Tema 5 : POO

Ciclo Superior DAW

Asignatura: Desarrollo web en entorno servidor

Curso 20/21

Introducción

- En este capítulo veremos los siguientes conceptos:
  - Conocer los principios básicos de la programación orientada a objetos y su aplicación empleando el lenguaje PHP.
  - Conocer la sintaxis y las características de la programación orientada a objetos en el lenguaje PHP y usarla en el diseño de pequeños programas.
  - Ver las diferencias entre la realización de una aplicación empleando programación procedimental frente al uso de POO.

Programación orientada a objetos (POO)

- Es una técnica de programación que se acerca más a lo que pensamos las personas
- Proporciona herramientas que permiten al programador representar los elementos del problema (objetos) de manera general

Programación orientada a objetos (POO)

- Podemos pensar en cualquier elemento de la vida real como un sistema basado en objetos: un televisor, un teléfono, un coche, una persona…

- Pueden ser objetos que tienen unas determinadas características y capacidades. Por ejemplo, **un televisor** tiene un tamaño, resolución, y puede hacer tareas como encender, apagar, cambiar de canal, ...

Programación orientada a objetos (POO). Ejemplo

Gestión de las cuentas bancarias

- **Cuenta bancaria** sería una clase con propiedades como número de cuenta, saldo, tipo de cuenta y titular/es.

- **Cliente** sería una clase con propiedades como NIF, nombre completo, data de nacimiento, dirección, teléfono

Programación orientada a objetos (POO). Ejemplo

Gestión de las cuentas bancarias

- **Movimientos** sería una clase con información relativa a los movimientos que se realizan en esa cuenta bancaria.

Actividad 1

Utilizando POO, crea la gestión de un equipo de fútbol.

Consideraremos que existen futbolistas, entrenadores, staff, directiva y aficionados.

¿Qué diferentes clases y acciones tendremos?

POO. Clases

- Una clase es un molde del que después podremos crear múltiples objetos, con similares características
- Las variables que están definidas dentro de una clase reciben el nombre de atributos o propiedades.
- Las funciones que están definidas dentro de una clase reciben el nombre de métodos, que son usados para manipular sus propias propiedades

POO. Clases

- Su sintaxis es:

```php
<?php
class Nombre_clase
{
    //declaración de propiedades
    const CONSTANTE="valor";
    public $propiedad_1;
    public $propiedad_2;

    //declaración de métodos
    function método_1($parametro)
    {
    instrucciones_del_método;
    }
}
?>
```

POO. Clases

- Para definir una propiedad con valor variable:
    - En PHP5, debemos indicar el nivel de acceso que puede ser **public**, **private** o **protected**
    - En caso de que queramos asignarle un valor bastará con poner detrás del nombre de la variable el signo = y a continuación el valor.

POO. Clases

- Es recomendable que **cada clase figure en su propio fichero que debe tener como nombre el nombre de la clase que contiene.**
- Por ejemplo, **si tenemos una clase Libro**, esta **debería crearse dentro de un archivo de nombre Libro.php**.

POO. Clases

- Si se almacena cada clase en un archivo diferente, para poder utilizarlas debemos incluir una sentencia **include** o **require** por cada una de las clases que vayamos a usar.

```
require_once("Libro.php");
require_once("Revista.php");
require_once("Socio.php");
...
```

POO. Clases

- Una alternativa es usar la **función mágica __autoload**, que se ejecutará siempre que se intente utilizar una clase o interfaz que no fue definida hasta el momento. Por ejemplo:

```
function __autoload($clase) {
  require_once $clase.'.php';
}
```

Actividad 2

Utilizando POO, crea la gestión de un equipo de fútbol.

Consideraremos que existen futbolistas, entrenadores, staff, directiva y aficionados.

Define las clases y los atributos que va a tener cada una.

POO. Variable $this

- Esta variable es una variable especial de auto-referencia, que permite acceder a las propiedades y métodos de la clase actual.
- Por ejemplo:

```php
<?php
class Persona {
   private $nombre;
   function set_nombre($nom)
  {
    $this->nombre=$nom;
  }
   function imprimir()
  {
      echo $this->nombre;
  }
   }
?>
```

POO. Objetos

- **Un objeto es una instancia de una clase.**
- Para crear un objeto debemos usar la siguiente sintaxis:

```
$p = new Persona()
```

- *new* hace que la clase Persona junto con todas sus propiedades y métodos se copien a $p.
- Para acceder a sus métodos y atributos públicos:

```
$p->nombre = 'Lucía';
$p->imprimir();
```

Actividad 3

Considerando las clases definidas en las actividades previas, ¿cuántos objetos podríamos crear?

