UNIDAD 4: Tarea

Programación orientada a objetos en PHP

Actividades de OOP, MVC y otros

1. Repasa la POO y detalla sus conceptos y características principales.

2. Haz este tutorial desde aquí hasta aquí de POO; para ello, ejecuta todos sus ejemplos y explícalos.

3. Resuelve estos diez ejercicios sobre clases usando POO en PHP.
    - 3.1. Clase Punto
    - 3.2. Clase Linea
    - 3.3. Clase Rectangulo
    - 3.4. Clase Circulo
    - 3.5. Clase Estudiante
    - 3.6. Función que reciba dos parámetros de entrada de tipo clase Punto y que devuelva la distancia euclídea entre esos dos puntos
    - 3.7. Clase Linea2D
    - 3.8. Clase Forma
    - 3.9. Función que reciba un parámetro de entrada de tipo array/lista cuyos elementos sean de tipo array/lista de clase Estudiante y que devuelva el índice del array/lista grupos cuyo promedio de notas del grupo de estudiantes sea el más alto
    - 3.10. Crea una clase C que herede de una clase B y que la clase B herede de A. La clase C heredará todos los métodos y atributos de B y B de A. Como mínimo, una función de A, de B y de C tienen que tener el mismo nombre pero que hagan cosas distintas

4. Define programación por capas y sus capas como BLL (lógica de negocio) o DAL y detalla la three-tier.

Consiste en una arquitectura cliente-servidor cuyo objetivo es dividir la presentación, el procesamiento de la aplicación y la gestión de los datos.

La arquitectura más extendida divide la aplicación en tres capas (three-tier):

5. Lee exhaustivamente y explica con tus propias palabras lo que es un patrón de diseño software.

6. Lee exhaustivamente y explica con tus propias palabras lo que es el patrón de diseño software MVC.

7. Mira este vídeo de principio a fin, resúmelo y explica el flujo de MVC, empezando por el usuario.

8. Lee bien este y este artículo y ejecuta, modifica y explica exhaustivamente esta plantilla MVC básica.

9. Lee exhaustivamente y explica con tus propias palabras lo que es un framework web.

10. Lee exhaustivamente y explica con tus propias palabras lo que es un Laravel.

Ejercicios realizados por José Carlos López Henestrosa

1. Repasa la POO y detalla sus conceptos y características principales.

La Programación Orientada a Objetos (Object-oriented Programming en inglés) es un **paradigma de programación** que encapsula el código de un programa en clases. Dentro de estas clases, podemos declarar sus propiedades y sus métodos, los cuales tendrán un alcance determinado (público, privado, protegido…).

Dichas propiedades (estado) podrán ser modificadas por sus métodos (comportamiento). Usualmente, cada propiedad podrá ser obtenida mediante el método get y modificada por el método set.

Algunos lenguajes orientados a objetos son Java, C#, C++ o Ruby.

Los **conceptos principales** del paradigma de la programación orientada a objetos son los siguientes:

- Clase: Es una plantilla que contiene propiedades y métodos de un tipo de objeto. Dicho objeto se genera mediante la instanciación de una clase. Por ejemplo, si tenemos una clase *Coche* en Java y queremos crear un objeto de la clase *Coche* para que cuente con las propiedades y métodos definidos en la clase *Coche*, tendremos que hacer lo siguiente: `Coche coche = new Coche();`
- Herencia: Permite que una clase (A) integre las propiedades y métodos definidos en otra clase (B). Por lo tanto, el resultado es como si las propiedades y los métodos de la clase B heredados de la clase A hubiesen sido definidos por la clase B. Es importante matizar que la clase B heredará aquellas propiedades y métodos declarados con alcance *público* en A.
- Objeto: Es la instancia de una clase. Cada objeto declarado es independiente el uno del otro, por lo que, por ejemplo, `Coche coche1 = new Coche();` y `Coche coche2 = new Coche();` referencian a espacios en memoria distintos pese a ser instancias de la misma clase.
- Método: Son algoritmos asociados a un objeto que son capaces de modificar el estado (las propiedades) de dicho objeto. El conjunto de los métodos de un objeto determinan el comportamiento general del objeto.
- Evento: Son aquellas acciones que generan una respuesta en el programa. Por ejemplo, un click sobre un botón.
- Propiedades: Características definidas en una clase.
- Estado interno: Es una propiedad declarada con el alcance *privado* que puede ser alterada por un método del objeto y que indica posibles situaciones del objeto. Un ejemplo de ello podrían ser los flags de estado que varían el algoritmo ejecutado por un método dependiendo del estado de la clase del objeto.

Las **características** de la POO son las siguientes:

- Abstracción: Una clase abstracta se podría considerar como una declaración de la estructura de una clase pero no de la funcionalidad de la misma, puesto que el cuerpo de los métodos (también abstractos) de la clase abstracta están vacíos. Esto se realiza cuando, por ejemplo, sabemos que dos clases tienen en común ciertos comportamientos que se realizan de una forma u otra, dependiendo de la clase.

  Por ejemplo, si tenemos una clase que carga una lista, la cual no sabemos qué tipo de información va a tener, podremos definir una clase abstracta *Lista* que contendrá métodos que serán implementados por todas aquellas clases que hereden de la clase *Lista*.

- Encapsulamiento: Implica que el valor de las propiedades (estados) de un objeto solo puedan ser cambiados mediante los métodos internos de la clase. Genéricamente, el encapsulamiento implica que las propiedades tengan un alcance *privado* y los métodos un alcance *público.* Con ello, evitamos que los atributos tomen valores inconsistentes y/o no controlados.

- Polimorfismo: Son comportamientos diferentes, asociados a objetos distintos, que pueden compartir el mismo nombre. Al llamarlos por ese nombre, se utilizará el comportamiento correspondiente al objeto que se esté usando.

