

# Programación orientada a objetos en PHP.



## Caso práctico

**Carlos** empieza a darse cuenta de que, con lo que lleva aprendido de PHP, ya es capaz de hacer bastantes cosas. Ha acabado de programar la aplicación web para gestionar su colección de comics, y está satisfecho con el resultado obtenido. De vez en cuando ha tenido que echar mano de la documentación del lenguaje, para buscar información sobre cómo hacer algo, o los parámetros que requiere cierta función, pero siempre ha podido solucionarlo por sí mismo.



Sin embargo, cuando revisa el código de su aplicación, se da cuenta de que está muy desorganizado. Cada vez que necesita hacer algún cambio, o introducir un añadido, tiene que rebuscar entre las páginas para encontrar el código a reprogramar.

Y si eso le pasa con su pequeña aplicación, no se imagina lo que sucederá cuando tenga que programar una aplicación más compleja. —¡Tiene que haber algún modo de obtener un código más limpio y estructurado!



[Ministerio de Educación y Formación Profesional](#) (Dominio público)

**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Orientación a objetos en PHP.



## Caso práctico

**Carlos** comenta a **Juan** su problema, y éste le dice que seguramente gran parte del problema se arregle si utiliza programación orientada a objetos en sus aplicaciones. Le explica por encima de qué se trata y cuáles son sus ventajas, pero no puede ayudarlo mucho más. La última vez que programó en PHP, intentó utilizar objetos en su aplicación, pero las características de orientación a objetos del lenguaje dejaban mucho que desear, por lo que acabó haciéndola como siempre.



Sin embargo, por lo que ha oído, parece ser que en las últimas versiones de PHP eso ha cambiado considerablemente. El lenguaje evoluciona, y él lleva bastante tiempo sin utilizarlo, así que tendrá que actualizarse.

Mientras tanto, **Carlos** ya se ha puesto un nuevo reto: debe aprender a utilizar objetos en PHP. Y cuando tenga cierta soltura, lo primero que hará es modificar el código de su aplicación de comics.

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. En ellas se escribe el código que define el comportamiento de los objetos y se indican los miembros que formarán parte de los objetos de dicha clase. Entre los miembros de una clase puede haber:

- ✓ **Métodos.** Son los miembros de la clase que contienen el código de la misma. 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 (y por tanto, 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**.

Los pilares fundamentales de la POO son:

- ✓ **Herencia.** Es el proceso de crear una clase a partir de otra, heredando su comportamiento y características y pudiendo redefinirlos.
- ✓ **Abstracción.** Hace referencia a que 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. Visto desde el exterior, cada objeto es un ente abstracto que realiza un trabajo.
- ✓ **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.

Las 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 facilita esta tarea 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. No debe haber dos clases en un mismo fichero, ni otro código aparte del propio de la clase.



## Para saber más

En este módulo no se pretende realizar un estudio profundo de las ventajas y características de la POO, sino simplemente recordar conceptos que ya deberías haber asimilado con anterioridad. Si tienes dudas sobre algo de lo que acabamos de repasar, puedes consultar este tutorial de la web desarrolloweb.com.

[Tutorial de la web desarrolloweb.com.](http://desarrolloweb.com)

# 1.1.- Características de orientación a objetos en PHP.

---

Seguramente todo, o la mayoría de lo que acabas de ver, ya lo conocías, y es incluso probable que sepas utilizar algún lenguaje de programación orientado a objetos, así que vamos a ver directamente las peculiaridades propias de PHP en lo que hace referencia a la POO.

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



[Everaldo Coelho](#) (GNU/GPL)

```
// utilizando programación estructurada
$conProyecto = mysqli_connect('localhost', 'gestor', 'secreto', 'proyecto');
// utilizando POO
$conProyecto = new mysqli();
$conProyecto->connect('localhost', 'gestor', 'secreto', 'proyecto');
```

Sin embargo, el lenguaje PHP original no se diseñó con características de orientación a objetos. Sólo 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.

A partir de la versión de, PHP5, se reescribió el soporte de orientación a objetos del lenguaje, ampliando sus características y mejorando su rendimiento y su funcionamiento general. Aunque iremos detallando y explicando cada una posteriormente con detenimiento, las características de POO que soporta PHP incluyen:

- ✓ Métodos estáticos.
- ✓ Métodos constructores y destructores.
- ✓ Herencia.
- ✓ Interfaces.
- ✓ Clases abstractas.
- ✓ 🐘 Traits (A partir de la versión 5.4.0).

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

- ✓ Herencia múltiple.
- ✓ Sobrecarga de métodos.(incluidos los métodos constructores).
- ✓ Sobrecarga de operadores.



## Autoevaluación

Antes de PHP5, el comportamiento cuando se pasaba una variable a una función era siempre el mismo, independientemente de si la variable fuera un objeto o de cualquier otro tipo: siempre se creaba una nueva variable copiando los valores de la original.

- ☐ Verdadero.
- ☐ Falso.

Sí, es cierto. Solo a partir de PHP5, cuando se pasa un objeto como parámetro a una función, se hace por referencia y no por valor.

Recuerda que acabas de aprender que en PHP4 los objetos se pasan por valor.

### Solución

1. Opción correcta
2. Incorrecto

## 1.2.- Creación de clases.

La declaración de una clase en PHP se hace utilizando la palabra `class`. A continuación y entre llaves, deben figurar los miembros de la clase. Conviene hacerlo de forma ordenada, primero las propiedades o atributos, y después los métodos, cada uno con su código respectivo.



[Everaldo Coelho and Yellowicon](#) (GNU/GPL)

```
class Producto {  
    private $codigo;  
    public $nombre;  
    public $pvp;  
    public function muestra() {  
        echo "<p>" . $this->codigo . "</p>";  
    }  
}
```

Como comentábamos antes, es preferible que cada clase figure en su propio fichero (`producto.php`). 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');
```

Los atributos de una clase son similares a las variables de PHP. Es posible indicar un valor en la declaración de la clase. En este caso, todos los objetos que se instancien a partir de esa clase, partirán con ese valor por defecto en el atributo.

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 A6';  
$p->muestra();
```

---

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

- ✓ **public**. Los atributos declarados como **public** pueden utilizarse directamente por los objetos de la clase. Es el caso del atributo **\$nombre** anterior.
- ✓ **private**. Los atributos declarados como **private** sólo pueden ser accedidos y modificados por los métodos definidos en la clase, no directamente por los objetos de la misma. Es el caso del atributo **\$codigo**.

## 1.2.1.- Creación de clases (I).

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. O necesitas conocer cuál será el nuevo valor antes de asignarlo. 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:



[Everaldo Coelho \(Yellowicon\)](#)  
(GNU/GPL)

```
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; }
```

Aunque no es obligatorio, 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**, y a continuación el nombre del atributo con la primera letra en mayúsculas.



### Debes conocer

En PHP5 se introdujeron los llamados métodos mágicos, entre ellos `__set` y `__get`. Si se declaran estos dos métodos en una clase, PHP los invoca automáticamente cuando desde un objeto se intenta usar un atributo no existente o no accesible. Por ejemplo, el código siguiente simula que la clase `Producto` tiene cualquier atributo que queramos usar.

```
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.

[Métodos mágicos.](#)



## Autoevaluación

En lugar de programar un método `set` para modificar el valor de los atributos privados en que sea necesario, puedo utilizar el método mágico `__set`.

- ☐ Verdadero.
- ☐ Falso.

Sí, pero tendrías que comprobar el nombre del atributo usado y asignar el valor al adecuado.

Si el atributo es privado, no es accesible; por tanto, cuando se le intente asignar un valor, se llamará al método mágico `__set` si existe.

## Solución

1. Incorrecto
2. Opción correcta

## 1.2.2.- Creación de clases (II).

Cuando desde un objeto se invoca un método de la clase, a éste se le pasa siempre una referencia al objeto que hizo la llamada. Esta referencia se almacena en la variable `$this`. Se utiliza, por ejemplo, en el código anterior para tener acceso a los atributos privados del objeto (que sólo son accesibles desde los métodos de la clase).



[David Vignoni \(GNU/GPL\)](#)

```
echo "<p>" . $this->codigo . "</p>";
```

Dentro de la clase para acceder a sus métodos o atributos propios usaremos `$this->nombre`, (salvo que el atributo o el método sea **estático**) Fíjate que al atributo se le quita "\$". Veamos un ejemplo:

```
class Persona{
    private $nombre;
    private $perfil;
    public function getNombre(){
        return $this->nombre;
    }
    public function setNombre($n){
        $this->nombre=$n;
    }
    public function saludo(){
        //Fíjate como hacemos referencia al método getNombre
        echo "Hola {$this->getNombre()}, Buenos dias";
    }
}
$persona1=new Persona();
$persona1->setNombre("Luis");
$persona1->saludo();
```



### 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.

[Referencia.](#)

Además de métodos y propiedades, 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 (obviamente, de ahí su nombre), no usan el carácter `$` y, además, su valor va siempre entre comillas y 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 = 'gestor';  
    ...  
}  
echo DB::USUARIO;
```

