TEMA
4

2ºDAW

Orientación a objetos

1 Introducción a la POO

Mientras que la programación estructurada se basa en la utilización de estructuras de control y usa como almacenamiento de información las variables y los arrays; la **programación orientada a objetos** se basa, como su nombre indica, en la utilización de **objetos, clases o entidades**.

La programación estructurada y la POO no son excluyentes, un programa basado en objetos seguirá teniendo variables, bucles y sentencias condicionales, no obstante éste **último estará seguramente mejor organizado y será más legible y escalable. Los componentes básicos de la POO son**

Componente básico	Definición	Ejemplo
Clase	Concepto abstracto que forman la estructura de los datos y las acciones.Sirven para construir objetos cada uno con sus atributos a través de los métodos	Planos de un coche
Instancia clase u objeto	Objeto palpable, que se deriva de la concreción de una clase	Coche
Atributos	Conjunto de propiedades que se aplican al objeto	Matrícula, marca, modelo, color y número de plazas.
Métodos	Funciones o acciones que permite efectuar el objeto	ArrancarMotor, CambiarColor
Mensajes	Una forma de comunicar unos objetos con otros	Tocar el claxon

2 Clases en PHP

Las clases en PHP comienzan con una letra mayúscula. 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).

El **encapsulamiento** consiste en definir todas las propiedades y el comportamiento de una clase dentro de esa clase, nunca mezclar parte de una clase con otra.
La **ocultación** es una técnica que incorporan algunos lenguajes (entre ellos Java) que permite esconder los elementos que definen una clase, de tal forma que desde otra clase distinta no se pueden "ver las tripas" de la primera. La ocultación facilita, como veremos más adelante, el encapsulamiento.

A continuación tenemos un ejemplo muy sencillo. Se trata de la implementación de la clase **Persona**. Esta clase tendrá dos atributos: nombre, profesión y edad.

Se define también el constructor. Este método es muy importante ya que se llamará siempre que se creen nuevos objetos de la clase y servirá generalmente para inicializar los valores de los atributos. En PHP, el constructor de una clase se define con el nombre _construct()

Hay otro método especial muy útil, se trata **de _toString.** Si se define el método _toString en una clase, cuando se realice un echo sobre un elemento de esa clase, se ejecutará dicho _toString.

Se crea además un método que, aplicado a una instancia de la clase Persona, muestra un mensaje por pantalla.

```
<?php
class  Persona {
private $nombre;
private $profesion;
private $edad;

// Constructor
public function   __construct($nom,  $pro,  $edad)  {
$this->nombre =  $nom;
$this->profesion =  $pro;
$this->edad =  $edad;
}

public function getEdad()  {
return $this->edad;
}

public function setEdad($edad)  {
$this->edad=$edad;
}


public function presentarse()  {
return "Hola,  me llamo " . $this->nombre . " y  soy   " . $this->profesion . ".<br>";
}


public function    __toString()  {
return "<hr><b>$this->nombre</b><br> Profesión: $this->profesion<br> Edad: $this-
>edad<hr>";}}
```

Un elemento public (público) es visible desde cualquier clase, un elemento **protected (protegido)** es visible desde la clase actual y desde todas sus subclases (en el siguiente apartado se estudia el concepto de subclase) y, finalmente, un elemento **private (privado**) únicamente es visible dentro de la clase actual. Por regla general, se suelen definir los atributos como privados y los métodos como públicos.

```
<!DOCTYPE  html>
<html>
<head>
<meta  charset="UTF-8">
<title></title>
</head>
<body>
<?php
include_once 'Persona.php';

$unTipo =  new  Persona("Pepe Pérez",  "albañil",  30);
$unNota =  new  Persona("Rigoberto Peláez",  "programador", 25 );
echo   $unTipo->presentarse();
echo   $unNota->presentarse();
echo   $unTipo;
echo   $unNota;

?>
</body>
</html>
}
}
```