POO. Constantes

- **Una constante es un valor fijo establecido en la definición de la clase, que no se puede modificar en tiempo de ejecución.**
- **Para declarar una constante dentro de una clase usamos la palabra reservada *const*** antes del nombre de la constante que no debe llevar el carácter $.
- Por ejemplo:

```
class Calendario
{
    const NUM_MESES = 12;
}
```

POO. Constantes

- Para acceder al valor de la constante podemos usar el objeto o el nombre de la clase seguido de dos puntos dos veces (::)
- Por ejemplo:

```
echo Calendario::NUM_MESES;
echo $cal::NUM_MESES;
```

POO. Constantes

- PHP tiene algunas constantes predefinidas:
    - **__ CLASS__:** devuelve el nombre de la clase donde fue declarada
    - **__ METHOD__**: devuelve el nombre del método donde fue declarada
    - **__ FILE__**: Ruta completa y nombre del archivo. Usado dentro de un INCLUDE devolverá el nombre del fichero del INCLUDE.

POO. Constantes

- PHP tiene algunas constantes predefinidas:
    - **__ DIR__:** Directorio del fichero. Usado dentro de un INCLUDE, devolverá el directorio del archivo incluido.
    - **__ LINE__:** Línea actual del fichero.

Actividad 4

Define al menos dos constantes que podríamos necesitar para la gestión de las clases de las actividades anteriores

POO. Nivel de acceso

- Existen varios niveles de acceso que permiten controlar cómo acceder a las clases:
    - **Public**: Cualquiera puede acceder a las propiedades y métodos
    - **Private**: El acceso a las propiedades y métodos declarados como private están restringidos a la clase en la que fueron creados.
    - **Protected**: Sólo pueden acceder a estas propiedades y métodos la clase y sus clases derivadas.

POO. Nivel de acceso

```php
class Clase
{
    public $atributoPublico = "Atributo público";
    private $atributoPrivado = "Atributo privado";
    protected $atributoProtegido = "Atributo protegido";

    function imprimirAtributos()
    {
        echo $this->atributoPublico;
        echo $this->atributoPrivado;
        echo $this->atributoProtegido;
    }
}

$a = new Clase();

echo $a->atributoPublico;       //funcionaría correctamente
echo $a->atributoPrivado;       //erróneo
echo $a->atributoProtegido;     //erróneo
$a->imprimirAtributos();        //funcionaría correctamente
```

Actividad 5

Define el nivel de acceso correcto para los métodos y atributos de las clases creadas.

POO. Propiedades y métodos estáticos/no estáticos

- Las propiedades de una clase están encapsuladas en el objeto/clase y únicamente son accesibles a través de la clase o el objeto.
- Cuando creamos un objeto, las propiedades se copian a este, y a partir de aquí pueden modificarse usando el operador flecha ->.
- Para que los atributos o métodos de la clase sean accesibles sin necesidad de crear un objeto para acceder a ellos, debemos definirlos como **static**

POO. Propiedades y métodos estáticos/no estáticos

- Para acceder a una propiedad o método no se puede usar **$this->propiedad / $this->metodo**, sino que se debe emplear **self::propiedad / self::metodo.**
- Esto se debe a que:
    - **$this: Llamadas a propiedades y métodos que están dentro del objeto (no estáticos)**
    - **self: Palabra reservada que hace referencia al nombre de la clase actual.**

POO. Propiedades y métodos estáticos/no estáticos

- Por ejemplo, en la clase Data:

```
class Data {
    public static $calendario = "Calendario gregoriano";
    public static function getData(){
        $anho = date('Y');
        $mes = date('m');
        $dia = date('d');
        return $dia. '/'.$mes. '/'.$anho;
    }
}
```

- Para mostrar la propiedad:

```
echo Data::$calendario;
```

- Para llamar al método:

```
echo Data::getData();
```

Actividad 6

Comprueba los métodos creados.

Define si son estáticos o no estáticos.

Crea al menos un par de métodos estáticos para poder acceder sin instanciar clases

POO. Constructores y destructores

- Las clases tienen un método incorporado llamado **constructor**, que le **permite inicializar las propiedades de los objetos (dar valores a las propiedades) cuando instancias (creas) un objeto.**
- Si creamos una función con un nombre idéntico al nombre de la clase, éste se convertirá en constructor, y **se ejecuta de forma automática en el momento en que se define un nuevo objeto con new.**

POO. Constructores y destructores

- En el siguiente ejemplo vamos a crear un constructor para la clase Libro que recibe como parámetro el título del libro y actualiza el contador de libros por cada nuevo objeto que se crea.

