pueden crear nuevos interfaces heredando de otros ya existentes. Se hace de la misma forma que con las clases, utilizando la palabra **extends**.

### 1.6.1.- Interfaces (II) ¿Interfaces o clases abstractas?.

Ambas permiten definir reglas para las clases que los implementen o hereden respectivamente. Y ninguna permite instanciar objetos. Las diferencias principales entre ambas opciones son:

* **En las clases abstractas, los métodos pueden contener código**. Si van a existir varias subclases con un comportamiento común, se podría programar en los métodos de la clase abstracta. Si se opta por un interface, habría que repetir el código en todas las clases que lo implemente.
* **Las clases abstractas pueden contener atributos**, y los interfaces no.
* No se puede crear una clase que herede de dos clases abstractas, pero **sí se puede crear una clase que implemente varios interfaces**.

**Por ejemplo**: en la tienda online va a haber dos tipos de usuarios: clientes y empleados. Si necesitas crear en tu aplicación objetos de tipo **Usuario** (por ejemplo, para manejar de forma conjunta a los clientes y a los empleados), tendrías que crear una clase no abstracta con ese nombre, de la que heredarían **Cliente** y **Empleado**.

class Usuario {

…

}

class Cliente extends Usuario {

…

}

class Empleado extends Usuario {

…

}

Pero si no fuera así, tendrías que decidir si crearías o no **Usuario**, y si lo harías como una clase abstracta o como un interface.

**Si** por ejemplo, **quisieras definir en un único sitio los atributos comunes** a **Cliente** y a **Empleado**, deberías crear una clase abstracta **Usuario** de la que hereden.

abstract class Usuario {

public $dni;

protected $nombre;

…

}

Pero esto no podrías hacerlo si ya tienes planificada alguna relación de herencia para una de estas dos clases.

Funciones de utilidad para interfaces en PHP 5

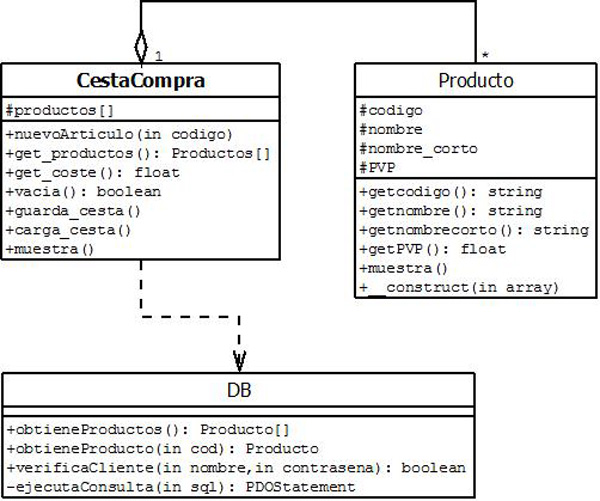
|  |  |  |
| --- | --- | --- |
| Función | Ejemplo | Significado |
| Get\_declared\_interfaces | print\_r  (get\_declared\_interfaces()); | Devuelve un array con los nombres de los interfaces declarados. |
| Interface\_exists | if (interface\_exists('iMuestra') {  …  } | Devuelve **true** si existe el interface que se indica, o **false** en caso contrario. |

Texto

Descripción generada automáticamente

## 1.7.- Ejemplo de POO en PHP (I).

Es hora de llevar a la práctica lo aprendido, llevándonos la POO a la aplicación de tienda web con la que trabajamos en la unidad anterior:



* **DB**. Va a ser la clase encargada de interactuar con la base de datos.
* **Producto**. Las instancias de esta clase representan los productos que se venden en la tienda.
* **CestaCompra**. Con esta clase vas a gestionar los productos que escoge el cliente de la tienda para comprar.

Obviamente en una tienda web real, la lista de clases que deberíamos crear sería mucho más amplia y debería haberse estudiado el diagrama resultante.

