SOC Servei d'Ocupació de Catalunya

nascor formación

MINISTERIO DE TRABAJO E INMIGRACIÓN

SERVICIO PÚBLICO DE EMPLEO ESTATAL

Certificado de profesionalidad IFCD0210

Generalitat de Catalunya
Departament de Treball

Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

Las Clases

Una **clase** define los datos y la lógica de un objeto.
La lógica se divide en funciones (**métodos**) y variables (**propiedades**).

Propiedades

```
class Coche {
        public $color;
}
```

Métodos

```
class Coche {
        public function getColor() {
            // Contenido de la función
        }
}
```

Las Clases

Para **crear un objeto** hay que **instanciar una clase**:

```
class Coche {
        // Contenido de la clase
}

$miCoche = new Coche(); // Objeto
```

Las Clases

La **clase** es como una **plantilla** que define características y funciones.
El **objeto** agrupa los datos de la clase y permite utilizarlos desde una unidad.

```
class Coche {
        public $color;
        public $potencia;
        public $marca;
}

$miCoche = new Coche();
$miCoche->color = 'rojo';
$miCoche->potencia = 120;
$miCoche->marca = 'audi';
```

Las Clases

De momento sólo hemos definido **propiedades** del objeto. Si queremos mostrar alguna característica ahora:

```
//...
echo 'Color del coche: ' . $miCoche->color; // Muestra Color del coche: rojo
```

Las Clases

Ahora vamos a definir un **método que devuelve una propiedad**:

```
class Coche {
    //...

    public function getColor()
    {
        return $this->color;
    }
}
//...
```

La variable ***$this*** se puede utilizar en cualquier método, y hace referencia al objeto que hemos instanciado, en este caso $miCoche

Las Clases

El método ***getColor()*** nos permite devolver el color del objeto instanciado. Podemos obtener el mismo resultado que antes para mostrar el color del coche pero ahora utilizando un **método** para mostrar la **propiedad** color:

```
//...
echo 'Color del coche: ' . $miCoche->getColor();
```

Las Clases

Las **propiedades** pueden tener **valores por defecto**:

```
class Coche { public $color = 'rojo'; //... }
```

Las Clases

De esta forma siempre que se instancie un objeto de la clase **Coche**, éste será de color rojo a no ser que se modifique después. Ahora creamos también el objeto *$otroCoche* y le ponemos otras características:

```
class Coche {
        public $color = 'rojo';
        public $potencia;
        public $marca;

        public function getColor() {
            return $this->color;
         }
}
```

```
$miCoche = new Coche();
$miCoche->color = 'verde';
$miCoche->potencia = 120;
$miCoche->marca = 'audi';

$otroCoche = new Coche();
$otroCoche->color = 'azul';
$otroCoche->potencia = 100;
$otroCoche->marca = 'bmw';
```

Las Clases

Creamos ahora dos "*getters*" para la potencia y la marca:

```
class Coche {
    public $color = 'rojo';
    public $potencia;
    public $marca;
    public function getColor() {
            return $this->color;
    }
    public function getPotencia() {
            return $this->potencia;
    }
    public function getMarca() {
            return $this->marca;
    }
}
```

Las Clases

Y creamos una función (fuera de la clase) que devuelve las caracteristicas totales de un objeto

```
function printCaracteristicas($cocheConcreto) {
    echo 'Color: '. $cocheConcreto->getColor();
    echo "<br>";
    echo 'Potencia: '. $cocheConcreto->getPotencia();
    echo "<br>";
    echo 'Marca: '. $cocheConcreto->getMarca();  }
$miCoche = new Coche();
$miCoche->color = 'verde';
$miCoche->potencia = 120;
$miCoche->marca = 'audi';

$otroCoche = new Coche();
$otroCoche->color = 'azul';
$otroCoche->potencia = 100;
$otroCoche->marca = 'bmw';
printCaracteristicas($miCoche);
printCaracteristicas($otroCoche);
```

La función *printCaracteristicas()* requiere un argumento, *$cocheConcreto*, que es una instancia de la clase Coche que mostrará las características del propio objeto.

Interacciones entre objetos

Supongamos ahora que queremos comparar y mostrar **qué coche es más rápido** dependiendo de su potencia. Añadimos un nuevo **método** a la clase **Coche**:

```
class Coche { //...
  public function elCocheElegidoEsMasRapido($cocheElegido) {
        return $cocheElegido->potencia > $this->potencia;
    }
 }
 //...
```

Interacciones entre objetos

Tenemos dos **objetos** *$miCoche* y *$otroCoche*. La condición que mostrará que coche es más rápido:

```
//...
if ($miCoche->elCocheElegidoEsMasRapido($otroCoche)) {
      echo 'El ' . $otroCoche->marca . ' es más rápido';
    }
else {
      echo 'El ' . $miCoche->marca . ' es más rápido';
}

$miCoche->elCocheElegidoEsMasRapido($otroCoche);

// En este caso el Audi ($miCoche) es más rápido que el BMW ($otroCoche)
```

Interacciones entre objetos

Al utilizar el operador de comparación (==), se comparan de una forma sencilla las variables de cada objeto, es decir: dos instancias de un objeto son iguales si tienen los mismos atributos y valores (los valores se comparan con ==), y son instancias de la misma clase.

