Unidad 5 – Programación orientada a objetos

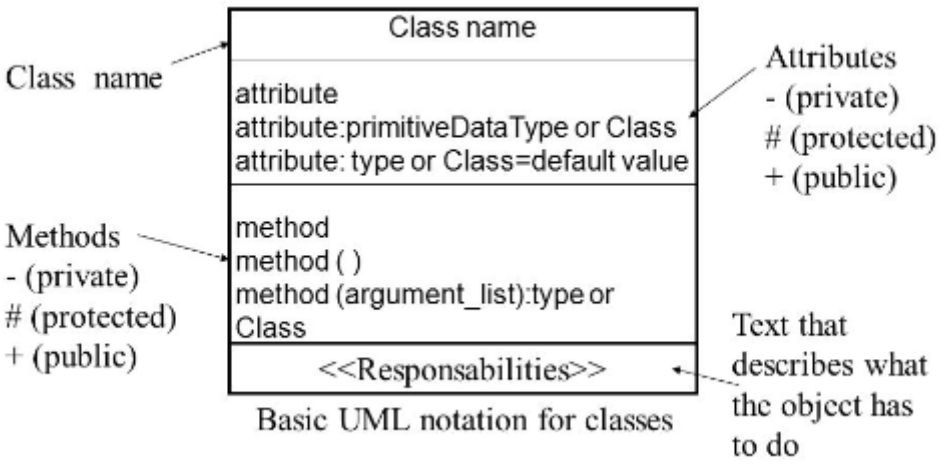
ÍNDICE

1. Características de orientación a objetos en PHP

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 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 (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 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.

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.

En la versión actual, php8 se ha reescrito y potenciado el soporte de orientación a objetos del lenguaje, ampliando sus características y mejorando su rendimiento y su funcionamiento general.

Las características de POO que soporta php8 incluyen:

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

2. Creación de clases

La declaración de una clase en PHP se hace utilizando la palabra **`class.`** Es muy recomendable separar la implementación de las clases del programa principal en ficheros diferentes. Desde el programa principal se puede cargar la clase mediante **`include o include_once`** seguido del nombre del fichero de clase. El nombre de la clase debe coincidir con el nombre del fichero que la implementa (con la extensión .php)

A continuación, y entre llaves deben figurar los miembros de la clase. De forma ordenada: primero las propiedades o atributos y después los métodos.

```php
<?php
class NombreClase {
// propiedades
// y métodos
}
?>
```

Es preferible que cada clase figure en su propio fichero. Además, se suele usar para las clases nombres que comiencen por letra mayúscula para distinguirlos de los objetos y otras variables.

```php
<?php

class Producto {
    private $codigo;
    public $nombre;
    public $PVP;

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

?>
```

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

```php
$p = new Producto();
```

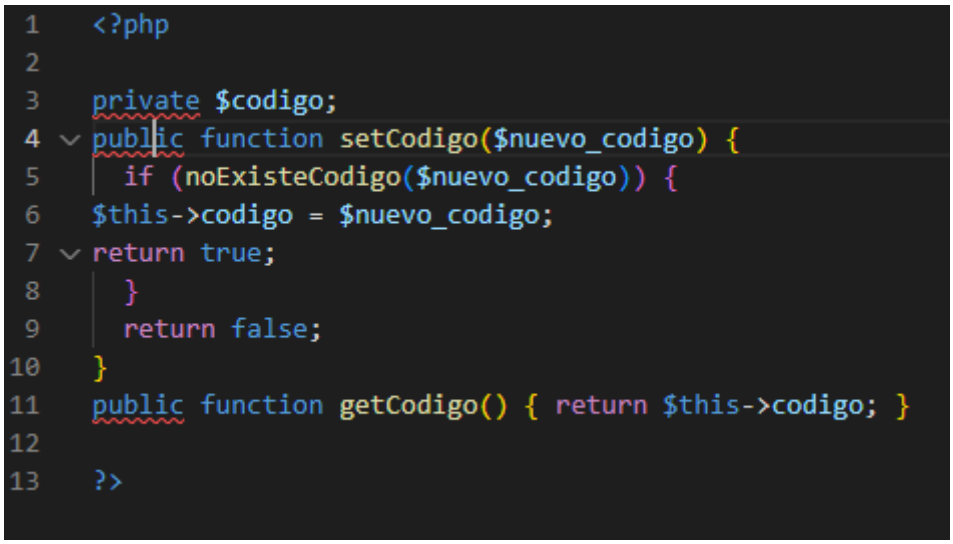
Para acceder desde un objeto a sus atributos o a los métodos de la clase, debes utilizar el operador flecha ->