**Como se puede ver en el diagrama**:

* **Cada valor se almacenará en un atributo de tipo protected**, para limitar el acceso a su contenido y, a la vez, permitir su herencia (en caso de que en el futuro creemos clases heredadas).
* **En nuestra aplicación no será necesario cambiar sus valores**, pero sí acceder a ellos, por lo que se creará un método de tipo **get** para cada uno.

En nuestro caso, **para facilitar la creación de objetos de la clase Producto a partir del contenido de la base de datos**, le pasaremos **como parámetro un array** obtenido de una fila de la base de datos.

La **clase Producto** quedaría como sigue:

class Producto {

protected $codigo;

protected $nombre;

protected $nombre\_corto;

protected $PVP;

public function getcodigo() {return $this->codigo; }

public function getnombre() {return $this->nombre; }

public function getnombrecorto() {return $this->nombre\_corto; }

public function getPVP() {return $this->PVP; }

public function muestra() {print "<p>" . $this->codigo . "</p>";}

public function \_\_construct($row) {

$this->codigo = $row['cod'];

$this->nombre = $row['nombre'];

$this->nombre\_corto = $row['nombre\_corto'];

$this->PVP = $row['PVP'];

}

}

La **clase DB** no necesita almacenar ninguna información; simplemente deberá contener métodos para realizar acciones sobre la base de datos. Por tanto, vamos a definir en la misma únicamente métodos estáticos. No hará falta instanciar ningún objeto de esta clase. **Los métodos son**:

* **obtieneProductos**: devuelve un array con todos los productos de la BB.DD.
* **obtieneProducto($codigo)**: devuelve el producto que coincide con el código que se indica.
* **verificaCliente($nombre, $contrasena)**: devuelve **true** o **false**, según sean correctas o no las credenciales que se proporcionen.
* Utilizaremos de apoyo un método **protected**, **ejecutaConsulta**, que será el que realmente ejecute las consultas sobre la base de datos. Los otros métodos llaman a este método.

Por último, la **clase CestaCompra** debe almacenar un array con los productos que figuran en la cesta. Ese array lo crearemos como **protected**, para controlar el acceso a su contenido, pero necesitaremos un método **get\_productos** para devolverlo.

Además, hemos implementado los siguientes métodos en la misma:

* **nuevo\_articulo($codigo)**. Introduce en la cesta el artículo indicado por su código.
* **get\_productos**. Devuelve un array con todos los productos de la base de datos.
* **get\_coste**. Devuelve el coste de los productos que figuran en la cesta.
* **vacia**. Devuelve **true** o **false**, según la cesta esté o no vacía.
* **obtieneProductos**. Devuelve un array con todos los productos de la base de datos.

También necesitamos **dos funciones para guardar la cesta en la sesión del usuario, y para recuperarla**. Y programaremos otra más para mostrar el contenido de la cesta en formato HTML.

* **guarda\_cesta**. Guarda la cesta en la sesión del usuario.
* **carga\_cesta**. Recupera el contenido de la cesta de la sesión del usuario.
* **muestra**. Genera una salida en formato HTML con el contenido de la cesta.

## 1.7.1.- Ejemplo de POO en PHP (I).

El resto de ficheros que componen la tienda web tendremos que reescribirlos para que utilicen las clases que acabamos de definir. En general, **el resultado obtenido es mucho más claro y conciso**. Veamos algunos ejemplos:

* En **login.php**, para autentificar al usuario, basta con hacer:

if (DB::verificaCliente($\_POST['usuario'], $\_POST['password'])) {

…

}

* En **productos.php**: el código para vaciar o añadir un nuevo producto a la cesta de la compra del usuario será:

// Recuperamos la cesta de la compra

$cesta = CestaCompra::carga\_cesta();

// Comprobamos si se ha enviado el formulario de vaciar la cesta

if (isset($\_POST['vaciar'])) {

unset($\_SESSION['cesta']);

$cesta = new CestaCompra();

}

// Comprobamos si se quiere añadir un producto a la cesta

if (isset($\_POST['enviar'])) {

$cesta->nuevo\_articulo($\_POST['cod']);

$cesta->guarda\_cesta();

}

Para tener el código más organizado hemos creado **una función para mostrar el listado de todos los productos**:

function creaFormularioProductos()

{

$productos = DB::obtieneProductos();

foreach ($productos as $p) {

// Creamos el formulario en HTML para cada producto

…

}

}

Resultado final de la aplicación de tienda online: ver en la carpeta de Recursos del tema.