  Por ejemplo, si tenemos una clase que se llama *Perro* y otra *Humano* en la que ambas contienen la propiedad *nombre*, sabremos que al acceder a dicha propiedad desde una instancia de la clase *Humano* obtendremos el valor de la propiedad *nombre* del objeto *Humano.* Es decir, gracias al polimorfismo podremos repetir el nombre de las propiedades y/o métodos de distintas clases ya que la referencia a dichos atributos dependerá del objeto con el que estemos accediendo a ellos.

- Herencia: Gracias a esta característica, podremos organizar jerarquías de clasificación en la que las clases heredarán las propiedades y los métodos de la clase de la que estén extendiendo. Organiza y facilita el polimorfismo y el encapsulamiento.

- Modularidad: Permite dividir un programa en partes más pequeñas (módulos), los cuales deben de ser tan independientes como sea posible. Por ejemplo, en Java, los módulos están organizados en paquetes. Hay distintos patrones de diseño que definen distintos módulos dependiendo del tipo de programa que estemos realizando, pero uno de los más populares es el de Modelo, Vista y Controlador (MVC) en el que cada uno se encargará de gestionar distintas partes de la aplicación.

- Ocultación: Suele estar muy relacionado con el encapsulamiento. Consiste en proteger a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas. Esto ocurre cuando declaramos una propiedad con un alcance *privado*.

- Recolección de basura: Libera espacio en memoria ocupado por objetos que han quedado sin ninguna referencia a ellos. Esto implica que el programador no tiene que preocuparse de liberar recursos de la memoria, ya que será el propio entorno el que decida cuándo liberarlos.

  La liberación de memoria por parte del programador se da en el paradigma de programación imperativo/procedimental (como en el lenguaje C). No obstante, hay lenguajes híbridos, como el superset de C: C++, que se extendieron para dar soporte al paradigma de programación orientada a objetos, en la que los programadores tendrán que desasignar manualmente el espacio en memoria.

2. Haz este tutorial desde aquí hasta aquí de POO; para ello, ejecuta todos sus ejemplos y explícalos.

CLASSES/OBJECTS

```php
<?php
// Definición de clase
class Fruit {
  // Propiedades
  public $name;
  public $color;

  // Métodos
  function set_name($name) {
    // La palabra reservada $this hace referencia al objeto
actual. Solo se puede utilizar dentro de los métodos.
    $this->name = $name;
  }
  function get_name() {
    return $this->name;
  }
}
?>
```

```php
<?php
// Instanciación de objetos de la clase
$apple = new Fruit();
$banana = new Fruit();
// Las propiedades se modifican mediante el método set
$apple->set_name('Apple');
$banana->set_name('Banana');
// Obtenemos el estado mediante el método get
echo $apple->get_name();
echo "<br>";
echo $banana->get_name();
?>
```

```php
<?php
$apple = new Fruit();
// La palabra reservada instanceof comprueba si un objeto es
instancia de una clase determinada. En este caso, nos devolvería
el valor true ya que $apple es instancia de la clase Fruit.
var_dump($apple instanceof Fruit);
?>
```

CONSTRUCTOR

```php
<?php
class Fruit {
  public $name;
  public $color;

  // El método __construct() es llamado, automáticamente, por PHP al crear la instancia de una clase. Esto nos permitirá inicializar las propiedades de un objeto en su creación, sin necesidad de tener que invocar los métodos set.
  function __construct($name, $color) {
      $this->name = $name;
      $this->color = $color;
  }
  function get_name() {
    return $this->name;
  }
  function get_color() {
    return $this->color;
  }
}

// Ejemplo de creación de un objeto usando el constructor de la clase. Como podemos ver, no se hace uso del método set.
$apple = new Fruit("Apple", "red");
echo $apple->get_name();
echo "<br>";
echo $apple->get_color();
?>
```

DESTRUCTOR

PHP también tiene el método destructor, el cual es similar al usado en otros lenguajes orientados a objetos, como C++. El método será llamado cuando no haya ninguna referencia a un objeto particular o durante la secuencia de apagado (registrada con register_shutdown_function).

```php
<?php
class Fruit {
  public $name;
  public $color;

  function __construct($name, $color) {
      $this->name = $name;
      $this->color = $color;
  }
```

```
  function __destruct() {
    echo "Destroying " . __CLASS__ . "\n";
  }
}

$apple = new Fruit("Apple", "red");
?>
```

ACCESS MODIFIERS

```
<?php
class Fruit {
  // public es el modificador por defecto. Hace que la
propiedad/método pueda ser accedida desde cualquier punto del
programa.
  public $name;
  // protected permite que las propiedades/métodos sean accedidas
solo desde dentro de la clase o por subclases derivadas de la
clase.
  protected $color;
  // private permite que las propiedades/métodos sean accedidas
solo desde dentro de la clase.
  private $weight;
}

$mango = new Fruit();
$mango->name = 'Mango'; // OK
$mango->color = 'Yellow'; // ERROR
$mango->weight = '300'; // ERROR
?>
```

INHERITANCE

Como hemos visto en las características del paradigma de la programación orientada a objetos, la herencia ocurre cuando una clase se extiende de otra. En PHP, se utiliza la palabra reservada `extends` para aplicarla.

```
<?php
class Fruit {
  public $name;
  public $color;

  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
```

```
  protected function intro() {
    echo "The fruit is {$this->name} and the color is
{$this->color}.";
  }
}


class Strawberry extends Fruit {
  public function message() {
    echo "Am I a fruit or a berry? ";
    // OK. Se está llamando al método intro() desde dentro de una
clase derivada de la clase Fruit
     $this->intro();
  }
}

// La clase Strawberry también hereda el constructor del padre.
$strawberry = new Strawberry("Strawberry", "red"); // OK.
$strawberry->message(); // OK. message() es public y llama al
método intro() (protected) desde el interior de la clase derivada.
$strawberry->message();
$strawberry->intro(); // Error. intro() es protected y no se puede
acceder desde fuera de la clase o de una subclase que extienda de
la clase.
?>
```