A continuación se muestra la implementación de la clase Gato.

```
<?php

class Gato  {

// atributos

private $color;
private $raza;
private $edad;
private $peso;
private $sexo;

// métodos

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

public function getSexo()  {
return $this->sexo;
}

public function maulla()  {
echo   "Miauuuu<br>";
}
```

```
public function ronronea() {
echo "mrrrrrr<br>";
}

public function come($comida) {
if ($comida == "pescado") {
echo "Hmmmm, gracias<br>";
} else {
echo "Lo siento, yo solo como pescado<br>";
}
}

public function peleaCon($contrincante) {
if (($this->getSexo()) == "hembra") {
echo "no me gusta pelear<br>";
} else if (($contrincante->getSexo()) == "hembra") {
echo "Prefiero no pelear<br/>";
} else {

echo "ven aquí que te vas a enterar<br>";
}}}
```

El programa que prueba la clase Gato es el siguiente.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title></title>
</head>
<body>
<?php
include_once 'Gato.php';


$garfield = new Gato("macho");


echo "hola gatito<br>";
$garfield->maulla();


echo "toma tarta<br>";
$garfield->come("tarta selva negra");
echo "toma pescado, a ver si esto te gusta<br>";
$garfield->come("pescado");
```

```
$tom = new Gato("macho");

echo "Tom, toma sopita de verduras<br>";
$tom->come("sopa de verduras");

$lisa = new Gato("hembra");

echo "gatitos, a ver cómo maulláis<br>";
$garfield->maulla();
$tom->maulla();
$lisa->maulla();

$garfield->peleaCon($lisa);
$lisa->peleaCon($tom);
$tom->peleaCon($garfield);
?>
</body>
</html>
```

3 Herencia

La herencia es una de las características más importantes de la POO. Si definimos una serie de atributos y métodos para una clase, al crear una subclase, todos estos atributos y métodos siguen siendo válidos.

A continuación se muestra la implementación de la clase Animal. Uno de los métodos de esta clase es duerme. Luego crearemos las clases Gato y Ave como subclases de Animal. De forma automática, se podrá utilizar el método duerme con las instancias de las clases Gato y Ave.

```
<?php
// Declaramos como abstracta clase animal
abstract class Animal {

private $sexo;

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

public function __toString() {
return "Sexo: $this->sexo";
}

public function getSexo() {
return $this->sexo;
}
```

```
public function duerme()  {
return "Zzzzzzz";
}

public function aseate()  {
return "Me  gusta asearme, soy   un  animal.<br>";
}}
```

Clase abstracta (abstract): Una clase abstracta es aquella que no va a tener instancias de forma directa, aunque sí habrá instancias de las subclases (siempre que esas subclases no sean también abstractas). Por ejemplo, si se define la clase Animal como abstracta, no se podrán crear objetos de la clase Animal, es decir, no se podrá hacer **$mascota = new Animal(),** pero sí se podrán crear instancias de la clase Gato, Ave o Pinguino que son subclases de Animal.

La clase Ave es subclase de Animal y la clase Pinguino, a su vez, sería subclase de Ave y por tanto hereda todos sus atributos y métodos.

Para crear en PHP una subclase de otra clase existente se utiliza la palabra reservada **extends**. A continuación se muestra el código de las clases Gato, Ave y Pinguino, así como el programa que prueba estas clases creando instancias y aplicándoles métodos.

```
<?php

include_once 'Animal.php';

// Gato es subclase de Animal utilizando extends
class Gato  extends  Animal {

private $raza;

public function  __construct($s="macho",  $r="siames") {
parent:: __construct($s); //llamamos a la clase padre con parent
$this->raza =  $r;
}

public function    __toString()  {
return parent:: __toString() . "<br>Raza: $this->raza";
}


public function maulla()  {
return "Miauuuu<br>";
}


public function ronronea()  {
return "mrrrrrr<br>";
}
```

```
public function come($comida) {
if ($comida == "pescado") {

return "Hmmmm, gracias<br>";
} else {
return "Lo siento, yo solo como pescado<br>";
}
}

public function peleaCon($contrincante) {
if ((parent::getSexo()) == "hembra") {
echo "no me gusta pelear<br>";
} else if (($contrincante->getSexo()) == "hembra") {
echo "no me gusta pelear<br>";
} else {
echo "ven aquí que te vas a enterar<br>";
}
}
}
```

Mediante **parent** se puede hacer una llamada al método de la clase padre. Por ejemplo `parent:: __construct($s);` dentro de la definición de la clase Gato invoca al constructor de la clase padre, es decir, la clase Animal.

A continuación tenemos la definición de la clase Ave que es una subclase de Animal.