Texto

Descripción generada automáticamente

# 2.- Programación en capas.

**Caso práctico**: Su amigo trabaja en otra empresa y su función es diseñar el aspecto de los sitios web que crean. Cuando ve la aplicación de **Carlos**, queda muy impresionado por lo que ha avanzado en poco tiempo. Tras examinarla, le indica que tal y como está es muy difícil cambiar su aspecto. Tiene el código HTML distribuido en diversos ficheros, y entremezclado con el código PHP.

Si observaste el resultado obtenido en la aplicación del ejemplo programada en el apartado anterior, habrás visto como en muchas ocasiones se mezcla el código propio de la lógica de la aplicación, con el código necesario para crear el interface web que se presenta a los usuarios.

**Por ejemplo**, tanto la clase **CestaCompra** como la clase **Producto**, cuyo objetivo debería ser implementar la lógica de la aplicación, tienen un método llamado **muestra** destinado a generar etiquetas HTML. E inversamente, en algunas páginas que deberían simplemente generar HTML, puedes encontrar código que forma parte de la lógica de la aplicación.

**Existen varios métodos que permiten separar la lógica de presentación** (en nuestro caso, la que genera las etiquetas HTML) **de la lógica de negocio**, donde se implementa la lógica propia de cada aplicación.

**El más extendido es el patrón de diseño Modelo – Vista – Controlador (MVC)**. Este patrón pretende dividir el código en tres partes, dedicando cada una a una función definida y diferenciada de las otras.

* **Modelo**: es el encargado de manejar los datos propios de la aplicación. Debe proveer mecanismos para obtener y modificar la información del mismo. Si la aplicación utiliza algún tipo de almacenamiento para su información (como una BB.DD), tendrá que encargarse de almacenarla y recuperarla.
* **Vista**: es la parte del modelo que se encarga de la interacción con el usuario. En esta parte se encuentra el código necesario para generar el interface de usuario (en nuestro caso en HTML), según la información obtenida del modelo.
* **Controlador**: En este módulo se decide qué se ha de hacer, en función de las acciones del usuario con su interface.
  + Con esta información, interactúa con el modelo para indicarle las acciones a realizar y, según el resultado obtenido, envía a la vista las instrucciones necesarias para generar el nuevo interface.
  + **Resumen**: es el que controla y “sincroniza” de alguna forma la Vista con el Modelo.

La **gran ventaja de este patrón de programación** es que genera **código muy estructurado, fácil de comprender y de mantener**.

Aunque puedes programar utilizando MVC por tu cuenta, **es más habitual utilizar el patrón MVC en conjunción con un framework** o marco de desarrollo. Existen muchos frameworks para PHP.

## 2.1.- Separación de la lógica de negocio (I).

Otros mecanismos disponibles en PHP, menos complejos que la utilización del patrón MVC, y que también permiten la separación de la lógica de presentación y la lógica de negocio, son los llamados motores de plantillas.

Un **motor de plantillas web**, es una aplicación que **genera una página web a partir de un fichero con la información de presentación y otro con la lógica interna de la aplicación** (similar al modelo de MVC).

**De esta forma, es sencillo dividir el trabajo de programación de una aplicación web en dos perfiles:**

* **Un programador**, que debe conocer el lenguaje de programación en el que se implementará la lógica de la aplicación (en nuestro caso PHP)
* **Un diseñador**, que se encargará de elaborar las plantillas, (en el caso de la web básicamente en HTML, aunque como veremos la lógica de presentación que se incorpore utilizará un lenguaje propio).