POO. Constructores y destructores

```
class Libro {
    private $titulo;
    private static $cont_libros = 0;

    function Libro($titulo) {
     $ this->titulo = $titulo;
      self::$cont_libros++;
   }
  ....
```

POO. Constructores y destructores

- Al igual que el constructor se ejecuta automáticamente al crear un nuevo objeto, el destructor se ejecuta en el momento en el que el objeto deja de existir, con el cual debemos incluir en este las tareas que queramos ejecutar en el momento de liberar un objeto.

POO. Constructores y destructores

- Los métodos mágicos **__construct y __destruct** se ejecutarán automáticamente al crear o destruir un objeto de la clase.
- Un objeto puede ser destruido, por ejemplo, cuando es una variable local de una función y finaliza la ejecución de esa función.
- Podemos destruir un objeto de manera explícita empleando el método **unset()**, que permite destruir objetos pasándole como parámetro el nombre

POO. Constructores y destructores

```php
class Libro {
    private $titulo;
    private static $cont_libros = 0;

    function __construct($titulo) {
     $this->titulo = $titulo;
      self::$cont_libros++;
    }

    function __destruct() {
        self::$cont_libros--;
    }

    public static function num_libros(){
        return self::$cont_libros;
    }
}

$libro1 = new Libro("El niño mentiroso");
$libro2 = new Libro("Amigos");
echo "En la biblioteca tenemos ".Libro::num_libros()." libros. < br />";
unset($libro1);
echo "Si eliminamos uno pasamos a tener ".Libro::num_libros()." libros.<br/>
```

POO. Constructores y destructores

- El resultado será:

```
En la biblioteca tenemos 2 libros.
Si eliminamos uno pasamos a tener 1 libros.
```

Actividad 7

Para nuestra aplicación de gestión de un equipo de fútbol construiremos:

- *Dos constructores diferentes*
- *Un destructor para liberar los objetos*

POO. Implementar GET y SET en PHP

- Una de las principales características de la POO es la encapsulación de los datos que manejamos.
- No se debe acceder directamente desde el exterior a los datos que maneja un objeto.
- Se deben crear métodos que acceden al valor de una propiedad desde fuera del objeto.

POO. Implementar GET y SET en PHP

- Con el método **SET** establecemos un nuevo valor para la propiedad y con el **GET** lo obtenemos.
- Así posibilitamos que el objeto tenga control total sobre lo que hace con sus datos y simplificamos el acceso a estos desde el exterior

POO. Implementar GET y SET en PHP

- Por ejemplo, definimos los métodos:

```php
<?php
class Persona {
    private $nombre;
    function setNombre($nom) {
        $this->nombre = $nom;
    }
    function getNombre() {
        return $this->nombre;
    }
}
?>
```

POO. Implementar GET y SET en PHP

- Y los usamos:

```
$persona1 = new Persona();
$persona2 = new Persona();
$persona1->setNombre("Valeria");
$persona2->setNombre("Antía");
echo "El nombre del número 1 es: ". $persona1->getNombre();
echo "El nombre del número 2 es: ". $persona2->getNombre();
```

Actividad 8

Para nuestra aplicación de gestión de un equipo de fútbol implementaremos:

- *Métodos SET para establecer los atributos*
- *Métodos GET para obtener los atributos*

POO. Implementar GET y SET en PHP

- PHP5 incorpora dos métodos mágicos, **__get** y **__set**, que nos evitan tener que declarar un método set y get para cada propiedad de la clase.
- Su funcionamiento:
    - Cuando se intenta acceder a un atributo como si fuera público, PHP llama automáticamente a **__get**
    - Cuando establecemos el valor a un atributo llama al método **__set**

POO. Implementar GET y SET en PHP

- **__set** tendrá como parámetros de entrada el nombre del atributo y el valor que se le quiere dar
- **__get** únicamente tendrá como parámetro el nombre del atributo del cual quiere obtener el valor.

POO. Implementar GET y SET en PHP

- Por ejemplo:

```
class Persona {
    private $nombre;
    private $apellido1;
    private $apellido2;

    function __construct($nombre, $apellido1, $apellido2) {
     $this->nombre = $nombre;
     $this->apellido1 = $apellido1;
     $this->apellido2 = $apellido2;
   }
    public function __set($atributo, $valor) {
      if(property_exists(__ CLASS__, $atributo)) {
       $this->$atributo = $valor;
     }  else {
        echo "No existe el atributo $atributo.";
     }
   }
    public function __get($atributo) {
      if (property_exists(__ CLASS__, $atributo)) {
        return $this->$atributo;
     }
      return NULL;
   }
 }

 $persona1 = new Persona("Martin", "Garcia", "Figueira");
 echo $persona1->NIF; //intento mostrar un atributo que en el existe
 $persona1->nombre = "Jorge";  //cambio el valor del atributo nombre
 echo $persona1->nombre; //muestro el nuevo valor

 ?>
```

Actividad 9

Sustituiremos esos métodos SET y GET por los métodos mágicos __set y __get

¿El funcionamiento es el mismo?