```
<?php


include_once 'Animal.php';

class Ave extends Animal {

public function __construct($s) {
parent:: __construct($s);
}

public function aseate() {
return "Me estoy limpiando las plumas<br>" . parent::aseate();
}

public function vuela() {
return "Estoy volando<br>";
}
}
```

El siguiente código corresponde a la definición de la clase Pinguino que es subclase de Ave y, por lo tanto, también es subclase de Animal. Observa cómo se sobreescribe el método vuela(). Cuando se define un método en una subclase con el mismo nombre que tiene en la superclase, tiene preferencia en la ejecución el método de la subclase.

```
<?php

include_once 'Ave.php';

class  Pinguino extends Ave  {

public function    __construct($s)  {
parent::   construct($s);
}

public function aseate()  {
return parent::aseate()  . "A  los pingüinos nos   gusta asearnos<br>";
}

public function vuela() {
return "No  puedo volar<br>";
}
}
```

Por último mostramos el programa que prueba las clases definidas anteriormente.

```
<!DOCTYPE  html>
<html>
<head>
<meta  charset="UTF-8">
<title></title>
</head>
<body>
<?php
include_once 'Animal.php';  // no  es necesario incluirla
include_once 'Ave.php';  // no  es necesario incluirla
include_once 'Pinguino.php';
include_once 'Gato.php';

$garfield =  new  Gato("macho",  "romano");
$tom  =  new  Gato("macho");
$lisa =  new  Gato("hembra");
$silvestre =  new  Gato();

echo   "$garfield<hr>";
echo   "$tom<hr>";
echo   "$lisa<hr>";
echo   "$silvestre<hr>";
```

```
$miLoro = new Ave(“aguila”);
echo $miLoro->aseate();
echo $miLoro->vuela();

$pingu = new Pinguino("hembra");
echo $pingu->aseate();
echo $pingu->vuela();
?>
</body>
</html>
```

Ya que los ficheros Animal.php y Ave.php se cargan desde Gato.php y Pinguino.php respectivamente. No obstante, se pueden dejar por claridad, para indicar todas las clases que intervienen en el programa.

5 Atributos y métodos de clase (static)

Hasta el momento hemos definido atributos de instancia como $raza o $sexo y métodos de instancia como maulla(), come() o vuela(). De tal modo que si en el programa se crean 20 gatos, cada uno de ellos tiene su propia raza y puede haber potencialmente 20 razas diferentes. También podría aplicar el método maulla() a todos y cada uno de esos 20 gatos.

No obstante, en determinadas ocasiones, nos puede interesar tener **atributos de clase (variables de clase) y métodos de clase**. Cuando se define una variable de clase solo existe una copia del atributo para toda la clase y no una para cada objeto. Esto es útil cuando se quiere llevar la cuenta global de algún parámetro. Los métodos de clase se aplican a la clase y no a instancias concretas.

A continuación se muestra un ejemplo que contiene la variable de clase $kilometrajeTotal. Si bien cada coche tiene un atributo $kilometraje donde se van acumulando los kilómetros que va recorriendo, en la variable de clase **$kilometrajeTotal** se lleva la cuenta de los kilómetros que han recorrido todos los coches que se han creado.

También se crea un método de clase llamado **getKilometrajeTotal()** que simplemente es un getter para la variable de clase $kilometrajeTotal.