Para sobreescribir métodos, tenemos que redefinir los métodos (usar el mismo nombre) en la clase hija.

```
<?php
class Fruit {

  public $name;
  public $color;

  public function __construct($name, $color) {
     $this->name = $name;
     $this->color = $color;
  }

  public function intro() {
    echo "The fruit is {$this->name} and the color is
{$this->color}.";
  }
}

class Strawberry extends Fruit {
```

```
  public $weight;

  public function __construct($name, $color, $weight) {
      $this->name = $name;
      $this->color = $color;
      $this->weight = $weight;
  }

  public function intro() {
    echo "The fruit is {$this->name}, the color is {$this->color},
and the weight is {$this->weight} gram.";
  }
}

$strawberry = new Strawberry("Strawberry", "red", 50);
$strawberry->intro();
?>
```

La palabra reservada `final` se usa para evitar que la clase sea heredada por otra clase. También evita que sus métodos sean sobreescritos.

Para evitar herencia de la clase:

```
<?php
final class Fruit {
  // código
}

// Error
class Strawberry extends Fruit {
  // código
}
?>
```

Para evitar sobreescritura de métodos:

```
<?php
class Fruit {
  final public function intro() { // código }
}

class Strawberry extends Fruit {
  // Error
  public function intro() { // código }
}
?>
```

CONSTANTS

El valor de las variables constantes no puede ser modificado tras su declaración. Es recomendable que el nombre de las variables estén en mayúsculas para ser diferenciadas del resto de variables. Para que un objeto acceda a ellas, tenemos que usar el operador `::` seguido del nombre de la constante.

```
<?php
class Goodbye {
  const LEAVING_MESSAGE = "Thank you for visiting W3Schools.com!";
}

echo Goodbye::LEAVING_MESSAGE;
?>
```

Para acceder a una constante desde el interior de una clase, tendremos que usar la palabra reservada `self`.

```
<?php
class Goodbye {
  const LEAVING_MESSAGE = "Thank you for visiting W3Schools.com!";
  public function byebye() {
    echo self::LEAVING_MESSAGE;
  }
}

$goodbye = new Goodbye();
$goodbye->byebye();
?>
```

ABSTRACT CLASSES

Como ya vimos en las características del paradigma de programación orientado a objetos, las clases y los métodos abstractos declaran el nombre de un método sin determinar su funcionalidad, la cual quedará a manos de la clase(s) hija(s).

Una clase abstracta es una clase que, al menos, contiene un método abstracto. Un método abstracto tiene el nombre declarado pero no tiene código implementado.

```
<?php
abstract class ParentClass {
  abstract public function someMethod1();
  abstract public function someMethod2($name, $color);
  abstract public function someMethod3() : string;
}
?>
```

Para implementar el código de un método abstracto en una clase que extienda de la clase abstracta, tendremos que definir un método con el mismo nombre que el método

abstracto a implementar y con un modificador de acceso **igual** o **menos** restrictivo que el declarado. Por ejemplo, si el método abstracto está definido como *protected*, la clase hija puede definirlo como *public* o *protected*, pero nunca como *private*. Opcionalmente, la clase hija puede añadir parámetros al método, pero nunca quitar.

```php
<?php
// Clase padre
abstract class Car {
  public $name;
  // El constructor no es abstracto, por lo que no es necesario
sobreescribirlo.
  public function __construct($name) {
      $this->name = $name;
  }
  abstract protected function intro() : string;
}

// Clases hijas
class Audi extends Car {
  public function intro($country) : string {
    return "Choose $country quality! I'm an $this->name!";
  }
}

class Volvo extends Car {
  public function intro($country) : string {
    return "Proud to be $country! I'm a $this->name!";
  }
}

class Citroen extends Car {
  public function intro($country) : string {
    return "$country extravagance! I'm a $this->name!";
  }
}
// Creamos objetos de las clases hijas
$audi = new audi("Audi");
echo $audi->intro("German");

$volvo = new volvo("Volvo");
echo $volvo->intro("Swedish");

$citroen = new citroen("Citroën");
echo $citroen->intro("French"); ?>
```

INTERFACES

Las interfaces nos permiten especificar los métodos que una clase debe implementar. Se declaran con la palabra reservada `interface` y se implementan en una clase con la palabra reservada `implements`

```php
<?php
interface InterfaceName {
  public function someMethod1();
  public function someMethod2($name, $color);
  public function someMethod3() : string;
}
?>
```

Se diferencian de las clases abstractas en los siguientes rasgos:

- Las interfaces no pueden tener propiedades, mientras que las clases abstractas sí.
- Todos los métodos de una interfaz tienen que ser *public*, mientras que los métodos de una clase abstracta son *public* o *protected*.
- Todos los métodos en una interfaz son abstractos, por lo que el cuerpo de los métodos de una interfaz tendrán que estar vacíos. No es necesario declararlos con la palabra reservada `abstract`.
- Las clases pueden implementar (`implements`) una interfaz y, a su vez, extender de otra clase (`extends`).

Una clase que implemente una interfaz tendrá que implementar **todos** los métodos de la interfaz.

```php
<?php
interface Animal {
  public function makeSound();
}

class Cat implements Animal {
  public function makeSound() {
    echo "Meow";
  }
}

class Dog implements Animal {
  public function makeSound() {
    echo " Bark ";
  }
}

class Mouse implements Animal {
  public function makeSound() {
    echo " Squeak ";
```

```
  }
}

// Se crea una lista de animales
$cat = new Cat();
$dog = new Dog();
$mouse = new Mouse();
$animals = array($cat, $dog, $mouse);

// Por cada animal, se reproduce su sonido característico
foreach($animals as $animal) {
  $animal->makeSound();
}
?>
```

Como podemos apreciar, sabemos que todos los animales tienen un comportamiento común: hacer un sonido, pero no sabemos la forma en la que cada animal realiza ese sonido. Es ahí donde declarar e implementar una interfaz nos ayudará a estructurar el código de una forma óptima, sin la necesidad de declarar la misma función en cada clase. Ventajas del polimorfismo en su máximo esplendor.