POO. Funciones de clases/objetos

- Estas funciones nos permiten obtener informaciones relativa a las clases y objetos, como por ejemplo saber a qué clase pertenece un objeto o si un atributo pertenece o no la una clase.

POO. Funciones de clases/objetos

FUNCIÓN		EJEMPLO

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

get_class	Devuelve el nombre de la clase del objeto	echo "La clase es: " . get_class($p);
get_class_methods	Devuelve un array con los nombres de los métodos de una clase que son accesibles desde donde se hizo la llamada.	print_r(get_class_methods('Persona'));
get_class_vars	Devuelve un array con los nombres de los atributos de una clase que son accesibles desde donde se hizo la llamada.	print_r(get_class_vars('Persona'));
get_declared_classes	Devuelve un array con los nombres de las clases definidas.	print_r(get_declared_classes());
get_declared_interfaces	Devuelve un array con los nombres de las interfaces definidas.	print_r(get_declared_interfaces());
class_alias	Crea un alias para una clase.	class_alias('Persona', 'Habitante'); $p = new Habitante();

POO. Funciones de clases/objetos

FUNCIÓN		EJEMPLO
get_object_vars	Devuelve un array con las propiedades no estáticas del objeto especificado. Si una propiedad no tiene asignado un valor se devolverá con un valor NULL.	print_r(get_object_vars($p));

method_exists	Devuelve true si existe el método en el objeto o clase que se indica, y false en caso contrario, independientemente de si es accesible o no.	if (method_exists('Persona', 'imprimir') {
		...
		}

property_exists	Devuelve true si existe el atributo en el objeto o clase que se indica, y false en el caso contrario, independientemente de si es accesible o no.	if (property_exists('Persona', 'email') {
		...
		}

POO. Funciones de clases/objetos

- Además de estas funciones PHP dispone del operador **instanceof**, un operador de tipo que permite comprobar si una instancia es o no de una clase determinada.

```
if ($p_instanceof(Persona) {
    echo "El objeto es de la clase Persona";
}
```

Actividad 10

Utiliza al menos dos de las funciones de clases/objetos que vimos anteriormente para mejorar el funcionamiento de nuestra aplicación de gestión de un equipo de fútbol

POO. Referencias a objetos

- Una referencia en PHP5 es un **alias**, que permite que dos variables diferentes puedan escribir sobre el mismo valor.
- Nos permite acceder a un mismo valor desde nombres de variables diferentes y que se comporten como si fueran la misma variable.

POO. Referencias a objetos

- A continuación vemos un ejemplo:

```
$p = new Persona();
$p->nombre = "Jorge";
$a = $p;
```

- En la primera línea creamos un objeto de tipo Persona en memoria e indicamos que la variable $a es una referencia (apunta) al objeto creado.

POO. Referencias a variables

- El operador **=** crea un nuevo identificador a un objeto que ya existe.
- Si aplicamos este operador a variables de otros tipos (como números o cadenas de texto) crea una copia de la misma.
- **Para crear referencias a variables podemos usar el operador &**

POO. Referencias a variables

- Si al código anterior le añadimos:

```
$c = &$a;
```

- Tendremos una variable $c que hace referencia a la variable $a.
- A continuación veremos un ejemplo completo:

POO. Referencias a variables

```php
class Persona {
    private $nombre;
    public function __construct($nom)
    {
        $this->nombre=$nom;
    }
    public function getNombre()
    {
        return $this->nombre;
    }
    public function setNombre($ nom)
    {
        $this->nombre=$nom;
    }
}

$p = new Persona('Ángel');  //$p es una referencia al nuevo objeto creado
$a = $p;                    //$a es una referencia al mismo objeto

$c = &$a;                   //$c es una referencia a la variable $a

echo '<br/>Nombre de $a: '.$a->getNombre();
echo '<br/>Nombre de $c: '.$c->getNombre();

$c->setNombre('Carlos');    //cambiamos el nombre de la Persona empleando $c

$a = NULL;                  //Eliminamos la referencia que relaciona $a con el objeto
                            //con el cual $c también deja de estar relacionado con este.

                            // La siguiente línea daría error al perder la referencia
                            // de la variable $a al objeto
                            // echo '<br/>Nombre de $c: '.$ c-> getNombre();

echo '<br/>Nombre de $p cambiado a través de la referencia a la variable $c: '.$p->getNombre();

                            // $p sigue apuntando al objeto
```