Es importante resaltar que 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 PHP, una clase puede tener atributos o métodos estáticos, también llamados a veces atributos o métodos de clase. Se definen utilizando la palabra clave `static`.

```
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 utilizan para guardar información general sobre la misma, como puede ser el número de objetos que se han instanciado. Sólo existe un valor del atributo, que se almacena a nivel de clase.

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.



## Para saber más

El operador "::", tiene un nombre curioso en programación : **Paamayim Nekudotayim**, Significa "doble dos puntos" en Hebreo.

## 1.2.3.- Creación de clases (III).

PHP permite a los desarrolladores declarar métodos **constructores** para las clases. Aquellas que tengan un método constructor lo invocarán en cada nuevo objeto creado, lo que nos permite la inicialización que el objeto pueda necesitar antes de ser usado. Como PHP no admite sobrecarga de métodos **sólo podremos crear un constructor por clase**.



[David Vignoni \(GNU/GPL\)](#)

Por motivos de retrocompatibilidad con PHP3 y PHP4, si PHP no puede encontrar una función `__construct()` de una clase dada, se buscará la función constructora del estilo antiguo, por el nombre de la clase.

Veamos unos ejemplos de uso del constructor:

```
class Persona{
    public static $id;
    private $nombre;
    private $perfil;
    public function __construct(){
        $perfil="Público";
    }
}
//creamos persona1 que tiene inicializado su atributo $perfil a Público.
$persona1=new Persona();
// Podemos comprobarlo
var_dump($persona1);
```

```
class Persona{
    public static $id;
    private $nombre;
    private $perfil;
    public function __construct($n, $p){
        $this->nombre=$n;
        $this->perfil=$p;
    }
}
// Creamos un objeto de la clase Persona e inicializamos sus atributos;
// Fijate que ya NO podremos usar: $persona1=new Persona(); ya que el constructor espera dos parámetros
$persona1=new Persona("Juan", "Privado");
//Podemos comprobarlo
var_dump($persona1);
```

Con el uso de las funciones `"func_get_args()"`, `"fun_get_arg()"` y `"func_num_arg()"`, podemos pasar distinto número de parámetros a un constructor "simulando" la sobrecarga del mismo, en el **Anexo 1** de este tema se ve un ejemplo de como hacerlo. [Ir a Anexo 1](#). Otra posibilidad es usar el método mágico `"__call"` para capturar llamadas a métodos que no estén implementados.

Los constructores del estilo antiguo (llamados como el nombre de la clase) están **OBSOLETOS** desde PHP 7.0, por lo que serán eliminados en futuras versiones. Se debería utilizar siempre "`__construct()`" en código nuevo.

También es posible definir un método destructor, que debe llamarse "`__destruct`" y 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');
```

Los métodos destructores aparecen en PHP5; no existían en versiones anteriores del lenguaje.



## Autoevaluación

¿Cuál es la utilidad del operador de resolución de ámbito `::`?

- ☐ Nos permite hacer referencia a la clase del objeto actual.
- ☐ Se utiliza para acceder a los elementos de una clase, como constantes y miembros estáticos.

Entonces, ¿cuál es el significado de `self`?

Sí; y por tanto debe usarse precedido por el nombre de una clase, o por una referencia a una clase como `self`.

# Solución

1. Incorrecto
2. Opción correcta

## 1.3.- Utilización de objetos.

Ya sabes cómo instanciar un objeto utilizando `new`, y cómo acceder a sus métodos y atributos públicos con el operador flecha:

```
$p = new Producto();
$p->nombre = 'Samsung Galaxy A6';
$p->muestra();
```



[Pixabay](#) (Dominio público)

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, a partir de PHP5 se incluyen una serie de funciones útiles para el desarrollo de aplicaciones utilizando POO.

### Funciones de utilidad para objetos y clases en PHP

Función	Ejemplo	Significado
<code>get_class()</code>	<pre>echo "La clase es: " . get_class(\$p);</pre>	Devuelve el nombre de la clase del objeto.
<code>class_exists</code>	<pre>if (class_exists('Producto')) {     \$p = new Producto();     . . . }</pre>	Devuelve <code>true</code> si la clase está definida o <code>false</code> en caso contrario.
<code>get_declared_classes()</code>	<pre>print_r(get_declared_classes());</pre>	Devuelve un array con los nombres de las clases definidas.



Función	Ejemplo	Significado
<code>class_alias()</code>	<pre>class_alias('Producto', 'Articulo'); \$p = new Articulo();</pre>	Crea un alias para una clase.
<code>get_class_methods()</code>	<pre>print_r(get_class_methods('Producto'));</pre>	Devuelve un array con los nombres de los métodos de una clase que son accesibles desde dónde se hace la llamada.
<code>method_exists()</code>	<pre>if (method_exists('Producto', 'vende') {     ... }</pre>	Devuelve <code>true</code> si existe el método en el objeto o la clase que se indica, o <code>false</code> en caso contrario, independientemente de si es accesible o no.
<code>get_class_vars()</code>	<pre>print_r(get_class_vars('Producto'));</pre>	Devuelve un array con los nombres de los atributos de una clase que son accesibles desde dónde se hace la llamada.
<code>get_object_vars()</code>	<pre>print_r(get_object_vars(\$p));</pre>	Devuelve un array con los nombres de los métodos de un objeto que son accesibles desde dónde se hace la llamada.
<code>property_exists()</code>	<pre>if (property_exists('Producto', 'codigo') {     . . . }</pre>	Devuelve <code>true</code> si existe el atributo en el objeto o la clase que se indica, o <code>false</code> 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 así como el tipo del dato devuelto (caso que lo haya). Para

ello, debes especificar el tipo antes del parámetro. Para el dato devuelto poner `":tipoDato"`, después de la declaración de la función o el método y antes de las llaves.

```
public function precioProducto(Producto $p) :float {  
    . . .  
    return $precio;  
}
```

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

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

Una característica de la POO que debes tener muy en cuenta es 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;
```



[Negative Space](#) (Dominio público)

En PHP4, la última línea del código anterior crea un nuevo objeto con los mismos valores del original, de la misma forma que se copia cualquier otro tipo de variable. 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.

Sin embargo, a partir de PHP5 este comportamiento varía. El código anterior simplemente crearía un **nuevo identificador del mismo objeto**. Esto es, 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. Recuerda que, aunque haya dos o más identificadores del mismo objeto, en realidad todos se refieren a la única copia que se almacena del mismo.



### Debes conocer

Para crear nuevos identificadores en PHP a un objeto ya existente, se utiliza el operador "=". Sin embargo, como ya sabes, este operador aplicado a variables de otros tipos, crea una copia de la misma. En PHP puedes crear referencias a variables (como números enteros o cadenas de texto), utilizando el operador & , que ya vimos en el paso de parámetros por referencia:

```
$a = 'Samsung Galaxy A6';  
$b = &$a;
```

En el ejemplo anterior, **\$b** es una referencia a la variable **\$a**. Cuando se cambia el valor de una de ellas, este cambio se refleja en la otra.

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
```

De esta forma, dentro de la función se puede modificar el contenido de la variable que se pasa, no el de una copia.

[Referencias en 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 del mismo en un nuevo objeto.

```
$p = new Producto();  
$p->nombre = 'Samsung Galaxy A6';  
$a = clone($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 y, por tanto, quizás no tenga sentido copiarlo al crear un nuevo objeto. Si éste fuera el caso, puedes crear un método de nombre `__clone` en la clase. 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();  
    }  
    ...  
}
```



## Autoevaluación

¿Cuál es el nombre de la función que se utiliza para hacer una copia de un objeto?

- ☐ `clone.`
- ☐ `__clone.`

Efectivamente. Además, cuando utilizas la función `clone`, si la clase tiene definido un método de nombre `__clone`, se llama automáticamente.

Revisa lo último que acabas de aprender.

## Solución

1. Opción correcta
2. Incorrecto

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

A veces tienes dos objetos y quieres saber su relación exacta. Para eso, en PHP puedes utilizar los operadores "==" y "===".

Si utilizas el operador de comparación "==", comparas los valores de los atributos de los objetos. Por tanto dos objetos serán iguales si son instancias de la misma clase y, además, sus atributos tienen los mismos valores.