TRAITS

Una clase hija solo puede heredar de una clase padre. Los traits permiten que una clase herede múltiples comportamientos (métodos). Pueden ser usados en varias clases, pueden tener métodos abstractos y "normales" y pueden tener cualquier modificador de acceso (*public, private* o *protected*). Son declarados con la palabra reservada `trait` y se usan con la palabra reservada `use` dentro de la clase.

```
<?php
trait message1 {
  public function msg1() {
    echo "OOP is fun! ";
  }
}

trait message2 {
  abstract public function msg2() { }
}

class Welcome {
  use message1;
}

class Welcome2 {
  use message2;
  public function msg2() {
```

```
    echo "OOP reduces code duplication!";
  }
}

$obj = new Welcome();
$obj->msg1();
echo "<br>";

$obj2 = new Welcome2();
$obj2->msg1();
$obj2->msg2();
?>
```

STATIC METHODS

Los métodos estáticos pueden ser llamados sin la necesidad de instanciar una clase. Son declarados mediante la palabra reservada `static`. Se acceden a ellos mediante el operador `::`

```
<?php
class Greeting {
  public static function welcome() {
    echo "Hello World!";
  }
}

// Invocamos al método estático
Greeting::welcome();
?>
```

Para invocar un método estático de una clase dentro de la misma usamos la palabra reservada `self` junto al operador `::`

```
<?php
class Greeting {
  public static function welcome() {
    echo "Hello World!";
  }

  public function __construct() {
     self::welcome();
  }
}
?>
```

Para invocar a un método estático de la clase padre, podemos usar la palabra reservada `parent` dentro de la clase hija. Alternativamente, podemos usar el nombre de la clase junto con el nombre del método estático y el operador `::`

```
<?php
class Domain {
  protected static function getWebsiteName() {
    return "W3Schools.com";
  }
}

class DomainW3 extends Domain {
  public $websiteName;

  public function __construct() {
    // Alternativa al uso de parent: Domain::getWebsiteName()
    $this->websiteName = parent::getWebsiteName();
  }
}
?>
```

STATIC PROPERTIES

Análogamente a los métodos estáticos, las propiedades estáticas pueden ser invocadas sin la necesidad de crear la instancia de una clase. Son exactamente iguales que los métodos estáticos.

NAMESPACES

Son contenedores abstractos en los que un grupo de uno o más identificadores únicos pueden existir. Son especialmente útiles para evitar ambigüedad entre clases con el mismo nombre pertenecientes al mismo namespace.

Por ejemplo, si usamos una librería externa que contiene la clase Pdf() y nosotros creamos también una clase llamada Pdf(), tendríamos que definir un `namespace` para nuestra clase Pdf() con el fin de distinguirla de la clase Pdf() de la librería externa.

En Java, por ejemplo, no se usan los **namespaces,** puesto que las clases y las interfaces se organizan en paquetes, los cuales son el contenedor que estaríamos creando con los **namespaces**.

Los **namespaces** (declarados con la palabra reservada `namespace`) tienen que estar declarados al **principio** del fichero PHP.

```
<?php
namespace Html;
// Las constantes, clases y funciones declaradas en este fichero
pertenecerán al namespace Html
```

```
class Table {
  public $title = "";
  public $numRows = 0;


  public function message() {
    echo "<p>Table '{$this->title}' has {$this->numRows}
rows.</p>";
  }
}
$table = new Table();
$table->title = "My table";
$table->numRows = 5;
?>
```

Para mayor anidamiento, se pueden crear **namespaces** anidados: `namespace Code\Html;`

Las clases que pertenecen a un `namespace` determinado pueden ser instanciadas de forma habitual. Es decir, sin la necesidad de especificar su `namespace`. No obstante, para acceder a una clase declarado en un `namespace` desde fuera de ese mismo `namespace`, necesitaremos especificar el `namespace`.

```
$row = new Html\Row(); // HTML es el namespace y Row la clase
perteneciente a ese namespace
```

Adicionalmente, los **namespaces** y las **clases** pertenecientes a un namespace concreto pueden poseer un alias para facilitar su escritura. Se concretan mediante la palabra reservada `use`.

```
use HTML as H;
$table = new H\Table();

use HTML\Table as T;
$table = new T();
```

ITERABLES

Un iterable es cualquier pseudo-tipo que puede ser recorrido mediante el bucle `foreach()`. Puede ser usado como tipo de dato de un argumento de una función o como el tipo de dato del valor que devuelve una función.

Pasando como parámetro una variable de tipo iterable:

```
<?php
function printIterable(iterable $myIterable) {
  foreach($myIterable as $item) {
    echo $item;
```

```
  }
}

$arr = ["a", "b", "c"];
printIterable($arr);
?>
```

Una función devolviendo un iterable:

```
<?php
function getIterable():iterable {
  return ["a", "b", "c"];
}

$myIterable = getIterable();
foreach($myIterable as $item) {
  echo $item;
}
?>
```

Son tratados como iterables todos los arrays y los objetos que implementan la interfaz `Iterator`. Un iterable, en esencia, contiene una lista de items y métodos para recorrerlos. Siempre hay un puntero que apunta hacia un item de la lista, identificado por una key (en el caso de los arrays, puede ser un índice o una clave asociativa).