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 `$nombre` o `$PVP` anterior.
- `private`. Los atributos declarados como `private` solo 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`.

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. También se usan para 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 saber 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. Veamos un ejemplo:



```php
<?php

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. (Ejemplo línea 11). Y el que nos permite modificarlo por set (Ejemplo línea 4).

2.1 Métodos mágicos

En PHP 5, se introdujeron los llamados métodos mágicos, también conocidos como *magic methods* que se pueden sobrescribir para sustituir su comportamiento.
En la documentación de PHP tienes más información sobre los métodos mágicos:
https://www.php.net/manual/es/language.oop5.magic.php

Veamos los más destacables:

- `__construct()` lo veremos en párrafos siguientes
- `__destruct()` → se invoca al perder la referencia. Se utiliza para cerrar una conexión a la BD, cerrar un fichero, ...
- `__toString()` → representación del objeto como cadena. Es decir, cuando hacemos echo $objeto se ejecuta automáticamente este método.
- `__get(propiedad), __set(propiedad, valor)` → Permitiría acceder a las propiedad privadas, aunque siempre es más legible/mantenible codificar los *getter/setter*.
- `__isset(propiedad), __unset(propiedad)` → Permite averiguar o quitar el valor a una propiedad.
- `__sleep(), __wakeup()` → Se ejecutan al recuperar (*unserialize^*) o almacenar un objeto que se serializa (*serialize*), y se utilizan para permite definir qué propiedades se serializan.
- `__call(), __callStatic()` → Se ejecutan al llamar a un método que no es público. Permiten sobrecargan métodos.

Ejemplo: el código siguiente simula que la clase Producto tiene cualquier atributo que queramos usar:

```php
<?php

class Producto {
  private $atributos = array();

  public function __get($atributo) {
      return $this->atributos[$atributo];
  }
  public function __set($atributo, $valor) {
      $this->atributos[$atributo] = $valor;
  }
}

?>
```

2.2 Variable $this

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`.

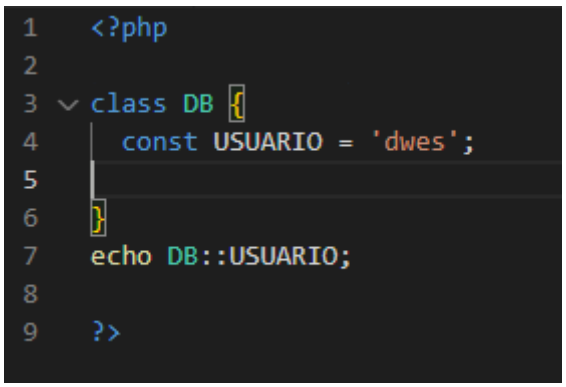
Se utiliza por ejemplo en el siguiente código para tener acceso a los atributos privados del objeto (que solo son accesibles desde los métodos de la clase).

```php
<?php

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; }

?>
```

2.3 Constantes

Además de métodos y propiedades, en una clase también se pueden definir **constantes**, utilizando la palabra `const`.

Es importante no confundir los atributos con las constantes.
Los constantes no pueden cambiar su valor, no usan el carácter $ y además, su valor va siempre entre comillas y está asociado a la clase. 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** (se utiliza para acceder a los elementos de una clase).



- No es necesario que exista ningún objeto de una clase para poder acceder al valor de las constantes que defina
- Sus nombres van en mayúsculas
- No se pueden confundir las constantes con los miembros estáticos de una clase.

2.4 Métodos 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 usando la palabra `static.`

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.

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. Se llaman desde la clase. No es posible llamarlos desde un objeto y por tanto, no podemos usar `$this` dentro de un método estático.