En PHP **existen varios motores de plantillas** con diferentes características. Quizás **el más conocido es Smarty**, de código abierto y disponible bajo licencia LGPL (Lesser General Public License). [PHP Template Engine | Smarty](https://www.smarty.net/)

**Entre las características de Smarty, cabe destacar**:

* **Permite** la inclusión en las plantillas de una **lógica de presentación compleja**.
* **Acelera la generación de la página web resultante**.
  + Uno de los problemas de los motores de plantillas es que su utilización influye negativamente en el rendimiento.
  + Smarty convierte internamente las plantillas a guiones PHP equivalentes, y posibilita el almacenamiento del resultado obtenido en memoria temporal.
* **Al ser usado por una amplia comunidad de desarrolladores**, existen multitud de ejemplos y foros para la resolución de los problemas que te vayas encontrando al utilizarlo.

Instalación de Smarty

1. Descárgalo de la página oficial.
2. Copia los archivos de la librería Smarty (el directorio **libs** que obtienes tras descomprimir el fichero que has descargado) en tu sistema; por ejemplo, en la carpeta /usr/share/smarty



1. Modificar el fichero **php.ini** para que se incluya en la variable **include\_path** de PHP la ruta en la que acabas de instalar Smarty, y reiniciar Apache para aplicar los cambios.



1. **Crear la estructura de directorios necesaria para Smarty**. Crear 4 directorios: ***templates***, ***templates\_c***, ***configs*** y ***cache***.
   1. Es **conveniente** que estén **ubicados en un lugar no accesible por el servidor web**, y en una ubicación distinta para cada aplicación web que programes.
   2. **Por ejemplo**, puedes crear en la raíz de tu servidor una carpeta llamada **smarty**, y bajo ella otra con el nombre de cada aplicación (por ejemplo, **stienda**), que será la que contendrá las carpetas.

Si tienes problemas con la instalación de Smarty, búscate la vida con un tutorial.

Captura de pantalla de computadora

Descripción generada automáticamente con confianza media

### 2.1.1.- Separación de la lógica de negocio (II).

**Para utilizar Smarty**, simplemente tienes que **añadir a tus páginas PHP el fichero** **Smarty.class.php**, que es donde está declarada la clase **Smarty**.

Después debes instanciar un nuevo objeto de esa clase, y configurar la ruta a cada uno de los directorios que acabas de crear.

require\_once('Smarty.class.php');

$smarty = new Smarty;

$smarty->template\_dir = '/web/smarty/stienda/templates/';

$smarty->compile\_dir = '/web/smarty/stienda/templates\_c/';

$smarty->config\_dir = '/web/smarty/stienda/configs/';

$smarty->cache\_dir = '/web/smarty/stienda/cache/';

**Cuando quieras que algún valor o variable obtenido en tus páginas, esté disponible para mostrarlo en las páginas web a través de una plantilla**, tienes que usar el método **assign**, indicando el nombre del identificador. Puedes utilizar **assign** con variables de cualquier tipo, incluyendo arrays asociativos y objetos.

$smarty->assign('usuario', $\_SESSION['usuario']);

$smarty->assign('productoscesta', $cesta->get\_productos());

$smarty->assign('coste', $cesta->get\_coste());

Y **una vez que hayas preparado los identificadores** que se usarán en la plantilla, deberás mostrarla utilizando el método **display**.