Todos ellos contienen los siguientes métodos:

- `current()` - Devuelve el elemento al que está apuntando el puntero.
- `key()` - Devuelve el identificador del elemento al que está apuntando el puntero.
- `next()` - Mueve el puntero hacia el siguiente elemento de la lista.
- `rewind()` - Mueve el puntero hacia el primer elemento de la lista.
- `valid()` - Si el puntero no está apuntando hacia ningún elemento (por ejemplo, si el método `next()` fuese invocado al final de la lista), retorna *false*.

Ejemplo de una clase implementando la interfaz `Iterator`.

```
<?php
class MyIterator implements Iterator {
  private $items = [];
  private $pointer = 0;

  public function __construct($items) {
    // array_values() comprueba que las keys del array son números
     $this->items = array_values($items);
  }

  public function current() {
```

```
    return $this->items[$this->pointer];
  }

  public function key() {
    return $this->pointer;
  }

  public function next() {
     $this->pointer++;
  }

  public function rewind() {
     $this->pointer = 0;
  }

  public function valid() {
    return $this->pointer < count($this->items);
  }
}

// Pasamos el iterable como parámetro de la función
function printIterable(iterable $myIterable) {
  foreach($myIterable as $item) {
    echo $item;
  }
}

// Instanciamos la clase y la inicializamos pasando al constructor
un array
$iterator = new MyIterator(["a", "b", "c"]);
printIterable($iterator);
?>
```

3. Resuelve estos diez ejercicios sobre clases usando POO en PHP.

3.1. Crea una clase llamada **Punto** con dos propiedades/atributos denominados **x** e **y**, con constructor y con cuatro métodos (*getter* y *setter*), uno para obtener **x**, otro para obtener **y**, otro para modificar **x** y otro método para modificar **y**. Crea 3 instancias/objetos de la clase **Punto** y ejecuta en ellos los cuatro métodos creados.

```php
<?php
class Punto {
  private $x;
  private $y;

  public function __construct($x, $y) {
      $this->x = $x;
      $this->y = $y;
  }

  public function getX() {
    return $this->x;
  }

  public function getY() {
    return $this->y;
  }

  public function setX($x) {
    $this->x = $x;
  }

  public function setY($y) {
      $this->y = $y;
  }
}

$punto1 = new Punto(0, 0);
$punto1->setX(1);
echo $punto1->getX();
$punto1->setY(1);
echo $punto1->getY();



$punto2 = new Punto(1, 1);
```

```php
$punto2->setX(2);
echo $punto2->getX();
$punto2->setY(2);
echo $punto2->getY();

$punto3 = new Punto(2, 2);
$punto3->setX(3);
echo $punto3->getX();
$punto3->setY(3);
echo $punto3->getY();
?>
```

3.2. Crea una clase llamada **Linea** con cuatro propiedades/atributos denominados **x1**, **x2**, **y1** e **y2**, con constructor y con un método que obtenga el punto medio del segmento usando dichas propiedades/atributos. Crea 3 instancias/objetos de la clase **Linea** y ejecuta en ellos el método creado.

```php
<?php
class Linea {
  private $x1;
  private $x2;
  private $y1;
  private $y2;

  public function __construct($x1, $x2, $y1, $y2) {
      $this->x1 = $x1;
      $this->x2 = $x2;
      $this->y1 = $y1;
      $this->y2 = $y2;
  }

  public function getPuntoMedio() {
    $x = ($this->x1 + $this->x2) / 2;
    $y = ($this->y1 + $this->y2) / 2;
    return "El punto medio del segmento es ($x, $y)";
  }
}

$segmento1 = new Linea(18, 1, 1, 10);
echo $segmento1->getPuntoMedio() . "\n";

$segmento2 = new Linea(1, 2, 3, 4);
echo $segmento2->getPuntoMedio() . "\n";

$segmento3 = new Linea(4, 3, 2, 1);   {
```

```
    $x = ($this->x1 + $this->x2) / 2;
    $y = ($this->y1 + $this->y2) / 2;
    return "El punto medio del segmento es ($x, $y)";
  }
}

$segmento1 = new Linea(18, 1, 1, 10);
echo $segmento1->getPuntoMedio() . "\n";

$segmento2 = new Linea(1, 2, 3, 4);
echo $segmento2->getPuntoMedio() . "\n";

$segmento3 = new Linea(4, 3, 2, 1);
echo $segmento3->getPuntoMedio() . "\n";
?>
```

3.3. Crea una clase llamada **Rectangulo** con dos propiedades/atributos denominados **longitud** y **ancho**, con constructor y con un método que calcule el **area** del rectángulo usando dichas propiedades/atributos. Crea 3 instancias/objetos de la clase **Rectangulo** y ejecuta en ellos el método creado.

```
<?php
class Rectangulo {
  private $longitud;
  private $ancho;

  public function __construct($longitud, $ancho) {
      $this->longitud = $longitud;
      $this->ancho = $ancho;
  }

  public function getArea() {
    return $this->longitud * $this->ancho;
  }
}
$rectangulo1 = new Rectangulo(1, 10);
echo $rectangulo1->getArea() . "\n";

$rectangulo2 = new Rectangulo(4, 20);
echo $rectangulo2->getArea() . "\n";

$rectangulo3 = new Rectangulo(7, 37);
echo $rectangulo3->getArea() . "\n";
?>
```

3.4. Crea una clase llamada **Circulo** con una propiedad/atributo denominado **radio**, con constructor y con dos métodos que calculen el **area** del círculo y la **circunferencia** del círculo usando dichas propiedades/atributos. Crea 3 instancias/objetos de la clase **Circulo** y ejecuta en ellos los dos métodos creados.

```php
<?php
class Circulo {
  private $radio;

  public function __construct($radio) {
      $this->radio = $radio;
  }

  public function getArea() {
    return pi() * ($this->radio)**2;
  }

  public function getCircunferencia() {
    return 2 * pi() * $this->radio;
  }
}

$circulo1 = new Circulo(10);
echo $circulo1->getArea() . "\n";
echo $circulo1->getCircunferencia() . "\n";

$circulo2 = new Circulo(24);
echo $circulo2->getArea() . "\n";
echo $circulo2->getCircunferencia() . "\n";

$circulo3 = new Circulo(37);
echo $circulo3->getArea() . "\n";
echo $circulo3->getCircunferencia() . "\n";
?>
```