```php
<?php

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.

```php
Producto::nuevoProducto();
```

- Si es privado, solo se puede 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

```php
self::$num_productos ++;
```

2.5 Métodos constructores y destructores

En PHP podemos definir en las clases métodos constructores que se ejecutan cuando se crea el objeto. El constructor de una clase debe llamarse `__construct.`

Se puede usar para asignar valores a atributos.

```php
<?php

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, en cuyo caso deberán pasarse cuando se crea el objeto. Sin embargo, **solo puede haber un método constructor en cada clase.**

En el siguiente ejemplo, definimos un constructor en el que haya que pasar el código, siempre que instanciemos un nuevo objeto de esa clase tendremos que indicar su código.

```php
<?php

class Producto {
  private static $num_productos = 0;
  private $codigo;

  public function __construct($codigo) {
      $this->$codigo = $codigo;
      self::$num_productos++;
  }
  ...
}
$p = new Producto('GALAXYS')

?>
```

También es posible definir un método destructor, que debe llamarse `__destruct` y permite definir acciones que se ejecutarán cuando se elimine el objeto.

```php
<?php

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');

?>
```

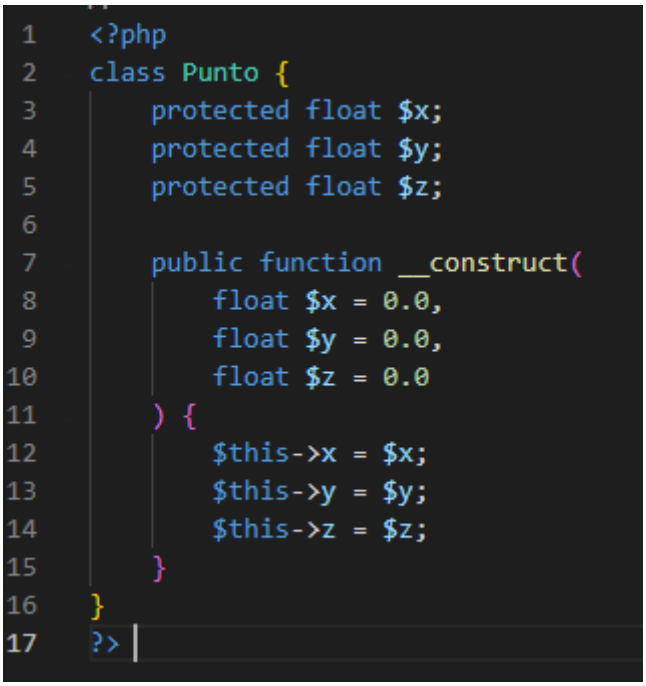
2.5.1 Constructores en PHP 8

Una de las grandes novedades que ofrece PHP 8 es la simplificación de los constructores con parámetros, lo que se conoce como **promoción de las propiedades del constructor**.

Para ello, en vez de tener que declarar las propiedades como privadas o protegidas, y luego dentro del constructor tener que asignar los parámetros a estas propiedades, el propio constructor promociona las propiedades.

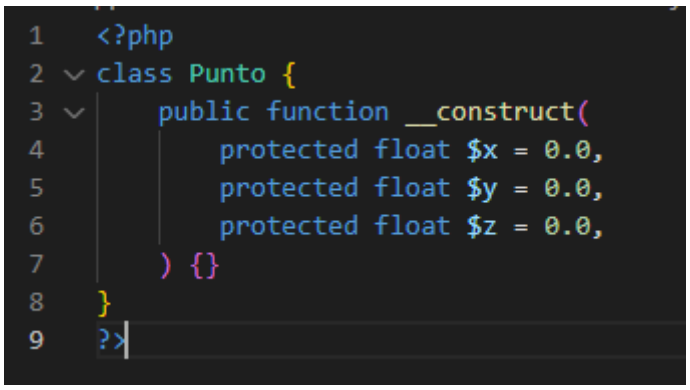
Ejemplo: Imaginemos una clase Punto donde queramos almacenar sus coordenadas:

En versiones anteriores a PHP 8:



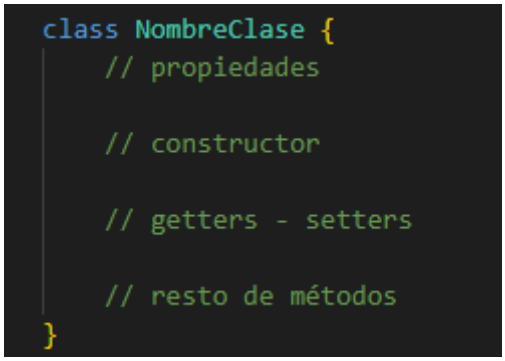
```php
<?php
class Punto {
    protected float $x;
    protected float $y;
    protected float $z;

    public function __construct(
        float $x = 0.0,
        float $y = 0.0,
        float $z = 0.0
    ) {
        $this->x = $x;
        $this->y = $y;
        $this->z = $z;
    }
}
?>
```

En PHP 8, quedaría del siguiente modo, mucho más corto:



```php
<?php
class Punto {
    public function __construct(
        protected float $x = 0.0,
        protected float $y = 0.0,
        protected float $z = 0.0,
    ) {}
}
?>
```

IMPORTANTE: EL ORDEN



```php
class NombreClase {
    // propiedades

    // constructor

    // getters - setters

    // resto de métodos
}
```

3. Utilización de objetos

Ya hemos visto cómo instanciar un objeto utilizando `new`. Y cómo acceder a sus métodos y atributos públicos con el operador flecha.

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

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

```php
<?php

if ($p instanceof Producto) {
    ...
}

?>
```

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

Funciones de utilidad para objetos y clases en PHP

Función	Significado	Ejemplo
get_class	Devuelve el nombre de la clase del objeto	`echo "La clase es: " . get_class($p);`

class_exists	Devuelve true si la clase está definida o false en caso contrario	`if (class_exists('Producto') {`
		`$p = new Producto();`
		`…`
		`}`

get_declared_classes	Devuelve un array con los nombres de las clases definidas	`print_r(get_declared_classes());`
class_alias	Crea un alias para una clase	`class_alias('Producto', 'Articulo');` `$p = new Articulo();`
get_class_methods	Devuelve un array con los nombres de los métodos de una clase que son accesibles desde dónde se hace la llamada	`print_r(get_class_methods('Producto'));`

method_exists	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	`if (method_exists('Producto', 'vende') {`
		`…`
		`}`

get_class_vars	Devuelve un array con los nombres de los atributos de una clase que son accesibles desde donde se hace la llamada	`print_r(get_class_vars('Producto'));`
get_object_vars	Devuelve un array con los nombres de los métodos de un objeto que son accesibles desde donde se hace la llamada	`print_r(get_object_vars($p));`

property_exists	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	`if (property_exists('Producto', 'codigo') {`
		`…`
		`}`

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.

```php
<?php

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 solo funciona con objetos (y a partir de PHP 5.1 también con arrays).

Una característica de la POO en PHP que hay que tener en cuenta es qué sucede con los objetos cuando los pasas a una función o simplemente cuando se ejecuta un código como el siguiente:

```php
<?php

$p = new Producto();
$p->nombre = 'Samsung Galaxy S';
$a = $p;

?>
```

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. Luego, el código anterior quedaría:

```php
<?php

$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 y, por tanto, quizás no tenga sentido copiarlo al crear un nuevo objeto. Si este 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.

```php
<?php

class Producto {
    ...
    public function __clone($atributo) {
        $this->codigo = nuevo_codigo();
    }
    ...
}

?>
```

Relación exacta entre dos objetos

A veces tienes dos objetos y quieres saber su relación exacta. En PHP, como ya vimos se utiliza == 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.

```php
<?php

$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

?>
```

Si utilizas el operador ===, el resultado de la comparación será true solo cuando las dos variables sean referencias al mismo objeto.