[Soumil Kumar](#) (Dominio público)

```
$p = new Producto();
$p->nombre = 'Samsung Galaxy A6';
$a = clone($p);
// El resultado de comparar $a == $p da verdadero
// pues $a y $p son dos copias idénticas
```

Sin embargo, si utilizas el operador "===", el resultado de la comparación será **true** sólo cuando las dos variables sean referencias al mismo objeto.

```
$p = new Producto();
$p->nombre = 'Samsung Galaxy A6';
$a = clone($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.
```

A veces puede ser útil mostrar el contenido de un objeto sin tener que usar `var_dump()` para ello podemos usar el método mágico `__toString()`. Este método siempre debe devolver un **String**.

```
class Persona{
    public $nombre;
    public $apellidos;
    public $perfil;
    public function __toString() :String{
        return "{$this->apellidos}, {$this->nombre}, Tu perfil es: {$this->perfil}";
    }
}

$persona = new Persona();
$persona->nombre="Manuel";
$persona->apellidos="Gil Gil";
$persona->perfil="Público";
echo $persona; //muestra: Gil Gil, Manuel, Tu perfil es: Público
```



## 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. El proceso es muy sencillo; se utiliza el array superglobal `$_SESSION`, añadiendo nuevos elementos para ir guardando la información en la sesión.



[Negative Space](#) (Dominio público)

El procedimiento para almacenar objetos es similar, pero hay una diferencia importante. Todas las variables almacenan su información en memoria de una forma u otra según su tipo. Los objetos, sin embargo, no tienen un único tipo. Cada objeto tendrá unos atributos u otros en función de su clase. Por tanto, para almacenar los objetos en la sesión del usuario, hace falta convertirlos a un formato estándar. Este proceso se llama serialización.

En PHP, para serializar un objeto se utiliza la función `serialize()`. El resultado obtenido es un **string** que contiene un flujo de **bytes**, en el que se encuentran definidos todos los valores del objeto.

```
$p = new Producto();  
$a = serialize($p);
```

Esta cadena se puede almacenar en cualquier parte, como puede ser la sesión del usuario, o una base de datos. A partir de ella, es posible reconstruir el objeto original utilizando la función `unserialize()`.

```
$p = unserialize($a);
```



### 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, deberíamos hacer por tanto:



```
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()`.

```
session_start();
$_SESSION['producto'] = $p;
```

Para poder deserializar un objeto, debe estar definida su clase. Al igual que antes, si lo recuperamos de la información almacenada en la sesión del usuario, no será necesario utilizar la función `unserialize`.

```
session_start();
$p = $_SESSION['producto'];
```



## Debes conocer

Como ya viste en el tema anterior, el mantenimiento de los datos en la sesión del usuario no es perfecta; tiene sus limitaciones. 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.

En PHP además tienes la opción de personalizar el proceso de serialización y deserialización de un objeto, utilizando los métodos mágicos "`__sleep()`" y "`__wakeup()`". Si en la clase está definido un método con nombre "`__sleep()`", se ejecuta antes de serializar un objeto. Igualmente, si existe un método "`__wakeup()`", se ejecuta con cualquier llamada a la función `unserialize()`.

[Métodos mágicos `\_\_sleep\(\)` y `\_\_wakeup\(\)`.](#)



## Para saber más

A partir de PHP7 se han implementado filtros para `unserialize()`: Esta característica busca el proporcionar una mejor seguridad al deserializar objetos en datos no fiables. Previene de posibles inyecciones de código al

capacitar al desarrollador a crear listas blancas de clases que deben ser deserializadas.

```
// convertir todos los objetos a un objeto __PHP_Incomplete_Class
$data = unserialize($foo, ["allowed_classes" => false]);
// convertir todos los objetos a un objeto __PHP_Incomplete_Class excepto a los de MiC
$data = unserialize($foo, ["allowed_classes" => ["MiClase", "MiClase2"]]);
// comportamiento predeterminado (lo mismo que omitir el segundo argumento) que acepta
$data = unserialize($foo, ["allowed_classes" => true]);
```



## Autoevaluación

Si serializas un objeto utilizando `serialize`, ¿puedes almacenarlo en una base de datos MySQL?

- ☐ Verdadero.
- ☐ Falso.

No solo puedes, sino que además es la forma correcta de hacerlo. `serialize` convierte el objeto en una cadena, que se puede almacenar donde sea necesario.

Piensa en cómo almacenarías el objeto entonces.

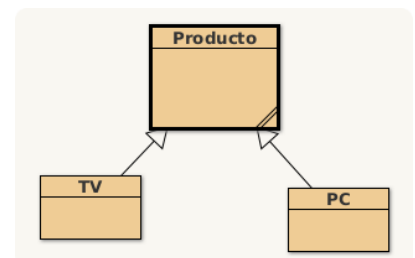
## Solución

1. Opción correcta
2. Incorrecto

## 1.5.- Herencia.

La herencia es un mecanismo de la POO que nos permite definir nuevas clases en base a otra ya existente. Las nuevas clases que heredan también se conocen con el nombre de **subclases**. La clase de la que heredan se llama **clase base** o **superclase**.

Por ejemplo, en nuestra tienda web vamos a tener productos de distintos tipos. En principio hemos creado para manejarlos una clase llamada **Producto**, con algunos atributos y un método que genera una salida personalizada en formato HTML del código.



Captura de pantalla BlueJ (Elaboración Propia)

```
class Producto {  
    public $codigo;  
    public $nombre;  
    public $nombre_corto;  
    public $pvp;  
    public function muestra() {  
        echo "<p>" . $this->codigo . "</p>";  
    }  
}
```

Esta clase es muy útil si la única información que tenemos de los distintos productos es la que se muestra arriba. Pero, si quieres personalizar la información que vas a tratar de cada tipo de producto (y almacenar, por ejemplo para los televisores, las pulgadas que tienen o su tecnología de fabricación), puedes crear nuevas clases que hereden de **Producto**. Por ejemplo, **TV**, **PC**, **Movil**.

```
class TV extends Producto {  
    public $pulgadas;  
    public $tecnologia;  
}
```

Como puedes ver, para definir una clase que herede de otra, simplemente tienes que utilizar la palabra **extends** indicando la superclase. 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.  
    . . .  
}
```

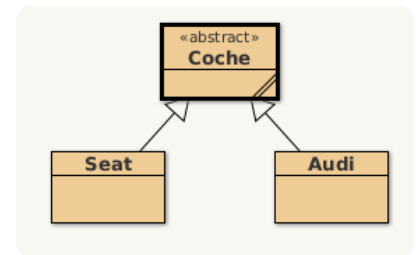
Antes viste algunas funciones útiles para programar utilizando objetos y clases. Las de la siguiente tabla están además relacionadas con la herencia.

### Funciones de utilidad en la herencia en PHP

Función	Ejemplo	Significado
<code>get_parent_class()</code>	<code>echo get_parent_class(\$t);</code>	Devuelve el nombre de la clase padre del objeto o la clase que se indica.
<code>is_subclass_of()</code>	<code>if(is_subclass_of(\$t,'Producto'))</code>	Devuelve <code>true</code> si el objeto o la clase del primer parámetro, tiene como clase base a la que se indica en el segundo parámetro, o <code>false</code> en caso contrario.

## 1.5.1.- Herencia (I).

La nueva clase hereda todos los atributos y métodos públicos de la clase base, pero no los privados. Si quieres crear en la clase base un método no visible al exterior (como los privados) que se herede a las subclases, debes utilizar la palabra **protected** en lugar de **private**. 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.



Captura de pantalla BlueJ (Elaboración propia.)

```
class TV extends Producto {
    public $pulgadas;
    public $tecnologia;
    public function muestra() {
        echo "<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**. Si en nuestro ejemplo hubiéramos hecho:

```
class Producto {
    public $codigo;
    public $nombre;
    public $nombre_corto;
    public $pvp;
    public final function muestra() {
        echo "<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 {
    . . .
}
```

Opuestamente al modificador **final**, 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. Es decir, una clase con el modificador **abstract** no puede tener objetos que la instancien, pero sí podrá utilizarse de clase base y sus subclases sí podrán utilizarse para instanciar objetos.

```
abstract class Producto {  
    . . .  
}
```

Y un método en el que se indique **abstract**, debe ser redefinido obligatoriamente por las subclases, y no podrá contener código.

```
class Producto {  
    . . .  
    abstract public function muestra();  
}
```

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.



## Autoevaluación

La función `is_subclass_of` recibe como primer parámetro:

- ☐ Un objeto.
- ☐ Un objeto o una clase.

Revisa la información que contiene la última tabla de funciones.

Efectivamente, puedes pasarle un objeto o el nombre de una clase como primer parámetro, y te dirá si es o no una subclase del nombre de clase que le pases como segundo parámetro.

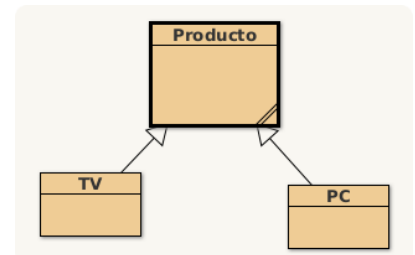
## Solución

1. Incorrecto
2. Opción correcta

## 1.5.2.- Herencia (II).

Vamos a hacer una pequeña modificación en nuestra clase **Producto**. 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 $pvp;
    public function __construct($row) {
        $this->codigo = $row['cod'];
        $this->nombre = $row['nombre'];
        $this->nombre_corto = $row['nombre_corto'];
        $this->pvp = $row['pvp'];
    }
    public function muestra() {
        echo "<p>" . $this->codigo . "</p>";
    }
}
```



Captura de pantalla BlueJ (Elaboración Propia)

¿Qué pasa ahora con la clase **TV**, qué hereda de **Producto**? Cuando crees un nuevo objeto de esa clase, ¿se llamará al constructor de **Producto**? ¿Puedes crear un nuevo constructor específico para **TV** que redefina el comportamiento de la clase base?