3.5. Crea una clase llamada **Estudiante** con dos propiedades/atributos denominados **nombre** y **notas** (array/lista), con constructor y con métodos que obtenga el nombre, modifique el nombre, obtenga las notas, modifique las notas y, por último, que obtenga la media de esas notas y las muestre. Crea 3 instancias/objetos de la clase **Estudiante** y ejecuta en ellos el método creado.

```php
<?php
class Estudiante {
  private $nombre;
  private $notas;

  public function __construct($nombre, array $notas) {
      $this->nombre = $nombre;
      $this->notas = $notas;
  }

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

  public function setNombre($nombre) {
    $this->nombre = $nombre;
  }

  public function getNotas() {
    return $this->notas;
  }

  public function setNotas(array $notas) {
    $this->notas = $notas;
  }

  public function getNotaMedia() {
    return array_sum($this->notas) / count($this->notas);
  }
}

$estudiante1 = new Estudiante("Antonio", [7.6, 8, 7.5, 8]);
echo $estudiante1->getNotaMedia() . "\n";

$estudiante2 = new Estudiante("José", [9.2, 9.5, 10, 10]);
echo $estudiante2->getNotaMedia() . "\n";

$estudiante3 = new Estudiante("Francisco", [3.4, 4.1, 5, 3.7]);
echo $estudiante3->getNotaMedia() . "\n"; ?>
```

3.6. Crea una función que reciba dos parámetros de entrada de tipo clase **Punto** (realizado en ejercicio 01) y que devuelva la distancia euclídea entre esos dos puntos. Ejecuta 3 llamadas de ejemplo de la función creada.

Nota: Se define la distancia euclidiana entre el punto P y Q con la siguiente fórmula: d(P,Q) = √((XQ – XP) ^ 2 + (YQ – YP) ^ 2)

```
<?php
function getDistanciaEuclidea(Punto $punto1, Punto $punto2) {
  return sqrt(($punto1->getX() - $punto2->getX())**2 +
($punto1->getY() - $punto2->getY())**2);
}

echo getDistanciaEuclidea(new Punto(3, 2), new Punto(4, 1));
echo getDistanciaEuclidea(new Punto(5, 3), new Punto(8, 4));
echo getDistanciaEuclidea(new Punto(7, 4), new Punto(12, 7));
?>
```

3.7. Crea una clase llamada **Linea2D** con dos propiedades/atributos denominados **p1** y **p2** de tipo clase **Punto** (realizado en ejercicio 01) y con dos métodos, uno que obtenga el punto medio del segmento y otro que obtenga la distancia euclídea, ambos usando dichas propiedades/atributos. Crea 3 instancias/objetos de la clase **Linea2D** y ejecuta en ellos los dos métodos creados.

```
<?php
class Linea2D {
  private $p1;
  private $p2;

  public function __construct(Punto $p2, Punto $p1) {
      $this->p1 = $p1;
      $this->p2 = $p2;
  }

  public function getPuntoMedio() {
    return "El punto medio de la línea es el (" .
(($this->p1->getX() + $this->p2->getX()) / 2) . ", " .
(($this->p1->getY() + $this->p2->getY()) / 2) . ")";
  }

  public function getDistanciaEuclidea() {
    return sqrt(($this->p1->getX() - $this->p2->getX())**2 +
($this->p1->getY() - $this->p2->getY())**2);
```

```
  }
}

$linea2d1 = new Linea2D(new Punto(2, 4), new Punto(8, 5));
echo $linea2d1->getPuntoMedio() . "\n";
echo $linea2d1->getDistanciaEuclidea();

$linea2d2 = new Linea2D(new Punto(5, 9), new Punto(6, 13));
echo $linea2d2->getPuntoMedio() . "\n";
echo $linea2d2->getDistanciaEuclidea();

$linea2d3 = new Linea2D(new Punto(11, 2), new Punto(9, 17));
echo $linea2d3->getPuntoMedio() . "\n";
echo $linea2d3->getDistanciaEuclidea();
?>
```

3.8. Crea una clase llamada **Forma** con una propiedad/atributo denominada **centro** de tipo clase **Punto** y un método que se llame **area** y que devuelva un número, por ejemplo 0. A continuación, crea dos clases llamadas **Rectangulo** y **Circulo** (realizados en ejercicios 03 y 04) que hereden de la clase **Forma** ya creada. Crea 3 instancias/objetos de las clases **Rectangulo**, **Circulo**, de la clase que hereda **Forma** y ejecuta sus métodos.

```
<?php
abstract class Forma {
  private $centro;

  abstract protected function getArea() : int;
}

class Rectangulo extends Forma {
  private $longitud;
  private $ancho;

  public function __construct($longitud, $ancho) {
      $this->longitud = $longitud;
      $this->ancho = $ancho;
  }

  public function getArea() : int {
    return $this->longitud * this-> ancho;
  }
}
```

```php
class Circulo extends Forma {
  private $radio;

  public function __construct($radio) {
      $this->radio = $radio;
  }

  public function getArea() : int {
    return pi() * ($this->radio)**2;
  }

  public function getCircunferencia() {
    return 2 * pi() * $this->radio;
  }
}

$rectangulo1 = new Rectangulo(34, 12);
echo $rectangulo1->getArea() . "\n";

$rectangulo2 = new Rectangulo(12, 52);
echo $rectangulo2->getArea() . "\n";

$rectangulo3 = new Rectangulo(3, 4);
echo $rectangulo3->getArea() . "\n";

$circulo1 = new Circulo(10);
echo $circulo1->getArea() . "\n";

$circulo2 = new Circulo(23);
echo $circulo2->getArea() . "\n";

$circulo3 = new Circulo(341);
echo $circulo3->getArea() . "\n";
?>
```

3.9. Crea una función que reciba un parámetro de entrada de tipo array/lista, con identificador **grupos**, de tamaño 3 cuyos elementos sean de tipo array/lista de clase **Estudiante** (realizado en ejercicio 05). La función tiene que devolver el índice del array/lista **grupos** cuyo promedio de notas del grupo de estudiantes sea el más alto. Ejecuta 1 llamada de ejemplo de la función creada.