```
<?php


class Coche {

// atributo de clase
private static $kilometrajeTotal = 0;


// método de clase
public static function getKilometrajeTotal() {
return Coche::$kilometrajeTotal;
```

```
}


private $marca;
private $modelo;
private $kilometraje;

public function __construct($ma, $mo) {
$this->marca = $ma;
$this->modelo = $mo;
$this->kilometraje = 0;
}

public function getKilometraje() {
return $this->kilometraje;
}

public function recorre($km) {
$this->kilometraje += $km;
Coche::$kilometrajeTotal += $km;
}}
```

El atributo `$kilometrajeTotal` almacena el número total de kilómetros que recorren todos los objetos de la clase Coche, es un único valor, por eso se declara como `static`. Por el contrario, el atributo `$kilometraje` almacena los kilómetros recorridos por un objeto concreto y tendrá un valor distinto para cada uno de ellos. Si en el programa principal se crean 20 objetos de la clase Coche, cada uno tendrá su propio `$kilometraje`.

A continuación se muestra el programa que prueba la clase Coche.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title></title>
</head>
<body>
<?php
include_once 'Coche.php';


$cocheDeLuis = new Coche("Saab", "93");
$cocheDeJuanK = new Coche("Toyota", "Avensis");


$cocheDeLuis->recorre(30);
$cocheDeLuis->recorre(40);
$cocheDeLuis->recorre(220);
$cocheDeJuanK->recorre(60);
$cocheDeJuanK->recorre(150);
```

```
$cocheDeJuanK->recorre(90);
echo  "El  coche de  Luis ha  recorrido " . $cocheDeLuis->getKilometraje()  . "Km<br>";
echo  "El  coche de  Juan Carlos  ha  recorrido " . $cocheDeJuanK->getKilometraje()  . "Km\
<br>";
echo  "El kilometraje total  ha  sido de  " . Coche::getKilometrajeTotal()  . "Km";
?>
</body>
</html>
```

El método getKilometrajeTotal() se aplica a la clase Coche por tratarse de un método de clase (método static). Este método no se podría aplicar a una instancia, de la misma manera que un método que no sea static no se puede aplicar a la clase sino a los objetos.

6 Serialización de objetos

Como hemos visto en capítulos anteriores, cuando se recarga una página se pierden todos los datos que no se hayan guardado en una sesión. Lo mismo sucede con los objetos, si se crean instancias de una clase, se perderán en el momento en que se recargue la página o cuando el flujo de la aplicación nos lleve a una página diferente a la actual.

Los objetos creados a partir de una clase se pueden guardar en variables de sesión pero es necesario tratarlos convenientemente; hay que serializarlos antes de asignarlos. En el proceso de serialización, una estructura - en este caso un objeto - queda transformada en una cadena de caracteres. Cuando se quiera recuperar el objeto a partir de la sesión, habrá que hacer el proceso inverso, es decir des-serializar la variable de sesión para obtener la instancia.

Aunque parece algo complicado, este proceso se verá claramente con un ejemplo. En primer lugar definimos la clase **MonstruoDeLasGalletas.**

```
<?php


class MonstruoDeLasGalletas {

private $galletas;  // galletas  comidas

public function    __construct($s)  {
$this->galletas =  0;
}

public function getGalletas()  {
return $this->galletas;
}

public function come($g)  {
$this->galletas =  $this->galletas +  $g;
}
```

```
}
```

El programa principal es el siguiente.

```
<?php
session_start();


include_once 'MonstruoDeLasGalletas.php';


if (!isset($_SESSION['coco'])) {
$_SESSION['coco'] = serialize(new MonstruoDeLasGalletas());
}


$coco = unserialize($_SESSION['coco']);
?>

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title></title>
</head>
<body>
<?php



$numeroDeGalletas = $_POST['numeroDeGalletas'];
if (isset($numeroDeGalletas)) {
$coco->come($numeroDeGalletas);

}
?>


<h2>Soy Coco y he comido <?=$coco->getGalletas(); ?> galletas.</h2>
<form action="index.php" method="POST"> Nº de galletas:
<input type="number" name="numeroDeGalletas" min="1">
<input type="submit" value="Comer">
</form>


<?php
$_SESSION['coco'] = serialize($coco);
?>
```

```
</body>
</html>
```

En la primera carga de página se crea un objeto de la clase MonstruoDeLasGalletas, se serializa ese objeto y se guarda en una variable de sesión con nombre $_SESSION['coco'].

```
if (!isset($_SESSION['coco'])) {
  $_SESSION['coco'] = serialize(new  MonstruoDeLasGalletas());
}
```

Antes de operar con el objeto, debe extraerse de la sesión.

```
$coco = unserialize($_SESSION['coco']);
```

A partir de ahora, $coco es un objeto de la clase MonstruoDeLasGalletas al que le podemos aplicar cualquiera de los métodos definidos.

Con el fin de conservar el objeto en memoria, lo serializamos y lo guardamos en la sesión.

```
$_SESSION['coco'] = serialize($coco);
```

Ejercicios

- **Ejercicio 1.** Crea las clase Abstracta **Animal** con la variable de instancia sexo y la subclase **Gato** con la variable de instancia raza.

  Para la Clase Animal

  - Contructor establece el sexo (un valor por defecto por si no se introduce)
  - ToString mostrando el sexo
  - GetSexo y SetSexo
  - Algún método de instancia más

  Para la Clase Gato

  - Contructor establece la raza (un valor por defecto por si no se introduce)
  - ToString mostrando la raza y el sexo
  - GetRaza y SetRaza
  - Algún método de instancia más Ej comer($comida) que le agradezca si le introducen pescado.

  Prueba las clases creadas mediante una aplicación que realice, al menos, las siguientes acciones:

  - Crear nuevas instancias de gato estableciendo raza y sexo, alguna de las 2 o ninguna.
  - Realiza echo sobre los objetos creados.
  - Prueba los diferentes métodos creados .

- **Ejercicio 2:** Crea la clase Abstracta Vehiculo, así como las clases Bicicleta y Coche como subclases de la primera.

  Para la clase Vehiculo,

  - El constructor actualizará la variable VehiculosCreados y inicializa a 0 KmRecorridos
  - Crea los métodos de clase getVehiculosCreados() y getKmTotales();
  - Crea método de instancia getKmRecorridos y recorre($km).

  Crea también algún método específico para cada una de las subclases y alguna variable de instancia (Ej cilindrada o piñones).

  - En el constructor creas el valor de la variable de instancia y llamas al constructor de la clase padre.
  - Método de instancia quemarueda() y hazcaballito()

  Prueba las clases creadas mediante una aplicación que realice, al menos, las siguientes acciones:

  - Anda con la bicicleta
  - Haz el caballito con la bicicleta
  - Anda con el coche
  - Quema rueda con el coche
  - Ver vehículos creados
  - Ver kilometraje de la bicicleta
  - Ver kilometraje del coche
  - Ver kilometraje total

- **Ejercicio 3(Serialización):** Crea la clase DadoPoker. Las caras de un dado de poker tienen las siguientes figuras: As, K, Q, J, 7 y 8 . Crea el método `tira()` que no hace otra cosa que tirar el dado, es decir, genera un valor aleatorio para el objeto al que se le aplica el método. Crea también el método `nombreFigura()`, que diga cuál es la figura que ha salido en la última tirada del dado en cuestión. Crea, por último, el método `getTiradasTotales()` que debe mostrar el número total de tiradas entre todos los dados. Realiza una aplicación que permita tirar un cubilete con cinco dados de poker.

  Nota: En el index.php crea 2 variables de sesión: Una con un array serializado con los 5 objetos y otra con las tiradas totales inicializada a 0. Comprueba siempre si existen con anterioridad antes de crearlas.

  Al final del index.php pon una frase que con f5 actualiza la página y puede seguir tirando el dado.

EJERCICIO DE AMPLIACIÓN

- **Ejercicio 4:** Queremos gestionar la venta de entradas (no numeradas) de Expocoches Campanillas que tiene 3 zonas, la sala principal con 1000 entradas disponibles, la zona de compra-venta con 200 entradas disponibles y la zona vip con 25 entradas disponibles. Hay que controlar que existen entradas antes de venderlas. Define las clase Zona con sus atributos y métodos correspondientes y crea un programa que permita vender las entradas. En la pantalla principal debe aparecer información sobre las entradas disponibles y un formulario para vender entradas. Debemos indicar para qué zona queremos las entradas y la cantidad de ellas. Lógicamente, el programa debe controlar que no se puedan vender más entradas de la cuenta