Cuando se utiliza el operador identidad (===), las variables de un objeto son idénticas sí y sólo sí hacen referencia a la misma instancia de la misma clase.

https://www.php.net/manual/es/language.oop5.object-comparison.php

Public vs private

Hasta ahora hemos definido todas las propiedades y métodos como ***public***. Esto significa que pueden ser accedidos y alterados desde cualquier parte fuera de la clase.

En el ejemplo anterior hemos podido definir *$miCoche->potencia* sin ningún problema, y podríamos aplicarle cualquier valor (un número, un string...).

```
class Coche {
 //...
public function setPotencia($potencia)
   {
      $this->potencia = $potencia;
   }
 //...
 }
```

Public vs private

Cuando se define una propiedad como ***private***, se indica que ésta no puede verse o modificarse a no ser que sea desde la propia clase. Si utilizamos *$miCoche->potencia* para verla o definirla, nos dará error. Si queremos definirla, se utiliza un ***setter***, un **método public** para definir una **propiedad private**:

```
class Coche {
private $potencia;
public function setPotencia($potencia)
  {
     $this->potencia = $potencia;
  }
 //...
 }
```

Public vs private

```php
<?php
class Coche {
    public $potencia;
    public function setPotencia($potencia)
    {
        $this->potencia = $potencia;
    }
    public function getPotencia()
    {
        return $this->potencia;
    }
}
$miCoche = new Coche;
$miCoche->setPotencia(120);
echo $miCoche->potencia;
?>
```

```php
<?php
class Coche {
    private $potencia;
    public function setPotencia($potencia)
    {
        $this->potencia = $potencia;
    }
    public function getPotencia()
    {
        return $this->potencia;
    }
}
$miCoche = new Coche;
$miCoche->setPotencia(120);
echo $miCoche->getPotencia();
?>
```

Public vs private

```php
<?php
class Coche {
    private $potencia;
    public function setPotencia($potencia)
    {
        //lanzamos un error en caso que
$potencia          //no sea un número
        if(!is_numeric($potencia)){
            throw new \Exception('Potencia no
válida: '. $potencia);
        }
        $this->potencia = $potencia;
    }
    public function getPotencia()
    {
        return $this->potencia;
    }
} …..
```

```php
……..
$miCoche = new Coche;

$miCoche->setPotencia(120);
echo $miCoche->getPotencia();

$miCoche->setPotencia('Hola');
echo $miCoche->getPotencia();
?>
```

Actidad 11: objetos

Declarar una clase Persona con las siguientes propiedades: DNI, Nombre, Apellido1, Apellido2 y Ciudad.

Crear sus correspondientes setters y getters de estos datos.

Crear 2 objetos instanciados de persona y mostrar en pantalla sus datos.

Al menos, uno de las propiedades se ha de definir como private.

Constructores y Destructores

```
Class Persona {
...

function __construct($dni,$nom,$ape1,$ape2,$ciutat,$Data) {
        $this->dni=$dni;
        $this->nombre=$nom;
        $this->apellido1=$ape1;
        $this->apellido2=$ape2;
        $this->ciudad=$ciutat;
        $this->fechanac=$Data;
    }
…
}
$persona1 = new Persona('38098925P','Borja','Mulleras','Vinzia','Barcelona','1972-01-19');
```

Constructores y Destructores

```
public function __destruct() {
    echo 'Destruyendo: ', $this->nombre;
}
```

Cuando se acaba el PHP se ejecuta esta función!!!!

Herencia

```
class Coche {
    private $color = 'red';
    protected $potencia = 120;
    public $marca = 'audi';
}
```

```
class CocheDeLujo extends Coche {
    // La función displayPotencia devolverá 120
    public function displayPotencia()
    {
        return $this->potencia;
    }
    // La función displayMarca devolverá audi
    public function displayMarca()
    {
        return $this->marca;
    }
}
```

Herencia

```
class Coche {
    private $color = 'red';
    protected $potencia = 120;
    public $marca = 'audi';
}
```

```
class CocheDeLujo extends Coche {
…
// La función displayColor devolverá un error porque es private
    public function displayColor()
    {
        return $this->color;
    }
…
}
```