Empezando por esta última pregunta, 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. Y dependiendo de si programas o no el constructor en la clase heredada, se llamará o no automáticamente al constructor de la clase base.

En **PHP**, si la clase heredada no tiene constructor propio, se llamará automáticamente al constructor de la clase base (si existe). Sin embargo, si la clase heredada define su propio constructor, deberás ser tú el que realice la llamada al constructor de la clase base si lo consideras necesario, utilizando para ello la palabra **parent** y el operador de resolución de ámbito.

```
class TV extends Producto {
    public $pulgadas;
    public $tecnologia;
    public function __construct($row) {
        parent::__construct($row); //llama al constructor de la clase padre
        $this->pulgadas = $row['pulgadas'];
        $this->tecnologia = $row['tecnologia'];
    }
    public function muestra() {
        echo "<p>" . $this->pulgadas . " pulgadas</p>";
    }
}
```

Ya viste con anterioridad cómo se utilizaba la palabra clave `self` para tener acceso a la clase actual. La palabra `parent` es similar. Al utilizar `parent` haces referencia a la clase base de la actual, tal y como aparece tras `extends`.



## Autoevaluación

**Si una subclase no tiene método constructor, y su clase base sí lo tiene, cuando se instancie un nuevo objeto de la subclase:**

- ☐ Se llamará automáticamente al constructor de la clase base.
- ☐ No se llamará automáticamente al constructor de la clase base.

Efectivamente. Fíjate que esta llamada automática solo ocurre cuando la clase heredada no tiene constructor; en otro caso tendrás que hacer tú la llamada manualmente.

Revisa el comportamiento de los constructores en la herencia que acabas de estudiar.

## Solución

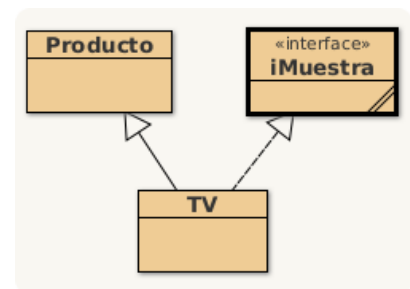
1. Opción correcta
2. Incorrecto



## 1.6.- Interfaces.

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

Por ejemplo, antes viste que podías crear nuevas clases heredadas de **Producto**, como **TV** o **Ordenador**. También viste que en las subclases podías redefinir el comportamiento del método **muestra** para que generara una salida en HTML diferente para cada tipo de producto.



Captura de pantalla BlueJ (Elaboración propia.)

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();
}
```

Y 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() {
        echo "<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.

Un interface es como un contrato que la clase debe cumplir. 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.

Antes aprendiste que en PHP una clase sólo puede heredar de otra, que 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**.



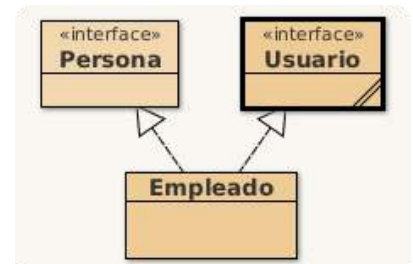
## Debes conocer

En PHP 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 (I).

Una de las dudas más comunes en POO, es qué solución adoptar en algunas situaciones: 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.



Captura de pantalla BlueJ (Elaboración propia.)

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.

Para finalizar con los interfaces, a la lista de funciones de PHP relacionadas con la POO puedes añadir las siguientes.

## Funciones de utilidad para interfaces en PHP

Función	Ejemplo	Significado
<code>get_declared_interfaces()</code>	<code>print_r(get_declared_interfaces());</code>	Devuelve un array con los nombres de los interfaces declarados.
<code>interface_exists()</code>	<code>if(interface_exists('iMuestra'))</code>	Devuelve <code>true</code> si existe el interface que se indica, o <code>false</code> en caso contrario.



### Autoevaluación

Si en tu código utilizas un interface, y quieres crear uno nuevo basándote en él:

- ☐ Puedes utilizar la herencia para crear el nuevo constructor extendiendo al primero.
- ☐ No puedes hacerlo, pues no se puede utilizar herencia con los interfaces; solo con las clases.

Sí; se hace utilizando `extends`, de la misma forma que con las clases.

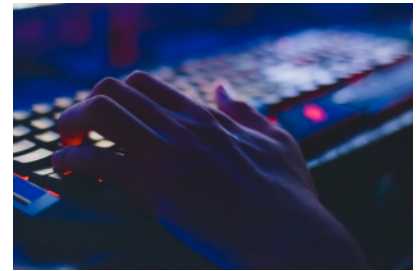
Al final del punto anterior se comenta este caso concreto. Revísalo.

### Solución

1. Opción correcta
2. Incorrecto

## 1.7.- Ejemplo de POO en PHP.

Es hora de llevar a la práctica lo que has aprendido. Vamos a aplicar los principios de la POO para realizar un "CRUD" a la tabla "**productos**" de la base de datos "**proyecto**" usada en unidades anteriores. Recuerda que las credenciales para acceder a la base de datos eran "**gestor::secreto**" y en la tabla usuarios habíamos creado los usuarios "**admin::secreto**" y "**gestor::pass**". En los enlaces siguientes puedes ver un vídeo en "**youtube**" de la aplicación ya terminada y un resumen textual del vídeo.



[Soumil Kumar](#) (Dominio público)

- ✓ [Enlace al Vídeo.](#)
- ✓ [Resumen textual.](#)

Básicamente si accedemos como invitado podremos ver el listado de productos y el detalle de cada uno de ellos, y si nos validamos podremos también crear, borrar y actualizar productos.

Vamos a utilizar una estructura para esta práctica muy común, en el directorio donde guardemos los archivos de la práctica crearemos una carpeta "**class**" donde guardaremos todas las clases y un carpeta "**public**" donde guardaremos todas las páginas "**php, html...**", que vayamos a necesitar visualizar en el navegador.

```
class
├── Conexion.php
├── Familia.php
├── Producto.php
├── Usuario.php
public
├── borrar.php
├── cerrar.php
├── crear.php
├── detalle.php
├── listado.php
├── login.php
└── update.php
```

Captura de pantalla  
terminal Ubuntu  
(Elaboración propia)


Crearemos una clase "**Conexion**" y una clase para todas las tablas que vayamos a usar. El convenio es poner el nombre del archivo exactamente igual al nombre de la clase que implementa, es decir, el archivo "**Usuario.php**" implementará la clase "**Usuario**".




Para no tener que estar haciendo el "**require**" de cada uno de los archivos de clase que vayamos a necesitar, vamos a utilizar la "**autocarga de las clases**", que es básicamente un mecanismo por el que cada vez que instanciamos un objeto de una clase hace el "**require**" del archivo donde se encuentra implementada. Por eso es muy importante que respete las convenciones de nombres. El código para ello sería el siguiente:

```
spl_autoload_register(function ($class) {
    require "../class/" . $class . ".php";
});
```

Utiliza una función anónima en la llamada, no te preocupes por eso ahora, el funcionamiento es, que cuando en cualquier archivo del directorio "**public**" hacemos una declaración del tipo "**\$objeto=new Objeto()**" automáticamente busca un archivo en el directorio "**class**" (un nivel por abajo, de ahí el "**..**") llamado "**Objeto.php**" para hacer el "**require**", si no existe me dará un error.

Veamos el contenido de los archivos:

- ✓ **Conexion.php:**  [Descargar conexion.php](#) (pdf - 28,44 KB) Implementa la clase "**Conexion**" que contendrá básicamente los parámetros para conectarnos, así como un método "**getConexion()**", que nos devolverá la conexión. Todas las demás clases **heredarán de esta**, todo sus atributos serán "**private**" menos el de la conexión que lo hacemos "**protected**".

- ✓ **Usuario.php:**  [Descargar usuario.php](#) (pdf - 28,02 KB) Implementa la clase "**Usuario**" que contendrá los atributos de la tabla **usuario**. Tiene el método "**isValidado(\$usu, \$pass)**" que devolverá **true** si encuentra a un usuario con esa contraseña y **false** en otro caso.
- ✓ **Familia.php:**  [Descargar familia.php](#) (pdf - 25,50 KB) Implementa la clase "**Familia**" que contendrá los atributos de la tabla **usuario**. Tiene el método "**recuperarFamilias()**" que usaremos para rellenar los "**options**" para la lista desplegable del campo familia en crear y actualizar productos, este método devuelve el "**\$stmt**".
- ✓ **Producto.php:**  [Descargar producto.php](#) (pdf - 38,89 KB) Implementa la clase "**Producto**" que contendrá los atributos de la tabla **producto**. Tiene a parte del constructor los "**setter**" para cada uno de los atributos, un método para cada una de las operaciones de CRUD, el método "**existeNombreCorto(\$n)**" que devolverá **true** si el nombre corto pasado por parámetro existe y **false** en otro caso y el método "**recuperarProductos()**" que devuelve el "**\$stmt**" de la consulta para recuperar los productos ordenados alfabéticamente, este método me servirá para mostrar los productos en "**listado.php**".