Por ejemplo, hay tres grupos de Bachillerato con 25 alumnos cada uno. Cada grupo será un array/lista de 25 estudiantes (25 objetos/instancias de la clase **Estudiante**) que se añadirá al array/lista **grupos** inicialmente vacío. Cada estudiante tiene su

media final, pero lo que queremos es la media de todo ese grupo de estudiantes y compararlos con los otros grupos. Lo que buscamos finalmente es conocer qué grupo de bachillerato tiene los alumnos con mejor promedio de nota. Si se va a usar este ejemplo en el ejercicio, no es necesario tantos alumnos (5 por grupo sería más que suficiente).

```php
<?php
function getMejorGrupo(array $grupos) : int {
  if (count($grupos) > 3) {
    return -1;
  }

  $mejorGrupo = [];
  foreach($grupos as $key => $grupo) {
    $notaMedia = 0;
    foreach($grupo as $estudiante) {
      $notaMedia += $estudiante->getNotaMedia();
    }

    $notaMedia = $notaMedia / count($grupo);
    if (!(!empty($mejorGrupo)) || $notaMedia > $mejorGrupo[1]){
      $mejorGrupo[0] = $key;
      $mejorGrupo[1] = $notaMedia;
    }
  }

  return $mejorGrupo[0];
}

$grupo1 = [
  new Estudiante("Jesús", [9.8, 10]),
  new Estudiante("David", [8, 9]),
  new Estudiante("Pablo", [8.5, 10]),
  new Estudiante("Juan", [7.8, 8.1]),
  new Estudiante("Clemente", [8.1, 8.5])
];

$grupo2 = [
  new Estudiante("Sofía", [10, 10]),
  new Estudiante("Beatriz", [9.8, 9.9]),
  new Estudiante("Raquel", [9.7, 8.4]),
  new Estudiante("Manuela", [8.5, 8.7]),
  new Estudiante("Lola", [8.3, 8.9])
];
```

```php
$grupo3 = [
  new Estudiante("Manuel", [5.5, 6]),
  new Estudiante("Óscar", [5.8, 6.6]),
  new Estudiante("Leticia", [5.7, 7.4]),
  new Estudiante("Rocío", [6.5, 7.7]),
  new Estudiante("Santi", [4.3, 6.9])
];

$grupo = [
  $grupo1,
  $grupo2,
  $grupo3
];

echo getMejorGrupo($grupo) . "\n";
?>
```

3.10. Crea una clase C que herede de una clase B y que la clase B herede de A. La clase C heredará todos los métodos y atributos de B y B de A. Como mínimo, una función de A, de B y de C tienen que tener el mismo nombre pero que hagan cosas distintas. Crea 3 instancias/objetos de las clases A, B y C y ejecuta todos los métodos que hayas creado.

```php
<?php
class A {
  private $a;

  public function __construct($a) {
    $this->a = $a;
  }

  public function getA() {
    return $this->a;
  }

  public function setA($a) {
    $this->a = $a;
  }

  public function mostrarInformacion() {
    echo "A es $this->a\n";
  }
}
```

```
class B extends A {
  private $b;

  public function __construct($a, $b) {
    parent::__construct($a);
    $this->b = $b;
  }

  public function getB() {
    return $this->b;
  }

  public function setB($b) {
    $this->b = $b;
  }

  public function mostrarInformacion() {
    echo "A es {$this->getA()} y B es $this->b\n";
  }
}

class C extends B {
  private $c;

  public function __construct($a, $b, $c) {
    parent::__construct($a, $b);
    $this->c = $c;
  }

  public function getC() {
    return $this->c;
  }

  public function setC($c) {
    $this->c = $c;
  }

  public function mostrarInformacion() {
    echo "A es {$this->getA()}, B es {$this->getB()} y C es
$this->c\n";
  }
}


$a1 = new A("a1");
```

```php
$a1->mostrarInformacion();
$a2 = new A("a2");
$a2->mostrarInformacion();
$a3 = new A("a3");
$a3->mostrarInformacion();

$b1 = new B("a1", "b1");
$b1->mostrarInformacion();
$b2 = new B("a2", "b2");
$b2->mostrarInformacion();
$b3 = new B("a3", "b3");
$b3->mostrarInformacion();

$c1 = new C("a1", "b1", "c1");
$c1->mostrarInformacion();
$c2 = new C("a2", "b2", "c2");
$c2->mostrarInformacion();
$c3 = new C("a3", "b3", "c3");
$c3->mostrarInformacion();
?>
```

4. Define programación por capas y sus capas como BLL (lógica de negocio) o DAL y detalla la three-tier.

Consiste en una arquitectura cliente-servidor cuyo objetivo es dividir la presentación, el procesamiento de la aplicación y la gestión de los datos.

La arquitectura más extendida divide la aplicación **en tres capas** (**three-tier**):

- Capa de presentación. Es la capa con la que interactúan los clientes. Se comunica solo con la capa de negocio.
- Capa de negocio (**Business Logic Layer**). Aquí se reciben las peticiones del usuario y se envían las respuestas tras el proceso. Conecta la capa de presentación y la capa de datos.
- Capa de datos (**Data Access Layer**). Se encarga de acceder y consultar datos manejados por uno o más sistemas gestores de bases de datos (DBMS). Se comunica con la capa de negocio.