Herencia

```
class Coche {
    protected $color;
    public function setColor($color)
    {
        $this->color = $color;
    }
    public function getColor()
    {
        return $this->color;
    }
    public function printCaracteristicas()
    {
        echo 'Color: '. $this->getColor;
    }
}
```

```
class CocheDeLujo extends Coche {
  protected $extras;
  public function setExtras($extras) {
      $this->extras = $extras;
  }
  public function getExtras() {
       return $this->extras;
  }
  public function printCaracteristicas() {
       echo 'Color: '. $this->color;
       echo '<hr/>';
       echo 'Extras: ' . $this->extras;
   }
}
```

```
$miCoche = new CocheDeLujo();
$miCoche->setColor('negro');
$miCoche->setExtras('TV');
$miCoche->printCaracteristicas();
// Devuelve Color : negro Extras: TV
```

Herencia

Private:
- Desde la misma clase que declara

Protected:
- Desde la misma clase que declara
- Desde las clases que heredan esta clase

Public:
- Desde la misma clase que declara
- Desde las clases que heredan esta clase
- Desde cualquier elemento fuera de la clase

Herencia

Es posible **impedir que un método pueda sobreescribirse** mediante la palabra **final**:

```
class Coche {
    final public function getColor()
    {
        echo "Rojo";
    }
}
class CocheDeLujo extends Coche {
    public function getColor()
    {
        echo "Azul";
    }
} // Dará un error fatal, getColor() no puede
sobreescribirse
```

Herencia

También se puede **impedir que la clase pueda heredarse** mediante la misma palabra:

```
final class Coche {
    public function getColor()
    {
        echo "Rojo";
    }
}
class CocheDeLujo extends Coche {
    // Error fatal, clase no heredable
}
```

¿Podemos guardar los objetos?

```
$data = file_get_contents("usuarios");
$personas = unserialize($data, ['allowed_classes' => true]);

…….....


$file = fopen("usuarios", "w"); // Abrir
fwrite($file,serialize($personas));
fclose($file); // Cerrar
```

Ver ejemplo: https://aspasia11.md360.es/ejemplos/anadirOB6.html

Actividad 12

Convertir la aplicación de personas usando objetos.

Modelo Vista Controlador

El paradigma modelo vista controlador (MVC) es un patrón de diseño que separa el código en tres capas. Utilizar un patrón de diseño a menudo es muy recomendable, ya que es una forma estandarizar nuestro código, optimizarlo y hacer que sea más legible.

Modelo Vista Controlador

Modelo
Gestiona todo lo relacionado con la información y la iteración con los datos de nuestra aplicación. Todas las peticiones de acceso a los datos pasará por esta capa. => Las clases con sus propiedades y métodos que se comunican con los datos.

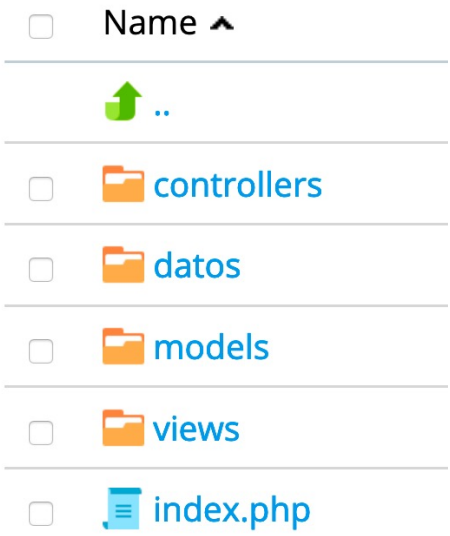
Vista
Es la capa que nos mostrará la información formateada. También desde dónde el usuario puede solicitar más información. => La parte visual: html, css. etc…

Controlador
Une la vista y el modelo. El usuario solicitará información por medio de la vista y esta hará la petición al controlador. Posteriormente, este, realizará la petición al modelo. => Intermediario entre los 2 anteriores

Funcionamiento de un MVC

1. El usuario realiza una petición.
2. El controlador captura la petición.
3. Hace la llamada al modelo correspondiente.
4. El modelo será el encargado de interactuar con los datos.
5. El controlador recibe la información y la envía a la vista.
6. La vista muestra la información.

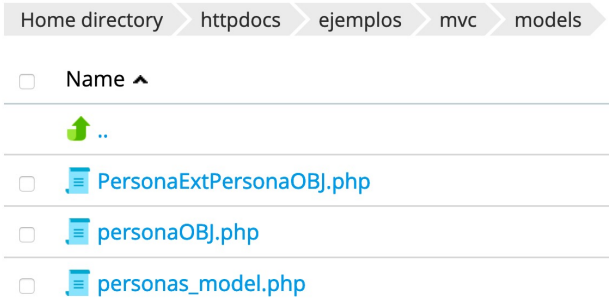
Funcionamiento de un MVC



Index.php

```
<?php
require_once("controllers/personas_controller.php");
?>
```

Funcionamiento de un MVC




```php
<?php //personas_model.php
require_once 'PersonaExtPersonaOBJ.php';
class personas_model{
    private $db;
    private $personas;

    public function __construct(){
        $this->personas=array();
    }
    public function get_personas(){
        $data = file_get_contents("datos/usuarios.dat");
        $this->personas = unserialize($data,
['allowed_classes' => true]);
        return $this->personas;
    }
}
?>
```

Funcionamiento de un MVC

El controlador debe tener siempre esta estructura llamada al modelo y debajo a la vista, si hubiera mas modelos y vistas se sigue haciendo así con todos.