POO. Referencias a variables

- Como vimos en este ejemplo, modificar cualquier referencia a un objeto implica modificar el objeto en sí mismo.
- La salida sería:

```
Nombre de $a: Ángel
Nombre de $c: Ángel
Nombre de $p cambiado usando la referencia a variable $c: Carlos
```

Actividad 11

Utiliza al menos dos de las funciones de clases/objetos que vimos anteriormente para mejorar el funcionamiento de nuestra aplicación de gestión de un equipo de fútbol

POO. Clonar objetos

- A veces queremos hacer una copia de un objeto, no tener dos referencias al mismo objeto. Empleando la palabra clave **clone** obtenemos dos instancias distintas con las mismas propiedades.
- Por ejemplo:

```
$p = new Persona();
$p->nombre = 'Ángel';
$s = clone($p);
```

POO. Clonar objetos

- Puede suceder que necesitemos hacer algún cambio en el objeto al clonarlo, por ejemplo, darle un nombre especial, limpiar algunas variables que contengan parámetros que ya no necesitemos, etcétera.
- Para eso existe el método mágico **__clone.**

POO. Clonar objetos

- El método mágico **__clone** se dispara automáticamente en el momento de clonar el objeto mediante la instrucción clone, permitiéndonos ejecutar sentencias al dispararse.
- A continuación, vemos un ejemplo:

POO. Clonar objetos

- Cambiamos el atributo nombre del objeto al clonarlo:

```
class Persona {
     private $localidad;
     private $nombre;

     public  function __construct($nom, $loc)
    {
        $ this->nombre=$nom;
        $ this->localidad=$loc;
    }
     public function __clone(){
        $this->nombre = 'Otro nombre';
    }
}

$p = new Persona('Pedro','Lugo');
$x = clone($p);

var_dump($p);
var_dump($x);
```

POO. Clonar objetos

- La instrucción clone disparó automáticamente el método mágico __clone que cambió el nombre del objeto clonado, dando como resultado:

```
object(Persona)[1]
   private 'localidad' => string 'Lugo' (length=4)
   private 'nombre' => string 'Pedro' (length=5)
object(Persona)[2]
   private 'localidad' => string 'Lugo' (length=4)
   private 'nombre' => string 'Otro nombre' (length=10)
```

Actividad 12

Utiliza al menos dos de las funciones de clases/objetos que vimos anteriormente para mejorar el funcionamiento de nuestra aplicación de gestión de un equipo de fútbol

POO. Comparar objetos

- PHP dispone de dos operadores que permiten comparar objetos:
  - El **operador de comparación simple** (==): dará como resultado VERDADERO si los objetos que se comparan son instancias de la misma clase y sus atributos tienen los mismos valores.
  - El **operador de identidad** (===): dará como resultado verdadero si las variables comparadas son referencias a la misma instancia.

POO. Comparar objetos

- Por ejemplo:

```
$p = new Persona();
$p->nombre = 'Sara';
$x = clone($p);
$a = &$p;
```

$x == $p VERDADERO

$x===$p FALSO

$a===$p VERDADERO

POO. Comparar objetos

- El resultado de:
    - **Comparar $x == $p será verdadero** ya que $x y $p son dos copias idénticas (con los mismos valores nos sus atributos)
    - **Comparar $x===$ p será falso** ya que $x y $ p no hacen referencia al mismo objeto
    - **Comparar $a===$p** será verdadero ya que $a y $p son referencias al mismo objeto.

Actividad 13

Utiliza al menos dos de las funciones de clases/objetos que vimos anteriormente para mejorar el funcionamiento de nuestra aplicación de gestión de un equipo de fútbol

POO. Implicación de tipos

- Consiste en especificar la clase a la que deben pertenecer los objetos que se pasen como parámetros a las funciones.

- Si cuando se realiza la llamada, el parámetro no es del tipo idóneo, se produce un error que se podrá capturar.

- Por ejemplo:

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

Actividad 14

Asegúrate que los métodos que hemos creado para nuestra aplicación de gestión de un equipo de fútbol cumplen con la implicación de tipos.

¿Qué pasaría si le pasamos un objeto de otra clase?