5. Lee exhaustivamente y explica con tus propias palabras lo que es un patrón de diseño sofware.

Se define como una solución aplicable a un problema recurrente dado en un contexto determinado.

Es importante destacar que, dependiendo del paradigma de programación al que pertenezca un lenguaje, podremos usar unos patrones u otros. Por ejemplo, hay patrones que pueden ser aplicados en lenguajes orientados a objetos pero no ser usados en la programación funcional.

Un ejemplo de patrón de diseño es el patrón Singleton, el cual evita que se creen más instancias de las necesarias de un objeto. Además, permite que se acceda a él desde cualquier punto de la aplicación (por el hecho de ser *estático*).

```
public final class Singleton {
  private static final Singleton INSTANCE = new Singleton();

  private Singleton() {}

    public static getInstance() {
        return INSTANCE;
    }
}
```

6. Lee exhaustivamente y explica con tus propias palabras lo que es el patrón de diseño software MVC.

Cabe destacar que MVC **NO** es un patrón de diseño: es un patrón **arquitectónico**. MVC sigue el principio de la programación en tres capas (**three-tier**, visto en el ejercicio 4): capa de presentación, de negocio y de datos.

Los principales objetivos de este patrón son reutilizar el código y separar la funcionalidad de la aplicación (encapsulación, ocultación y modularización del proyecto).

7. Mira este vídeo de principio a fin, resúmelo y explica el flujo de MVC, empezando por el usuario.

MVC es un patrón arquitectónico para una solución general y repetible a problemas comunes. No se limita a un lenguaje de programación, por lo que puede ser utilizado en frameworks de distintos lenguajes (Java, PHP, Python)...

Las siglas de MVC son **M**odelo (model), **V**ista (view) y **C**ontrolador (controller), las cuales son capas en las que se divide la aplicación/programa con el fin de separar sus responsabilidades.

En el **modelo**, se almacenan y administran datos.

En la **vista** se encuentra el código y elementos con los que interactúa el usuario y se representan los datos rescatados del modelo.

El **controlador** es el intermediario entre la *vista* y el *modelo*.

En el flujo de la imagen, se entiende que el controlador es el “cerebro” que conecta la vista (lo que ve el usuario) con el modelo (los datos a los que accede o envía el usuario). El controlador reacciona a cada acción del usuario y, dependiendo de su interpretación sobre ella, mandará al modelo que devuelva o almacene información. A ello, el modelo responde con los datos demandados por el controlador para que el propio controlador pueda enviársela a la vista y que la información proveída al usuario se actualice.

8. Lee bien este y este artículo y ejecuta, modifica y explica exhaustivamente esta plantilla MVC básica.

La estructura del ejemplo es la siguiente:

```
plantilla_mvc_simple/
├── bootstrap
│   └── bootstrap.min.css
├── plantillas/
│   ├── about.php
│   ├── inicio.php
│   └── modelo.php
├── Controlador.php
├── index.php
├── Modelo.php
└── Vista.php
```

Dentro del directorio *bootstrap* encontramos la hoja de estilos utilizada en todas las páginas.

Dentro del directorio *plantillas* están las plantillas/código HTML. En ellas, el único código PHP que encontramos está en el fichero *modelo.php*, en el que se realiza un *echo* de la propiedad *x* del modelo. Hay que tener en cuenta que todas estas plantillas son cargadas mediante la capa **Vista** (en el caso de este ejemplo, el archivo *Vista.php*).

También es importante enfatizar que en el *nav* de todas las páginas, cada atributo *href* de cada elemento *a* contiene como parámetro en su URL una *acción*. Lo que se hace, en esencia, es cargar el archivo *index.php* pero con una plantilla distinta, dependiendo de la acción.

La lógica de la acción mencionada anteriormente es gestionada por el **Controlador** (en el caso de este ejemplo, el archivo *Controlador.php*). Dependiendo de la acción pasada como parámetro en la URL, el controlador mandará a la vista la orden de cargar una plantilla determinada. A su vez, el controlador también le pasará un **Modelo** (en el caso de este ejemplo, el archivo *Modelo.php*) a la **Vista** en el caso de que sea necesario cargar información adicional.

El **Modelo**, en este caso, contiene una propiedad llamada *x* (con su respectivo getter y setter) que se usa para mostrar su valor en la plantilla de *modelo.php*.

9. Lee exhaustivamente y explica con tus propias palabras lo que es un *framework* web.

Para explicar lo que es un framework web, primero tendríamos que explicar lo que es un framework. Consisten en una herramienta que provee una serie de componentes o soluciones para agilizar y facilitar la etapa de desarrollo. En líneas generales, añade una capa de abstracción al lenguaje de programación.

Un framework web es un framework aplicado al desarrollo web. Hay muchos frameworks usados en web, como *Boostrap* para CSS, *Laravel* y *Symphony* para PHP, *Django* para Python, *Spring* para Java, *Node.js* para JavaScript en entorno servidor y Vue, React y Angular para Javascript en entorno cliente.

Añadir que hay distintas estructuras de frameworks, los cuales están basados mayoritariamente en el MVC, y frameworks de aplicación que abstraen el lenguaje de programación, los cuales serían los mencionados en el párrafo anterior.

10. Lee exhaustivamente y explica con tus propias palabras lo que es un Laravel.

Laravel es un framework web de PHP que abstrae las tareas comunes y monótonas que se repiten en multitud de páginas web dinámicas, como la autenticación, las sesiones o el cacheo. Sigue el patrón de diseño MVC y está basado en Symfony, otro framework de PHP.