```
function existeNombreCorto($nc){
    if ($this->id == null) {
        $consulta = "select * from productos where nombre_corto=:nc";
    } else {
        $consulta = "select * from productos where nombre_corto = :nc AND id != :i";
    }
    $nc = func_get_arg(0);
    $stmt = $this->conexion->prepare($consulta); //podemos acceder a conexion al ser "protec
    try {
        if ($this->id == null)
            $stmt->execute([':nc' => $nc]);
        else
            $stmt->execute([':nc' => $nc, ':i' => $this->id]);
    } catch (PDOException $ex) {
        die("Error al comprobar el nombre corto: " . $ex->getMessage());
    }
    if ($stmt->rowCount() == 0) return false; //No existe
    return true; //existe
}
```

Fíjate que en este método tenemos que controlar si estamos en **crear** (compruebo que no existe ese **nombre\_corto** en toda la base de datos) o **actualizar** (no comprobaremos el **nombre\_corto** del código del producto a actualizar ya que ese **nombre\_corto** ya existe).

Para todas las clases que heredan de la clase **Conexión** el constructor será:

```
public function __construct(){
    parent::__construct();
}
```



## Recomendación

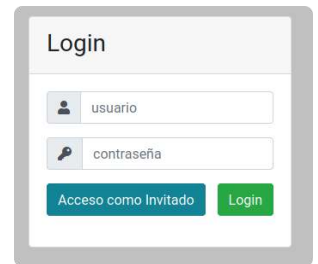
Podemos instalar en **Visual Studio Code**, multitud de extensiones que nos ayudarán en nuestro desarrollo, una de ellas es "**PHP Getters & Setters**" que nos crea automáticamente los "**getters**" y los "**setters**" del atributo que deseemos. Una vez instalada sólo tenemos que hacer **click** derecho sobre el atributo que queramos y nos aparecerá en el menú contextual la opción de crear el "**get**", el "**set**" o ambos.

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

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

- ✓ El action de "login.php" para autentificar quedaría:

```
$nombre = trim($_POST['usuario']);
$pass = trim($_POST['pass']);
if (strlen($nombre) == 0 || strlen($pass) == 0) {
    error("Error, El nombre o la contraseña no pueden contener s
}
$usuario = new Usuario();
if (!$usuario->isValidado($nombre, $pass)) {
    $usuario = null;
    error("Credenciales Inválidas");
}
$usuario = null;
$_SESSION['nombre'] = $nombre;
header('Location: listado.php');
```



Captura de pantalla de Firefox  
(Elaboración propia)

- ✓ En "listado.php", el código recuperar todos los productos quedaría:

```
//Hacemos el autoload de las clases
spl_autoload_register(function ($class) {
    require "../class/" . $class . ".php";
});
// recuperamos los productos
$productos = new Producto();
$stmt = $productos->recuperarProductos();
// Cerramos todo
$productos = null;
```

- ✓ En "detalle.php" el código para recuperar los detalles del producto seleccionado quedaría:

```
$producto = new Producto();
$producto->setId($id);
$datos = $producto->read();
$producto = null;
```

- ✓ En "borrar.php" el código para borrar el producto de código \$cod:



```
$producto = new Producto();
$producto->setId($cod);
$producto->delete();
$producto = null;
$_SESSION['mensaje'] = "Artículo Borrado Correctamente";
header('Location: listado.php');
```

✓ En "crear.php" por una parte para comprobamos que no existe el nombre corto:

```
if ($producto->existeNombreCorto($nc)) {
    error("Error ya existe un nombre corto con ese valor.");
}
```

Y el código para guardar el producto nuevo

```
//recogemos los datos del formulario, trimamos las cadenas
$nombre = trim($_POST['nombre']);
$nomCorto = trim($_POST['nombrec']);
$pvp = $_POST['pvp'];
$des = trim($_POST['descripcion']);
$familia = $_POST['familia'];
comprobar($nombre, $nomCorto);
// si hemos llegado aqui todo ha ido bien vamos a crear el registro
$producto->setNombre($nombre);
$producto->setNombre_corto($nomCorto);
$producto->setPvp($pvp);
$producto->setFamilia($familia);
$producto->setDescripcion($des);
$producto->create();
$_SESSION['mensaje'] = 'Producto creado Correctamente';
$producto = null;
header('Location: listado.php');
```

Unas últimas consideraciones, obviamente todo el código es mejorable, algunas partes de este código tienen carácter meramente didáctico y obviamente se pueden conseguir mejores implementaciones del mismo. Se podría haber hecho uso de `includes` y funciones para evitar repetir algunas partes de código. Para completar los productos en el "create" y el "update" se podría haber hecho con el constructor inicializando los atributos con los nuevos datos y no usando los `setters`. Piensa que mejoras podrías hacer al código propuesto y compártelas con tus compañeros, es una forma muy útil de aprender.

Prueba a instalar y ejecutar la aplicación resultante. Revisa el código y comprueba que entiendes su funcionamiento.

 [Aplicación de Ejemplo](#) (zip - 15,83 KB)



## Autoevaluación

La clase DB tiene todos sus métodos estáticos. No tiene sentido por tanto crear ningún objeto de esa clase, y podría haberse implementado igualmente como un interface.

- ☐ Verdadero.
- ☐ Falso.

¿Seguro? Piensa en cómo lo harías.

Efectivamente, no se podría, pues no se pueden programar métodos en los interfaces.

### Solución

1. Incorrecto
2. Opción correcta

## 2.- Programación en capas.



### Caso práctico

Después de unas semanas, **Carlos** ha conseguido reprogramar su aplicación de catalogación utilizando orientación a objetos. Le ha costado bastante esfuerzo, pero reconoce que el resultado merece la pena. El código resultante es mucho más limpio, y cada clase de las que ha creado tiene un cometido concreto y bien definido.



ora es mucho más fácil hacer cambios en el código. Definitivamente, tendrán que utilizar orientación a objetos cuando empiecen el nuevo proyecto. Mientras tanto, se propone volver sobre una asignatura pendiente: mejorar el aspecto de sus páginas. Y aunque conoce el lenguaje HTML, y sabe utilizar las hojas de estilo, decide pedirle consejo a un amigo suyo que se dedica al diseño de webs.

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.

Le comenta que en su empresa utilizan mecanismos de separación del código y le anima a que los pruebe. Si lo hace, él se ofrece a darle un diseño más profesional a su aplicación. —¡Manos a la obra!

En el ejemplo anterior, hemos programado una aplicación web sencilla utilizando programación orientada a objetos. Sin embargo, si observaste el resultado obtenido, 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.

De igual manera el encabezado de las páginas era repetido una y otra vez (importar **Bootstrap**, **FontAwesome**...). Además existe un problema añadido: si el proyecto es extenso y en distintos archivos tenemos clases de igual nombre podremos tener colisión entre ellas.

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 un SGBD), 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.

La gran ventaja de este patrón de programación es que genera código muy estructurado, fácil de comprender y de mantener. En la web puedes encontrar muchos ejemplos de implementación del modelo MVC en PHP.

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 numerosos frameworks disponibles en PHP, muchos de los cuales incluyen soporte para MVC como **Laravel** y **Symfony**. En esta unidad no profundizaremos en la utilización de un framework específico, pero utilizaremos **Blade** que es el sistema de plantillas de **Laravel**, el cual nos permite generar HTML dinámico con una sintaxis mucho más limpia que si usáramos PHP plano.

## 2.1.- Namespaces en PHP.

---

Los **namespaces** o espacios de nombres permiten crear proyectos complejas con mayor flexibilidad evitando problemas de colisión entre clases, métodos, funciones y mejorando la legibilidad del código. aparecen a partir de **PHP5.3**



[Open Clipart \(CC0\)](#)

Los **namespace** son un contenedor que nos permitirá agrupar nuestro código para darle un uso posterior de esta manera evitamos conflictos de nombre. Por ejemplo, tenemos 2 funciones con el mismo nombre esto generaría un conflicto de nombre pero mediante el uso de **namespace** se da solución a este problema. En definitiva los **namespace** nos ayudaran para organizar y reestructurar mejor nuestro código y proporcionan una manera para agrupar clases, interfaces, funciones y constantes relacionadas.

Podemos compararlo con el sistema de archivos, los archivos están dentro de carpetas y dentro de éstas hay a su vez otras carpetas con otros archivos. Una carpeta se comporta como si fuera un **namespace**, por ejemplo, no puede haber dos archivos con el mismo nombre en la misma carpeta, pero sí puede haber dos archivos con el mismo en distintas.

Veamos como funcionan. Hay que declararlo en la primera línea de código