```php
<?php

$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.

?>
```

4. Mecanismos de mantenimiento del estado

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 `seralize`.
El resultado obtenido es un `string` que contiene un flujo de bytes, en el que se encuentran definidos todos los valores del objeto.

```php
<?php

$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.`

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

Las funciones `serialize` y `unserialize` se utilizan muchos 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:

```php
<?php

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`.

```php
<?php

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`.

```php
<?php

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

En PHP además tienes la opción de personalizar el proceso de serialización y deserialización de un objeto, utilizando los métodos mágicos `__sleep` y `__wakeup`.
Para más información:
https://www.php.net/manual/es/language.oop5.magic.php#language.oop5.magic.sleep

5. Herencias

La herencia es un mecanismo de la POO que nos permite definir nuevas clases con 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.

```php
<?php

class Producto {
    public $codigo;
    public $nombre;
    public $nombre_corto;
    public $PVP;

    public function muestra() {
        print "<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.

Si queremos 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, Ordenador, Movil.`

```php
<?php

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 operdor `instanceof.`

```php
<?php

$t = new TV();
if ($t instanceof Producto) {
    // Este código se ejecuta pues la condición es cierta
…
}

```

Antes hemos visto algunas funciones útiles para programar utilizando objetos y clases.
A continuación, vamos a ver algunas funciones relacionadas con la herencia.

Función	Significado	Ejemplo
`get_parent_class`	Devuelve el nombre de la clase padre del objeto o la clase que se indica	echo "La clase padre es: " . get_parent_class($t);
`is_subclass_of`	Devuelve true si el objeto o la clase del primer parámetro, tiene como clase base a la que se indica el segundo parámetro, o false en caso contrario	if (is_subclass_of($t, 'Producto') {

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.

```php
<?php

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 utilizar la palabra `final`. Si en nuestro ejemplo, hubiéramos hecho:

```php
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.

```php
final class Producto {
    ...
}
```

Opuestamente al modificador `final`, existe también `abstract`. Se utiliza de la misma forma, tanto con métodos como 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 utilizase de clase base y subclases sí podrán utilizarse para instanciar objetos.

```php
abstract class Producto {
    ...
}
?>
```

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

```php
class Producto {
    ;

    abstract public function muestra();
}
```

NOTA: No se puede declarar una clase como `abstract` y `final` simultáneamente.
`abstract` obliga a que se herede para que se pueda utilizar
`final` indica que no se podrá heredar.

EJEMPLO

Veamos un ejemplo con todo esto que hemos visto.
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.

```php
<?php

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'];
    }
}
```

¿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 al constructor de la clase base.
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 ámbito.

```php
<?php

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'];
    }
}
?>
```

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`.

6. Interfaces

Una **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` u `Ordenador`. También hemos visto que en las subclases podías redefinir el comportamiento del método muestra para que generara una salida en HTML diferente para cada tipo de producto.

Si quieres asegurarte de que todos los tipos de productos tengan un método `muestra`, puedes crear una interface como el siguiente

```
3   interface iMuestra {
4       public function muestra();
5   }
6
```

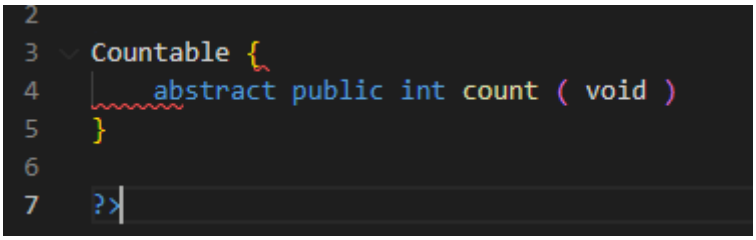
Y cuando crees las subclases deberás indicar con la palabra `implements` que tienen que implementar los métodos declarados en esta interface.

```
1   <?php
2
3   class TV extends Producto implements iMuestra {
4       ...
5       public function muestra() {
6           print "<p>" . $this->pulgadas . " pulgadas</p>";
7       }
8       ...
9   }
10
```

NOTA: Todos los métodos que se declaren en una interface deben ser públicos. Además de métodos, las interfaces podrán contener constantes, pero no atributos.

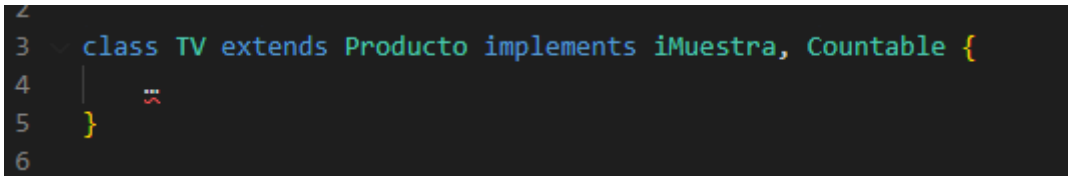
Una interface es como un contrato que la clase debe cumplir. Al implementar todos los métodos declarados en la interface se asegura la interoperabilidad entre clases. Si sabes que una clase implementa una interface determinada, sabes qué nombre tienen sus métodos, qué parámetros les debes pasar y, probablemente,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

Por ejemplo, en la librería de PHP está definido la interface `Countable`.



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

Antes vimos que en PHP una clase solo puede heredar de otra.
En PHP 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`.



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

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`.

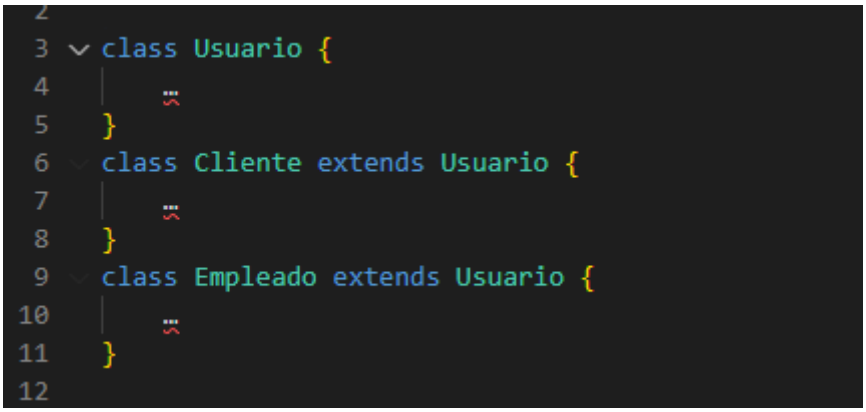
6.1 Interfaces o clases abstractas

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:

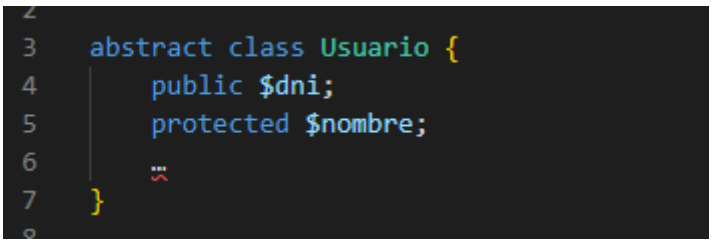
- En las clases abstractas, los métodos pueden contener código. Si van a existir varias subclases con un comportamiento común, se podría programar en los métodos de la clase abstracta. Si se opta por una interface, habría que repetir el código en todas las clases que lo implemente.
- Las clases abstractas 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`.



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

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



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.

Función	Significado	Ejemplo
`get_declared_interfaces`	Devuelve un array con los nombres de los interfaces declarados.	print_r (get_declared_interfaces());
`Interface_exists`	Devuelve true si existe la interface que se indica, o false en caso contrario	if (interface_exists('iMuestra') { … }

7. Ejemplo de POO en PHP

Es hora de llevar a la práctica lo que has aprendido:

- `Producto`. Las instancias de esta clase representan los productos que se venden en la tienda (Familias)
- `SubProducto`. Las instancias de esta clase representan los `SubProductos` que se venden en la tienda.

Las instancias de la clase Producto van a almacenar la siguiente información de cada producto: `códigoProd, nombreProd, nombre_corto_prod`. Cada valor se almacenará en un atributo de tipo protected, para limitar el acceso a su contenido y, a la vez, permitir su herencia. Además, en nuestra aplicación será necesario cambiar sus valores y acceder a ellos, por lo que se creará un método de tipo `get` y `set` para cada uno.

Las instancias de la clase `SubProducto (herencia de Producto)` van a almacenar la siguiente información de cada producto: código, nombre, PVP.
Cada valor se almacenará en un atributo de tipo protected. Además, en nuestra aplicación será necesario cambiar sus valores y acceder a ellos, por lo que se creará un método tipo get y set para cada uno.

Por último, vamos a crear también un método para mostrar el código del producto y otro para mostrar el precio del subproducto. Crearemos un fichero elpProducto.php

```php
<?php
class Producto {
    protected $codigoProd;
    protected $nombreProd;
    protected $nombre_corto_prod;

    public function getcodigo() {return $this->codigoProd; }
    public function getnombre() {return $this->nombreProd; }
    public function getnombrecorto() {return $this->nombre_corto_prod; }


    //Set
    function setCodigo($codigoProd) {$this->codigoProd=$codigoProd; }
    function setNombreProd($nombreProd) {$this->nombreProd=$nombreProd; }
    function setNombreCortoProd($nombre_corto_prod) {$this->nombre_corto_prod=$nombre_corto_prod; }

    public function muestra() {print "El producto es: ".$this->codigoProd."";}

    public function __construct($row) {
        $this->codigoProd = $row['codigo'];
        $this->nombreProd = $row['nombre'];
        $this->nombre_corto_prod = $row['nombre_corto'];
    }

}

class SubProducto extends Producto {
    protected $codigo;
    protected $nombre;
    protected $PVP;
    //get
    public function getcodigo() {return $this->codigo; }
    public function getnombre() {return $this->nombre; }
    public function getPVP() {return $this->PVP; }

    //set
    function setCodigo($codigo) {$this->codigo=$codigo; }
    function setNombre($nombre) {$this->nombre=$nombre; }
    function setPVP($PVP) {$this->PVP=$PVP; }

    public function muestraSubProducto() {print "El SubProducto es: ".$this->codigo." y el precio es: ".$this->PVP."";}

    public function __construct($producto,$codigo,$nombre,$PVP) {
        parent::__construct($producto); //El constructor Padre
        $this->codigo = $codigo;
        $this->nombre = $nombre;
        $this->PVP = $PVP;
    }
}
```

Tras crear el fichero anterior, crearemos uno nuevo (elpMostrarProductos.php) en el que mostraremos el nombre del producto (desde el objeto del subproducto) y el precio de los subproductos

```php
<?php

    include_once('elpProducto.php');
    $producto = array(
        "codigo" => "P0",
        "nombre" => "Alimentacion",
        "nombre_corto" => "ALI",
    );
    $prod1= new Producto($producto);

    $subprodA= new SubProducto($producto,'P0-1','Manzanas','12.5');
    $subprodB= new SubProducto($producto,'P0-2','Tomates','6');
    $subprodC= new SubProducto($producto,'P0-3','Arroz','1.5');

    $subprodA->muestra();
    $subprodA->muestraSubProducto();
    $subprodB->muestraSubProducto();
    $subprodC->muestraSubProducto();

    ?>
```

El resultado en pantalla será el siguiente:

El producto es: P0El SubProducto es: P0-1 y el precio es: 12.5El SubProducto es: P0-2 y el precio es: 6El SubProducto es: P0-3 y el precio es: 1.5

8. Programación en capas

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, que introdujimos en la unidad 1, es el patrón diseño Modelo – Vista – Controlador (MVC). Este patrón pretende dividir el código en 3 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.

Aunque puedas programar utilizando MVC por tu cuenta, es más habitual utilizar el patrón MVC en conjunto con un framework o marco de desarrollo.
Existen numerosos frameworks disponibles en PHP, muchos de los cuales incluyen soporte para MVC.
Lo veremos más adelante con el framework **Laravel.**