$smarty->display('cesta.tpl');

**Por ejemplo**: al usar Smarty en la página de **productos.php** de la tienda online, puedes obtener algo como:

require\_once('include/DB.php');

require\_once('include/CestaCompra.php');

require\_once('Smarty.class.php');

// Recuperamos la información de la sesión

session\_start();

// Y comprobamos que el usuario se haya autentificado

if (!isset($\_SESSION['usuario']))

die("Error - debe <a href='login.php'>identificarse</a>.<br />");

// Recuperamos la cesta de la compra

$cesta = CestaCompra::carga\_cesta();

// Cargamos la librería de Smarty

$smarty = new Smarty;

$smarty->template\_dir = '/web/smarty/stienda/templates/';

$smarty->compile\_dir = '/web/smarty/stienda/templates\_c/';

$smarty->config\_dir = '/web/smarty/stienda/configs/';

$smarty->cache\_dir = '/web/smarty/stienda/cache/';

// Comprobamos si se ha enviado el formulario de vaciar la cesta

if (isset($\_POST['vaciar'])) {

unset($\_SESSION['cesta']);

$cesta = new CestaCompra();

}

// Comprobamos si se quiere añadir un producto a la cesta

if (isset($\_POST['enviar'])) {

$cesta->nuevo\_articulo($\_POST['cod']);

$cesta->guarda\_cesta();

}

// Ponemos a disposición de la plantilla los datos necesarios

$smarty->assign('usuario', $\_SESSION['usuario']);

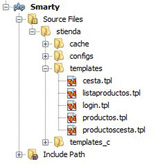
$smarty->assign('productos', DB::obtieneProductos());

$smarty->assign('productoscesta', $cesta->get\_productos());

// Mostramos la plantilla

$smarty->display('productos.tpl');

## 2.2.- Generación del interface de usuario.

Las plantillas de Smarty son ficheros con extensión **tpl**, en los que **puedes incluir prácticamente cualquier contenido propio de una página web**. Además, **en el medio intercalarás delimitadores para indicar la inclusión de datos** y de lógica propia de la presentación.

**Los delimitadores de Smarty son llaves**. De los distintos elementos que puedes incluir entre las llaves están:

* **Comentarios**: van encerrados entre asteriscos.

{\* Este es un comentario de plantilla en Smarty \*}

* **Variables**: Se incluye simplemente su nombre, precedido por el símbolo **$**.
  + **También se pueden especificar modificadores**, separándolos de la variable por una barra vertical.
  + Existen varios modificadores para, **por ejemplo, dar formato a una fecha** (**date\_format**) o **mostrar un contenido predeterminado** si la variable está vacía (**default**).

{$producto->codigo}

* **Estructura de procesamiento condicional**: if, elseif, else.
  + **Permite usar condiciones**, de forma similar a PHP, **para decidir si se procesa o no cierto contenido**.

{if empty($productoscesta)}

<p>Cesta vacía</p>

{else}

…

{/if}

* **Bucles foreach**: Son muy útiles para mostrar varios elementos, por ejemplo en una tabla.
  + Deberás indicar al menos con **from** el array en el que están los elementos, y con **item** la variable a la que se le irán asignado los elementos en cada iteración.

{foreach from=$productoscesta item=producto}

<p>{$producto->codigo}</p>

{/foreach}

* **Inclusión de otras plantillas**: Smarty permite descomponer una plantilla compleja en trozos más pequeños y almacenarlos en otras plantillas, que se incluirán en la actual utilizando la sentencia include.

<div id="cesta">

{include file="productoscesta.tpl"}

</div>

<div id="productos">

{include file="listaproductos.tpl"}

</div>

**Debes conocer**: Las que acabas de ver son una pequeña parte de las funcionalidades que ofrece Smarty. En su página web, puedes acceder a la [documentación completa](https://www.smarty.net/docsv2/es/). La última versión disponible en español de la documentación cuando se escribió esto era la versión 2.

**Ejercicio resuelto**: A partir del código obtenido utilizando POO para la tienda web, **aplica el motor de plantillas Smarty para dividir la lógica de presentación de la lógica de negocio**. Utiliza como apoyo la documentación disponible en Internet. Cuando acabes, puedes comparar lo que has obtenido con la solución propuesta, que está en la carpeta de Recursos de este tema.

Texto

Descripción generada automáticamente