POO. Métodos interceptores o “mágicos”

- Métodos que se llaman automáticamente por PHP cuando ocurre alguna acción concreta sobre un objeto o clase
- Vamos a ver:
  - __isset
  - __unset
  - __toString
  - __invoke
  - __call
  - __callstatic
  - __sleep
  - __wakeup

POO. Métodos interceptores o “mágicos”

Método __isset

- Se dispara automáticamente cuando tratamos de **comprobar que un atributo existe** usando la función **isset()** o si **comprobamos su contenido empleando la función empty()**.
- Como parámetro recibe el nombre del atributo y debe devolver un booleano.

POO. Métodos interceptores o “mágicos”

Método __isset

- Por ejemplo:

```
class Persona {
        private $nombre;
        public function __set($nombre, $valor) {
            $ this->$nombre = $valor;
        }
        public function __isset($atributo) {
            return isset($this->$atributo);
        }
    }
$p = new Persona();
$p->nombre = "Pilar";
 if(isset($p->nombre)) {
     echo "atributo definido";
}
 else {
     echo "atributo no definido";
}
```

POO. Métodos interceptores o “mágicos”

Método __unset

- Se dispara automáticamente cuando **se intenta destruir un atributo que no existe o es privado empleando la función unset()**, lo que nos permite modificar el comportamiento de esta función.

POO. Métodos interceptores o “mágicos”

Método __unset

- Por ejemplo:

```
function __unset($atributo) {
        if(isset($this->$atributo))
            unset($this->$atributo);
    }
```

POO. Métodos interceptores o “mágicos”

Método __toString

- Devolverá un string **cuando se usa el objeto como si fuera una cadena de texto**, por ejemplo, dentro de un echo o de un print.
- Debe devolver una cadena de texto.

POO. Métodos interceptores o “mágicos”

Método __toString

- Por ejemplo:

```
class Persona
{
    private $_nombre;
    public function __construct($nombre){
    $this->_nombre = $nombre;
    }
    public function __toString(){
    return $this->_nombre;
    }
}

$p = new Persona ('Martin' );
echo $p; //Escribirá el nombre: Martin
```

POO. Métodos interceptores o “mágicos”

Método __invoke()

- Se disparará automáticamente **cuando se intenta llamar a un objeto como si fuera una función**.

POO. Métodos interceptores o “mágicos”

Método __invoke()

- Por ejemplo:

```
class Persona {
    public  function __invoke() {
    echo "Soy una persona";
    }
}

$p = new Persona();
$p();  //Esto llamaría al método mágico __invoke
```

POO. Métodos interceptores o “mágicos”

Método __call

- Se dispara automáticamente **cuando se llama a un método que no está definido en la clase o que es inaccesible dentro del objeto** (por ejemplo: cuando se trata de un método privado).
- Recibe el nombre del método empleado en la llamada, y un array con la lista de argumentos que le estábamos pasando.

POO. Métodos interceptores o “mágicos”

Método __call

- Por ejemplo:

```
class Clase
   {
        public function __call($metodo, $parametros){
           $str = "Método inaccesible: <br/>".$metodo.
                  "<br/>Parámetros:<br/>";
           // mostramos los parámetros pasados al método
            foreach($parametros as $parametro){
               $str.= " ".$parametro."<br/>";
           }
            echo $str;
       }
   }

$a = new Clase();
$a->metodoNoExiste(TRUE,'dato',23);
```

POO. Métodos interceptores o “mágicos”

Método __callstatic

- Se disparará automáticamente **si llamamos a un método estático**, aunque esté definido **__call**
- Por ejemplo, si en el código anterior tuviéramos

```
Clase::NoExiste();
```

- se ejecutaría el método __callstatic en el caso de estar definido.

POO. Métodos interceptores o “mágicos”

Método __sleep

- Permiten preparar un objeto para ser serializado y reconstruir lo que sea necesario tras una deserialización.
- Se disparará automáticamente **con la función serialize()**
- Devuelve un array con los atributos que queremos que se muestren en la representación del objeto (serialización)

POO. Métodos interceptores o “mágicos”

Método __sleep

- Por ejemplo:

```
public function __sleep() {
    return array("nombre", "email");  //indicamos los atributos a serializar
}
```

POO. Métodos interceptores o “mágicos”

Método __wakeup

- Se dispara automáticamente **cuando se aplica la función unserialize() sobre el objeto**. No acepta argumentos y no devuelve nada en especial. Sirve para restablecer conexiones con bases de datos o alterar algún atributo que se haya perdido con la serialización.

Actividad 15

Asegúrate que los métodos que hemos creado para nuestra aplicación de gestión de un equipo de fútbol cumplen con la implicación de tipos.

¿Qué pasaría si le pasamos un objeto de otra clase?

POO. Herencia

- La herencia es un concepto de POO con el que **una clase puede heredar todas las propiedades y métodos de otra y además añadir los suyos propios.**
- **Para que una clase descienda de otra se utiliza la palabra clave extends.**

POO. Herencia

- Por ejemplo, la clase Alumno y la clase Profesor podrían heredar el contenido de la clase Persona y, cada una de ellas, además podría tener características propias.

```
class Persona {
    protected $nombre;
    protected $apellidos;
    …
}
class Alumno extends Persona {
private $numExpediente;
}
```

POO. Herencia

- La herencia permite reutilizar eficientemente el código de la clase base.
- Podemos decir que Alumno es un tipo de Persona, y estaremos reutilizando el mismo código en dos clases distintas haciendo el código más ligero.

POO. Herencia

- Las **nuevas clases** que heredan **se conocen** también **con el nombre de subclases**.
- La **clase de la que heredan se llama clase base o superclase.**
- Los nuevos objetos que se instancien a partir de la subclase son también objetos de la superclase

POO. Herencia

- Podemos comprobar que un objeto pertenece a una superclase así:

```
$a = new Alumno();
if ($a instanceof Persona) {
    // como Alumno es un tipo de Persona cumpliría la condición
   …
}
```

POO. Herencia. Sobrescribir métodos

- Utilizando la herencia puede que necesitemos cambiar uno de los métodos de la clase base. Lo que tenemos que hacer es **sobrescribir el método.**
- Si sobreescribimos un método, este debe debe recibir los mismos parámetros que el padre
- La única excepción es el **constructor,** que puede redefinirse con los parámetros que se desee.

POO. Herencia. Sobrescribir métodos

- En este ejemplo se ejecutará el constructor de la subclase:

```
class Persona {
    protected $nombre;
    protected $apellidos;


    public function __contruct($nom)
    {
        $this->nombre=$nom;
    }
    …
}
class Alumno extends Persona {
    private $numExpediente;
    public function __contruct($exp, $nom)
    {
        $this->numExpediente=$exp;
        $this->nombre=$nom;
    }
}
$a= new Alumno(2019,'Martin');
```

POO. Herencia. Sobrescribir métodos

- Si queremos que se llame al de la clase deberemos solicitarlo explícitamente.

```
public function __contruct($exp, $nom)
{
    $this->numExpediente=$exp;
    parent::__construct($nom);
}
```

- **parent** hace referencia a la clase de la cual se está heredando.
- La instrucción **parent::metodo()** se puede hacer tanto sobre objetos como sobre métodos estáticos de clases.

Actividad 16

En esta actividad deberemos:

- *Comprobar si las clases que tenemos creadas comparten algún tipo de parámetro en común*
- *Crear una superclase (o varias) para reutilizar el código fuente y de la que hereden el resto de clases*

¿Cuándo no es recomendable la herencia? ¿Por qué?

POO. Herencia. Visibilidad

- El nivel de protección **protected** es similar al privado ya que bloquea el acceso fuera de la clase, pero permite que las clases hijas puedan acceder y manipular el atributo o método en cuestión.
- En herencia, los niveles nunca pueden ser más restrictivos en la subclase que en la clase base

POO. Herencia. Visibilidad

- Por ejemplo, si tenemos una clase base con un método protegido:

```
class Clase
{
    //Método Protegido
    protected function metodo(){
        echo "Método protegido de la clase base";
    }
}
```

¿Desde qué clases podremos llamarlo?

POO. Herencia. Visibilidad

- Podremos llamar a este método desde la subclase:

```
class ClaseHija extends Clase
{
        public function llamarMetodo(){
            $this->metodo();
        }

}
```

POO. Herencia. Visibilidad

- Si creamos un objeto perteneciente a subclase, para llamar al método protected de la clase padre **deberemos hacerlo usando el método público** declarado en esta

```
$c = new ClaseHija();
$c->llamarMetodo();
```

- ya que si intentamos acceder directamente al método protegido de la clase base **no funcionaría**.

```
$c->miMetodo(); //ERROR
```

POO. Herencia. Visibilidad

- Para aclarar:
    - Los métodos y propiedades privados se diferencian de los protegidos **sólo en caso de que exista herencia.**
    - Los métodos y propiedades privados son **visibles únicamente a la clase base** (u objetos pertenecientes a esta clase) En los hijos actúan como si no habían existido.

Actividad 17

En esta actividad deberemos:

- *Definir los tipos necesarios para los métodos y las propiedades que tenemos en nuestra aplicación.*

¿Podríamos poner alguno de los métodos como protected?

POO. Herencia. Impedir herencia

- Podemos querer **evitar que se pueda heredar de una clase**. Esto se hace añadiendo en la declaración de la clase el operador **final**.