```
<?php
namespace Proyecto;
//la declaración del namespace debe ser la primera línea si no obtendremos un error
```

En un espacio de nombres se engloban:

- ✓ Constantes.
- ✓ Funciones.
- ✓ Clases, interfaces, traits, clases abstractas.

Vemos un ejemplo más completo. Creemos el archivo "ejemploNamespace.php"

```
<?php
namespace Proyecto;
const E = 2.7182;
function saludo(){
    echo "Buenos días";
}
class Prueba{
    private $nombre;
    public function probando(){
        echo "Esto es el método probando de la clase Prueba";
    }
}
```

Para poder usar este archivo en otra parte podemos hacerlo de varias formas:

```
<?php
include "ejemploNamespace.php";
echo Proyecto\E; // accedemos a la constante
Proyecto\saludo(); // accedemos a la función
$prueba=new Proyecto\Prueba();
$prueba->probando();
```

O bien :

```
<?php
include "ejemploNamespace.php";
use const Proyecto\E;
use function Proyecto\saludo;
use Proyecto\Prueba;
// ahora ya podemos usarlos
echo E;
saludo();
$prueba = new Prueba();
$prueba->probando();
```

Es una buena práctica de programación colocar los archivos del espacio de nombres en carpetas que tienen los mismos nombres que el **namespace** declarado.

La posibilidad de declarar el uso de una función o constante de un **namespace** está disponible solo a partir de PHP 5.6.



## Autoevaluación

En un **namespace** solo guardaremos las clases para evitar colisiones entre ellas.

 Sugerencia

☐ Verdadero ☐ Falso

**Falso**

Podemos guardar también funciones y constantes.

## 2.2.- Gestionar Dependencias.

A la hora de empezar el desarrollo de un proyecto en PHP, es necesario conocer todas las librerías que necesitaremos. La instalación de estas librerías puede ser una tarea trabajosa si lo hacemos a mano, pero existen gestores de dependencias que se encargan de realizarla de forma automática, gestionando esas librerías de forma sencilla y eficaz.

**Composer** es un gestor de dependencias en proyectos, para programación en PHP. Eso quiere decir que nos permite gestionar (declarar, descargar y mantener actualizados) los paquetes de software en los que se basa nuestro proyecto PHP. Entre sus principales virtudes destacan:



[WizardCat](#) (MIT)

- ✓ Es simple de utilizar
- ✓ Cuenta con un repositorio propio donde podemos encontrar casi de todo ([Packagist](#))
- ✓ Disminuye significativamente problemas de cambio de entorno de ejecución (Mediante su funcionalidad de *congelar* dependencias)
- ✓ Actualiza las librerías a nuevas versiones fácilmente si queremos mediante "composer update".

Vemos a empezar a trabajar con **Composer**, para ello veremos la instalación y los primeros pasos:

### ✓ Instalación

Nos descargamos el archivo desde la página oficial ([acceder a Composer](#)), desde esa página nos recomiendan ejecutar las siguientes instrucciones:

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') === 'e0012edf3e80b6978849f5eff0d4b4e4c79ff
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Debes bajarte los comandos actualizados en el enlace siguiente: [Descargar Composer](#).

Una vez hecho esto y si no ha habido problemas tendremos el archivo "composer.phar" en el directorio donde estemos, para hacer una instalación para todos los usuarios del sistema, es decir global (se recomienda hacerlo así), desde la terminal escribiremos lo siguiente:

```
sudo mv composer.phar /usr/local/bin/composer
```

Para comprobar que todo funciona teclea desde la terminal el comando "composer" te debe aparecer algo como lo siguiente:



Captura de pantalla Ubuntu (Elaboración propia)

## ✓ Primeros pasos

Veamos su uso, en la raíz del directorio donde vaya a estar nuestro proyecto y desde la terminal tecleamos **"composer init"** a continuación se nos hacen unas preguntas y vamos contestando:

- ➡ **Package name (<vendor>/<name>)** : aquí nos pregunta organización y nombre podemos poner **"usuario/usuario"** por ejemplo
- ➡ **Description []** : Ponemos una descripción del proyecto
- ➡ **Author** : ponemos autor y correo en el formato siguiente: autor <correo>, por ejemplo: **"usuario <usuario@correo.es>"**
- ➡ **Minimum Stability []** : Lo podemos dejar en blanco.
- ➡ **Package Type []** : Aquí elegiremos **project**.
- ➡ **License []** : Ponemos la licencia por ejemplo **GPL**
- ➡ A las siguientes preguntas sobre definir nuestras dependencias, ahora decimos que no.

Al final nos muestra un archivo **JSON** , se nos pregunta que si queremos generarlo, contestamos que si y nos genera el archivo **"composer.json"**. Una vez generado el archivo tecleando **"composer install"** nos instalará las dependencias.

Una de las cosas que hace **composer** es gestionarnos el tema de la **autocarga** de clases/librerías (acuérdate que en apartados anteriores ya habíamos visto una forma de hacerlo). Para ello debemos crearnos, en el raíz del proyecto que vayamos a usar, un directorio de nombre por ejemplo **"src"** y modificar el archivo **"composer.json"**. Para el **"autoload"** de clases y librerías se hace uso de los **namespaces**.

```

1  "name": "usuario/usuario",
2      "description": "Ejemplo Blade",
3      "type": "project",
4      "config": {
5          "optimize-autoloader": true
6      },
7      "autoload": {
8          "psr-4": {
9              "Clases\\": "src"
10         }
11     },
12     "license": "GNU/GPL",
13     "authors": [
14         {
15             "name": "usuario",
16             "email": "usuario@correo.es"
17         }
18     ]
19 }
```



Fíjate de la línea **7** a la **10** lo que estamos indicando es que vamos a tener un **namespace** de nombre Clases y que su directorio real va a ser "**src**". En este directorio meteremos todas las clases y comenzaremos el archivo como ya vimos "**namespace Clases**". Las tres líneas anteriores es para que se optimice la **autocarga** de las clases, no es necesario pero si recomendable. El nombre que se le suele dar al directorio de clases de un proyecto es "**src**" de "**source**".

Cada vez que modifiquemos el fichero "**composer.json**" hay que hacer un "**composer install**" para instalar las nuevas dependencias que hayamos puesto.

Si deseamos añadir dependencias a posteriori podemos escribir el comando: "**composer require vendor/libreria**", por ejemplo "**composer require fzaninotto/faker**", para instalar la librería que nos permite generar datos aleatorios de prueba.



## Para saber más

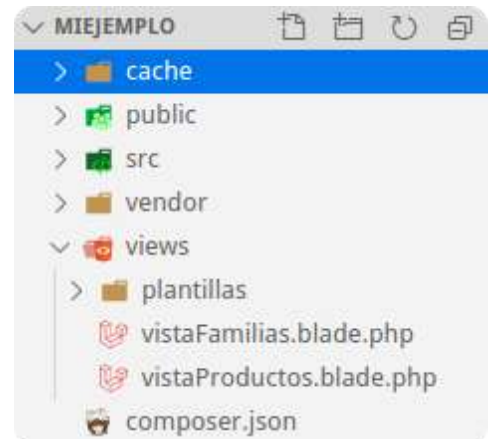
En **Packagist** hay paquetes para casi todo lo que se ocurra en **PHP**, tenemos **Carbon** para gestionar fechas, **PHPMailer** para gestionar mails, extensiones para crear códigos de barra, motores de plantillas como **Blade** usado en **Laravel** y que veremos a continuación y mucho mucho más. Antes de intentar programar una aplicación/librería mira si ya existe en **Packagist**, con **Composer** será muy sencillo instalarla.

Si te fijas para el **autoload** en el archivo "**composer.json**" utilizamos "**psr4**". Esto es un estándar de programación en **PHP** propuestos por el **php-fig** (Grupo de interoperabilidad para **Frameworks PHP**). En el enlace siguiente podrás obtener más información de estos estándares: [Información sobre estándares "psr"](#).

## 2.3.- Separación de la lógica de negocio.

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 (template engines).

Un **motor de plantillas web** es una aplicación que genera una página web a partir de un fichero con la información de presentación (denominado plantilla o template, que viene a ser similar a la vista en el patrón MVC) 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), y 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).



Captura de pantalla de Visual Studio Code (Elaboración propia.)

En PHP existen varios motores de plantillas con diferentes características. Nosotros trabajaremos con **Blade** que es el motor de plantillas que viene con Laravel.

**Blade** es un sistema de plantillas que nos permite generar HTML dinámico con una sintaxis mucho más limpia que si usáramos PHP plano. Veamos como instalarlo, configurarlo y unos primeros pasos

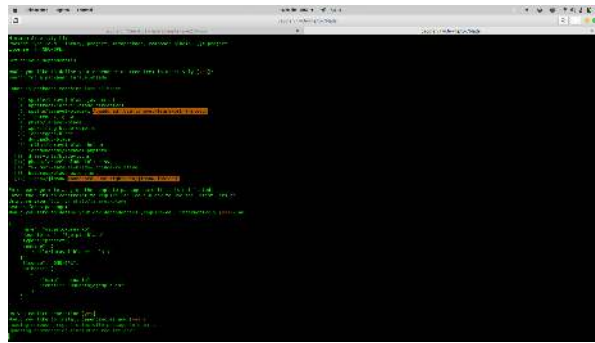
**Blade** necesita una carpeta "cache" (donde se guardara una cache de las vistas para su carga) y una carpeta "views" que será la carpeta donde guardaremos los archivos de plantilla que creemos. Todas las vistas **Blade** tienen que tener la extensión ".blade.php". Para configurar un proyecto donde vayamos a utilizar **Blade**, haremos lo siguiente:

En la carpeta del proyecto creamos las carpetas "public" (será la parte accesible desde el servidor web), "src" (guardaremos las clases que vayamos a crear si es el caso), "cache" y "views". En la carpeta "cache" el servidor web (Apache) necesita permiso de escritura, si estamos en **Windows** no hay problema, si estamos en **Linux** y sabes como hacerlo lo más seguro es que pongas como grupo de estas carpetas a Apache y des los permisos apropiados. Para no complicarnos desde la terminal y en la carpeta de nuestro proyecto una vez creadas las carpetas tecleamos: "chmod 777 cache" .

Una vez creada la estructura de nuestro proyecto iniciamos **composer** acuérdate que esto lo hacíamos con "composer init" contestamos a las preguntas iniciales como ya vimos y las siguientes así:

- ✔ "Would you like to define your dependencies (require) interactively [yes]?" en vez escribir **no**, le damos a intro ("yes" es la opción por defecto)
- ✔ "Search for a package:" ponemos "laravel-blade" y elegimos la opción: "philo/laravel-blade". A veces el paquete puede no aparecer (puede haber cambiado el nombre, el desarrollador...), no pasa nada, vete a [Packagist](https://packagist.org) , desde su buscador localiza el paquete y sigue las instrucciones, que básicamente son poner unas líneas en tu archivo "composer.json" y hacer "composer install". O generar el archivo y hacer el "composer require" después como ya vimos.
- ✔ Para las dependencias de desarrollo (dev) contestamos que **no**.
- ✔ "Do you confirm generation [yes]?" le damos a intro ("yes" es la opción por defecto)

- ✓ "Would you like to install dependencies now [yes]?" le damos a intro ("yes" es la opción por defecto) . En este paso se descargan las librerías seleccionadas, en este caso: "philo/laravel-blade" en la carpeta "vendor" que si no existe se creará.



Captura de pantalla Ubuntu (Elaboración propia)

Una vez hecho esto podemos implementar la "**autocarga**" y el "**optimizador**" de la misma como ya vimos. Al final el archivo "**composer.json**" nos debería quedar más o menos así:

```
{
  "name": "usuario/usuario",
  "description": "Ejemplo Blade",
  "type": "project",
  "require": {
    "philo/laravel-blade": "^3.1"
  },
  "config": {
    "optimize-autoloader": true
  },
  "autoload": {
    "psr-4": {
      "Clases\\": "src"
    }
  },
  "license": "GNU/GPL",
  "authors": [
    {
      "name": "usuario",
      "email": "usuario@ejemplo.es"
    }
  ]
}
```

Hacemos "**composer install**" y ya estamos en condiciones de empezar a usar **Blade** en nuestro proyecto.



## Autoevaluación

¿ Necesitamos usar el Framework Laravel pasar sacar ventajas del gestor de plantillas Blade ?

- ☐ Verdadero
- ☐ Falso.

Incorrecto. Con **Composer** podemos hacer uso de **Blade** sin tener que utilizar **Laravel**.

Correcto. Al utilizar **Composer** podemos instalar las dependencias adecuadas para poder usar **Blade** en nuestro proyecto.

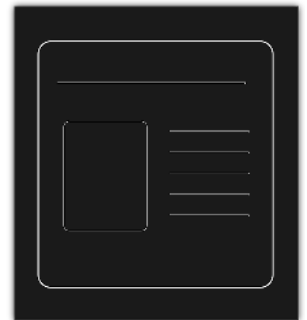
## Solución

1. Incorrecto
2. Opción correcta

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

Para utilizar **Blade**, y suponiendo que estamos trabajando con la estructura de directorios que se propuso en el apartado anterior, simplemente tienes que añadir a tus páginas del directorio "**public**" lo siguiente:

```
<?php
require 'vendor/autoload.php';
use Philo\Blade\Blade;
$views = '../views';
$cache = '../cache';
$blade = new Blade($views, $cache);
echo $blade->view()->make('vista')->render();
```

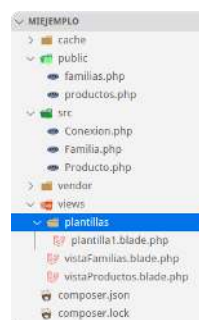


[OpenClipart-Vectors](#) (Dominio público)

Esto me cargaría la vista "**vista.blade.php**" de la carpeta "**views**". Fíjate que solo hace falta poner el nombre de la vista y no la extensión "**.blade.php**".

Para entender esto mejor desarrollaremos un proyecto con dos páginas, una para ver el listado de **productos** de la base de datos **proyecto** y otra para ver el listado de **familias**.

A las clases **Conexion**, **Productos** y **Familias** que ya la hemos visto a lo largo del tema, vamos a modificarlas para usar **namespaces** en todo el proyecto. La estructura de nuestro proyecto de ejemplo podría ser la siguiente:



Captura de pantalla  
Visual Studio Code  
(Elaboración propia)

- ✓ En "**src**" tendremos las clases para conectarnos a la base de dato y recuperar los datos de **familias** y **productos**.
- ✓ En "**public**" Las dos páginas que realmente mostraremos
- ✓ En "**views**" crearemos una carpeta "**plantillas**" (pondremos las plantillas que reusaremos las veces que queramos) y las dos vistas, la que me generará el listado de **productos** y la que me generará el listado de **familias**.

Vamos a ello:

Las clases ya sabemos como hacerlas simplemente vamos a poner en el encabezado de las mismas, lo siguiente. Ten en cuenta que el nombre del **namespace** viene dado por lo que

pusimos en el "autoload" del composer y la carpeta de las mismas también. Si los has cambiado debes reflejarlo en "composer.json"

```
<?php
namespace Clases;
use PDO;
use PDOException;
```

Al usar **namespace** es necesario poner "use" a los métodos y la clase **PDO**. Si no lo hacemos para no tener errores tendríamos que andar poniendo "\$conexion = new \PDO(. . . ), catch(\PDOException) . . ."

Las páginas que vamos a ver (**familias** y **productos**) comporten una estructura como el esqueleto **html**, la carga de **bootstrap** o las hojas de estilos, todo esto lo podemos meter en la plantilla y así nos evitamos tener que repetir el mismo código una y otra vez. Veamos la plantilla

```
1 <!doctype html>
2 <html lang="es">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6         content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, mi
7     <meta http-equiv="X-UA-Compatible" content="ie=edge">
8     <!-- css para usar Bootstrap -->
9     <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap
10 <title>@yield('titulo')</title>
11 </head>
12 <body style="background:#0277bd">
13 <div class="container mt-3">
14     <h3 class="text-center mt-3 mb-3">@yield('encabezado')</h3>
15     @yield('contenido')
16 </div>
17 </body>
18 </html>
```

cada instrucción de **Blade** viene precedida de "@". En las líneas 10, 14 y 15 verás "@yield(' . . . ')" se usa para mostrar el contenido de una sección determinada. Para cada página cambiaremos: **contenido**, **título** y **cabecera**.

Veamos como hacemos uso de esta plantilla en las vistas:

Vista: "vistaProductos.blade.php"

```
1 @extends('plantillas.plantilla1')
2 @section('titulo')
3     {{$titulo}}
4 @endsection
5 @section('encabezado')
6     {{$encabezado}}
```

```

7  @endsection
8  @section('contenido')
9      <table class="table table-striped">
10         <thead>
11             <tr class="text-center">
12                 <th scope="col">Código</th>
13                 <th scope="col">Nombre</th>
14                 <th scope="col">Nombre Corto</th>
15                 <th scope="col">Precio</th>
16             </tr>
17         </thead>
18         <tbody>
19             @foreach($productos as $item)
20                 <tr class="text-center">
21                     <th scope="row">{{ $item->id }}</th>
22                     <td>{{ $item->nombre }}</td>
23                     <td>{{ $item->nombre_corto }}</td>
24                     @if($item->pvp>100)
25                         <td class='text-danger'>{{ $item->pvp }}</td>
26                     @else
27                         <td class='text-success'>{{ $item->pvp }}</td>
28                     @endif
29                 </tr>
30             @endforeach
31         </tbody>
32     </table>
33 @endsection

```

### Vista "vistaFamilias.blade.php"

```

1  @extends('plantillas.plantilla1')
2  @section('titulo')
3      {{ $titulo }}
4  @endsection
5  @section('encabezado')
6      {{ $encabezado }}
7  @endsection
8
9  @section('contenido')
10     <table class="table table-striped">
11         <thead>
12             <tr class="text-center">
13                 <th scope="col">Código</th>
14                 <th scope="col">Nombre</th>
15             </tr>
16         </thead>
17         <tbody>
18             @foreach($familias as $item)
19                 <tr class="text-center">
20                     <th scope="row">{{ $item->cod }}</th>
21                     <td>{{ $item->nombre }}</td>
22                 </tr>
23             @endforeach
24         </tbody>
25     </table>

```

```
26 | </table>
27 | @endsection
```

Lo primero que hacemos con "`@extends('...')`" es llamar a la plantilla, para ello indicamos la ruta (fíjate que la extensión "`.blade.php`" no hace falta ponerla). Acuérdate que creamos tres "`yields`" uno llamado **título**, otro **encabezado** y otro **contenido**, para rellenarlos ponemos el bloque "`@section('...'), @endsection`" con el nombre que pusimos en los "`yields`".

A las vistas mandamos unas variables (en el apartado siguiente veremos como). A la vista "`vistaProductos.blade.php`" mandamos las variables "`$titulo`, `$encabezado` y `$productos`", `$producto` trae los productos de las base de datos (los mandamos con un "`fetchAll()`" de una consulta a la tabla productos). Cuando usamos **Blade** no hace falta poner "`<?php echo $titulo; ?>`" que es como lo hemos hecho hasta ahora, nos basta con poner "`{{...}}`"

Puedes observar además que también podemos poner un bucle para recorrer, en este caso, los productos "`@foreach(...), endforeach`".

De igual manera con un bloque "`@if, @else, @endif`" pintamos los precios de más de 100€ de rojo y el resto de verde.

El código **HTML** lo ponemos tal cual, si necesitásemos poner código **PHP** en una página **Blade** lo pondríamos entre las directivas "`@php, @endphp`"

En el enlace siguiente tenemos un manual muy completo de **Blade** en "**styde.net**"

[Manual Blade](https://styde.net/)



## 2.4.- Generación del interface de usuario.

En el apartado anterior configuramos las vistas y una plantilla para ahorrarnos repetir código, a las vistas las pasamos unas variables como título encabezado, las listas de datos que queramos mostrar . . . Veamos como podemos llamar a estas vistas y pasar variables a las mismas.

En la carpeta "**public**" (la única carpeta que realmente necesita ser accesible para el navegador) vamos a tener dos archivos: "**familias.php**" que se encargará de llamar a la vista "**vistaFamilias.blade.php**" y pasar las variables necesarias, y "**productos.php**" que hará lo mismo con la vista "**vistaProductos.blade.php**". Fíjate en el código.



[kreatikar](#) (Pixabay License)

Página: "**productos.php**"

```
1 <?php
2
3 require '../vendor/autoload.php';
4
5 use Philo\Blade\Blade;
6 use Clases\Producto;
7
8 $views = '../views';
9 $cache = '../cache';
10
11 $blade = new Blade($views, $cache);
12 $titulo='Productos';
13 $encabezado='Listado de Productos';
14 $productos=(new Producto())->recuperarProductos();
15 echo $blade->view()->make('vistaProductos', compact('titulo', 'encabezado', 'productos'))->render
```

El funcionamiento es muy sencillo:

- ✓ Con el "**require**" llamamos al archivo "**autoload.php**" de la carpeta "**vendor**" (esta carpeta la crea automáticamente "**composer**" con las librerías y clases necesarias. En este proyecto lo necesario para que funcione **Blade** y el **autoload**)
- ✓ Las líneas del "**use**" son para poder usar nuestras Clases y las de "**Philo\Blade**" (acuérdate del **namespace** de las mismas que era Clase).
- ✓ Las siguientes son para indicar la ruta relativa de las carpetas "**views**" y "**cache**" y guardarla en sendas variables.
- ✓ En la siguiente nos creamos un objeto de la clase "**Blade**", le pasamos las rutas de las carpetas "**views**" y "**cache**" e inicializamos las variables que pasaremos a la vista: **\$titulo**, **\$encabezado** y **\$productos**. Esta última línea la hemos simplificado, en realidad podíamos haber puesto:

```
$productos = new Productos();  
$productos->recuperarProductos();
```

- ✓ En la última línea llamamos a la vista: 'vistaProductos' de la carpeta "views" y le pasamos usando "compact()" las tres variables: \$titulo, \$encabezado y \$productos. En realidad a la vista hay que pasarle un **array** asociativo "nombre de la variable=>valor" que es lo que hace precisamente el método "compact()".

Página: "familias.php"

```
1 <?php  
2  
3 require '../vendor/autoload.php';  
4  
5 use Philo\Blade\Blade;  
6 use Clases\Familia;  
7  
8 $views = '../views';  
9 $cache = '../cache';  
10  
11 $blade = new Blade($views, $cache);  
12 $titulo='Familias';  
13 $encabezado='Listado de Familias';  
14 $familias=(new Familia())->recuperarFamilias();  
15 echo $blade->view()->make('vistaFamilias', compact('titulo', 'encabezado', 'familias'))->render()
```

Si te fijas es igual a lo que hicimos en "productos.php".

A continuación se deja un enlace con los archivos del proyecto. Para poder ver el mismo funcionando tendréis que tener **Composer** instalado hacer "composer init" instalar la librería de **Blade** y **autoload** y hacer "composer install". 📁 [Descargar archivos](#). (zip - 5,71 KB)



## Para saber más

En el enlace siguiente hay documentación sobre la función "compact()" :  
[Enlace a la documentación oficial.](#)



## Ejercicio resuelto

A partir de la página de "login" que ya vimos en el ejemplo 1.7.1 modifica el ejemplo que acabamos de ver para poder acceder a listar "productos" y

"familias" solo si estamos validados.

Usa **Blade** para crear la vista que cargue la página de "**login**" y modifica listar usuarios y productos para que muestren el usuario con el que estamos **logueados** y el botón **cerrar sesión**.

Cuando nos validemos accederemos a una página con dos "botones/enlaces" uno para ir a "**familias.php**" y otro a "**productos.php**".

**Se recomienda abordar este ejercicio desde cero para repasar todos los conceptos.**

Mostrar retroalimentación

Aquí se deja una posible solución, recuerda que tienes que montar el proyecto para que este funcione, es decir, inicializar **Composer** e instalar con él las dependencias necesarias.



[Descargar solución propuesta.](#) (zip - 9,96 KB)



## Autoevaluación

Las plantillas que crees en Blade es preferible alojarlas:

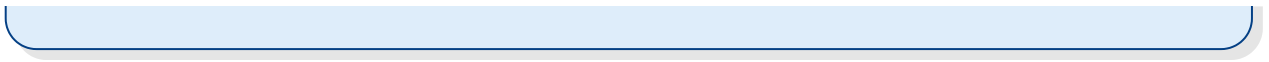
- ☐ En un lugar no accesible por el servidor web.
- ☐ En un lugar accesible por el servidor web.

Correcta. En realidad es PHP el que debe acceder a ellas, no el servidor web.

No, pues en este caso podrían acceder a ellas los usuarios de tu web. Recuerda que es el código PHP el que utiliza las plantillas para generar las páginas que ven los usuarios, y por tanto éstos no necesitan acceder a ellas a través del servidor web.

## Solución

1. Opción correcta
2. Incorrecto



# Anexo 1

---

Ejemplo de como podemos pasar distintos números de parámetros a un constructor, simulando la sobrecarga.

```
<?php

class Persona
{
    private $nombre;
    private $perfil;

    public function __construct()
    {
        $num = func_num_args(); //guardamos el número de argumentos
        switch ($num) {
            case 0:
                break;
            case 1:
                //recuperamos el argumento pasado
                $this->nombre = func_get_arg(0); // los argumentos empiezan a contar por 0
                break;
            case 2:
                $this->nombre = func_get_arg(0);
                $this->perfil = func_get_arg(1);
        }
    }
}

//Ahora será válido el siguiente código.
$persona1 = new Persona();
$persona2 = new Persona("Alicia");
$persona3 = new Persona("Alicia", "Público");
var_dump($persona1);
echo "<break>";
var_dump($persona2);
echo "<break>";
var_dump($persona3);
```