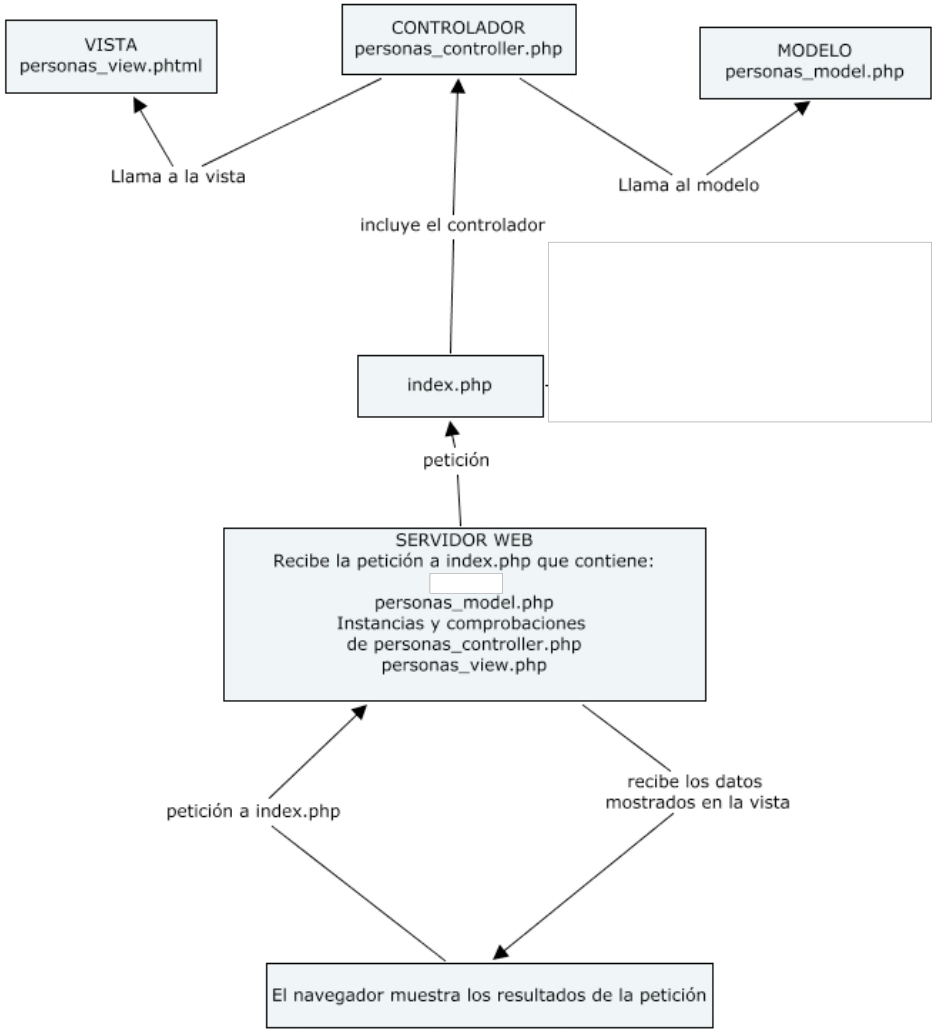
```
<?php // contoller/personas_controller.php
//Llamada al modelo
require_once("models/personas_model.php");
$per=new personas_model();
$datos=$per->get_personas();

//Llamada a la vista
require_once("views/personas_view.php");
?>
```

Funcionamiento de un MVC

view/personas_view.php

```
<!DOCTYPE html>
<html lang="es">
    <head>
        <meta charset="UTF-8" />
        <title>Personas</title>
    </head>
    <body>
        <?php
            foreach ($datos as $dato) {
                echo $dato->VerPersona()."<br/>";
            }
        ?>
    </body>
</html>
```

Funcionamiento de un MVC



https://aspasia11.md360.es/ejemplos/mvc/index.php

SOC Servei d'Ocupació de Catalunya

nascor formación

MINISTERIO DE TRABAJO E INMIGRACIÓN

SERVICIO PÚBLICO DE EMPLEO ESTATAL

Certificado de profesionalidad IFCD0210

Generalitat de Catalunya
Departament de Treball

Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur