1. Introducción

El lenguaje PHP original no se diseñó con características de orientación a objetos

Las características de POO que se soportan a partir de la versión PHP 5 (estamos en la 8) incluyen:

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

No se incluye:

- Herencia múltiple (Java NO tiene)
- Sobrecarga de métodos (Java SÍ tiene)
- Sobrecarga de operadores (Java NO tiene)

2. Las clases. Usos 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.

```
class Producto
{
    private int $codigo;
    public string $nombre;
    protected float $PVP;

    public function __toString()
    {
        return "{$this->nombre}";
    }
```

```
}
```

En el ejemplo hemos puesto para recordar los principales niveles de acceso de los atributos, recordad que por lo general los atributos deben ser privados

Recomendaciones:

- Cada clase en un fichero distinto
- Los nombres de las clases deben comenzar por mayúsculas para distinguirlos de los objetos y otras variables

3. Objetos. Declaración y usos

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.class.php');
```

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

```
$p->nombre ="Samsung Galaxy S23";
echo $p;
```

Ejercicio: Crea la clase Producto con los atributos que consideres necesarios.

4. Constructores

Podemos definir en las clases **métodos constructores**, que se ejecutan cuando se crea el objeto.

El constructor de una clase debe llamarse __construct. Se pueden utilizar, por ejemplo, para asignar valores a atributos.

```
    public function __construct()
    {
    }
```

4.1. Operador 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 código anterior para tener acceso a los **atributos privados del objeto** (que sólo son accesibles desde los métodos de la clase).

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

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, **sólo puede haber un método constructor en cada clase**

```
    public function __construct($codigo, $nombre, $precio)
    {
        $this->codigo = $codigo;
        $this->nombre = $nombre;
        $this->PVP = $precio;
    }
```

Ejercicio: Crea un constructor para tu clase Producto.

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

> Hoja04_POO_01

5. Métodos get y set

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

```
    public function setCodigo($nuevoCodigo)
    {
        $this->codigo =$nuevoCodigo;

    }

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

Ejercicio: crea los métodos get y set para la clase Producto creada por ti.

6. Constantes

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

No hay que confundir los atributos con las constantes.

Las constantes no pueden cambiar su valor (de ahí su nombre), no usan elcarácter **$** 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 BaseDatos{
    const USUARIO = "dwes";
    ...
}

echo BaseDatos::USUARIO;
```

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

Ejercicio: crea una clase BaseDatos que tendrá como constantes el dominio donde está alojada, el usuario y la contraseña y la base de datos a utilizar.

7. Métodos estáticos

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 $numProductos = 0;
    public static function nuevoProducto(){
        self::$numProductos++;
    }
...
}
```

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 $numProductos 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::$numProductos++;
```

Se puede aumentar el atributo estático en el constructor y decrementarlo en el destructor:

```
class Producto{
    private static $numProductos = 0;
    private $codigo;
    public function __construct($codigo){
        $this->codigo = $codigo;
        self::$numProductos++;
    }
    public function __destruct(){
        self::$numProductos--;
    }
    public static function getNumeroProductos(){
        return self::$numProductos;
    }
}

$p = new Producto('Iphone');
echo Producto::getNumeroProductos();
```

8. Utilización de objetos

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){
    ...
}
```

Se incluyen una serie de funciones útiles para el desarrollo de aplicaciones utilizando POO.

Función	Ejemplo	Significado

Función	Ejemplo	Significado
get_class	echo "La clase es:".get_class($p);	Devuelve el nombre de la clase del objeto
class_exists	if (class_exits('Producto')){ $p= new Producto(); ...}	Devuelve true si la clase está definida o false en caso contrario
class_alias	class_alias('Producto','Articulo'); $p = new Articulo();	Crea un alias para una clase
get_class_methods	print_r(get_class_methods('Producto'));	Devuelve un array con los nombres de los métodos de una clase que son accesibles desde donde se hace la llamada

Función	Ejemplo	Significado
method_exists	if (method_exists ('Producto','vende')) { ....}	Devuelve true si existe el método en el objeto o la clase que se indica, o false en caso contrario, independientemente de si es accesible o no
property_exits	if(property_exists('Producto','codigo')) {...}	Devuelve true si existe el atributo en el objeto o la clase que se indica, o false en caso contrario, independientemente de si es accesible o no

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

```
public function vendeProducto(Producto $p){
    ...
}
```

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

Pregunta: ¿Qué sucede al ejecutar el siguiente código?

```
$p = new Persona();
$p->nombre ='Pepe';
$a=$p;
```

Desde PHP5 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.

Aunque haya dos o más identificadores del mismo objeto, en realidad **todos se refieren a la única copia que se almacena del mismo**.

Tiene un comportamiento similar al que ocurre en Java.

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 = 'Xiaomi 13';
$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(){
        $this->codigo = nuevoCodigo();
    }
}
```

A veces tienes dos objetos y quieres saber su relación exacta. Para eso, 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.

```
$p = new Producto();
$p->nombre = 'Xiaomi 13';
$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 = 'Xiaomi 13';
```

```
$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.
```

9. Herencia. Redefinición de métodos

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 si tenemos la clase Persona, podemos crear las subclases Alumno y Profesor

```
class Persona{
    protected $nombre;
    protected $apellidos;
    public function __toString(){
        print "<p>".$this->nombre.",".$this->apellidos."</p>";
    }
}
```

Esto puede ser útil si todas las personas sólo tuviesen nombre y apellidos, pero los alumnos tendrán un conjunto de notas y los profesores tendrán, por ejemplo, un número de horas de docencia.

```
class Alumno extends Persona{
    private $notas;
}
```

Ejercicio : Codificar estas dos clases y crear un objeto de tipo Alumno. Comprobar mediante el operador instanceof si el objeto es de tipo Persona o de tipo Alumno

Podemos utilizar las siguientes funciones:

- **get_parent_class**: devuelve el nombre de la clase padre del objeto o la clase que se indica
- **is_subclass_of**: devuelve true si el objero o la clase del primer parámetro, tiene como clase base a la que se indica en el segundo parámetro, o false en caso contrario

```
echo "La clase padre es:".get_parent_class($p);

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.

```
class Profesor extends Persona{
    private $horas;
    public function __toString(){
        return "<p>".$this->nombre.",".$this->apellidos.":".$this->horas."
</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 Persona{
    protected $nombre;
    protected $apellidos;
    public final function __toString(){
        return "<p>".$this->nombre.",".$this->apellidos."</p>";
    }
}
```

En este caso el **método toString** no podría redefinirse en la clase Profesor.

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

```
final class Persona{
    ...
}
```

Opuestamente al **modificador final** existe también el **modificador abstract**. Se utiliza de la misma forma, tanto con métodos como con clases completas, pero en lugar de prohibir la herencia, obliga aque 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 Persona{
    ...
}
```

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

```
 class Persona{
    ...
    abstract public function mostrar();
}
```

Ejercicio: Crea un constructor para la clase Persona. ¿Qué pasará ahora con la clase Alumno, que hereda de Persona? Cuando crees un nuevo objeto de esa clase, ¿se llamará al constructor de Persona?¿Puedes crear un nuevo constructor específico para Alumno que redefina el comportamiento de la clase base?

Desde PHP7,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 Alumno extends Persona{
    private $notas;
    public function **construct($nombre,$apellidos,$notas){
        parent::**construct($nombre,$apellidos);
        $this->notas=$notas;
    }
}
```

La utilización de la palabra **parent** es similar a **self**. Al utilizar parent haces referencia a la clase base de la actual

> Hoja04_POO_02

10. Interfaces

Un interface es similar a 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 Persona como Profesor o Alumno. 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 persona.

Si quieres asegurarte de que todos los tipos de persona 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 Profesor extends Persona implements iMuestra{
    …
    public  function muestra(){
    print "<p>".$this->nombre .": ".$this->horas."</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**

```
interface Countable{
    abstract public int count ();
}
```

Desde PHP7, es posible crear clases que implementen varios interfaces, simplemente separando la lista de interfaces por comas después de la palabra **implements**.

```
class Profesor extends Persona 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**.

Desde PHP 7 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**.

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.

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

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

> Hoja04_POO_03