- Por ejemplo, para **bloquear la herencia** para **Clase:**

```
final class Clase {...}
```

POO. Herencia. Impedir herencia

- Si lo que queremos es simplemente **evitar que se sobrescriba un método**, para así asegurarnos que siempre se había ejecutado tal y como se ha definido en la clase base, basta con añadir el operador final en su declaración:

```
public final function metodo(){ ... }
```

POO. Herencia. Ejemplo completo

- En este ejemplo veremos qué sucede cuando heredamos de una **clase que contiene un método estático.**

```
class Persona {
    public static function nuevaPersona() {
        self::$numPersonas++;
    }
    …
}
class Alumno extends Persona {
    private static $numPersonas = 0;
    …
}
class Profesor extends Persona {
    private static $numPersonas = 0;
    …
}
```

POO. Herencia. Ejemplo completo

- Así, cuando hacemos esto:

```
Alumno::nuevaPersona();
```

- Lo que queremos es que añada 1 a la variable $numPersonas de la clase Alumnos, y cuando hagamos:

```
Profesor::nuevaPersona();
```

- Queremos que añada 1 a la variable $numPersonas de la clase Profesor.

POO. Herencia. Ejemplo completo

- Sin embargo, el código anterior **no funciona** como esperamos pues, al compilar encuentra en el método nuevaPersona una referencia a self::$numPersonas, e intenta enlazar con una variable de la clase a que pertenece (Persona) que no existe.

POO. Herencia. Ejemplo completo

- Tiene que **enlazar con la variable en tiempo de ejecución, no en tiempo de compilación,** y **emplee las propiedades de la clase que hace la llamada al método**.
- Esto se hace **usando la palabra static en vez de self con el operador de resolución de ámbito ::**

POO. Herencia. Ejemplo completo

- También tendremos que definir las propiedades numPersonas como protected para darles acceso desde el método heredado. Esto es:

```
class Persona {
public static function nuevaPersona () {
        static::$numPersonas++;
    }
    …
}
class TV extends Alumno {
    protected static $numPersonas = 0;
    …
}
class Ordenador extends Profesor {
    protected static $numPersonas = 0;
    …
}
```

POO. Herencia. Funciones relacionadas

FUNCIÓN		EJEMPLO

get_parent_class	Devuelve el nombre de la clase padre del objeto o la clase que se indica.	class Alumno extends Persona {
		function soyHijo ()
		{
		echo "Soy hijo de ", get_parent_class($this);
		}
		}

is_subclass_of	Verifica si el objeto tiene esta clase como uno de sus padres.	$a= new Alumno();
		
		if (is_subclass_of($a, 'Persona')) {
		echo " \$a es una subclase de Persona";
		} else {
		echo "\$a no es una subclase de Persona";
		}

POO. Clases abstractas

- **Se declaran con palabra clave abstract** y son similares a las clases normales excepto en dos aspectos:
    - **No se pueden crear objetos a partir de ellas.** Es decir: una clase abstracta no puede ser instanciada.
    - **Puede incorporar métodos abstractos,** que son aquellos de los que solo existe su declaración, dejando la implementación para las clases hijas o derivadas.

POO. Clases abstractas

- Declararemos una clase como abstracta cuando sabemos que no va a ser instanciada y que su función es servir de herencia a otra clase.

- **Cualquier clase que defina un método abstracto debe ser declarada como abstracta** (ya que no ha implementado algún método y por tanto no se podrán instanciar objetos a partir de ella)

POO. Clases abstractas. Ejemplo

- Por ejemplo, si no vamos a trabajar con objetos de la clase Persona, la podríamos hacerla abstracta y emplearla como base para definir los métodos comunes a las clases que deriven de ella:

```
abstract class Persona {
    protected $nombre;
    protected $apellidos;
    abstract public function imprimir();
}
```

Actividad 18

En esta actividad deberemos:

- *Comprobar si las clases que estamos usando podemos hacer alguna de ellas abstracta*
- *Crea una clase abstracta. ¿Podemos crear objetos de esa clase? ¿Y de alguna subclase?*

POO. Polimorfismo

- Consiste en que **un mismo identificador o función pueda comportarse de manera diferente en función del contexto en el que sea ejecutado.**
- Por ejemplo, podemos tener una **clase abstracta que sirva de base a tres o cuatro clases hijas**, de manera que las funcionalidades compartidas estarán en la clase base y únicamente modificas o añades pequeños matices en las subclases.

POO. Polimorfismo

- En el siguiente ejemplo tenemos una clase Coche y dos clases derivadas de esta que consumirán diferente cantidad de gasolina cuando arrancan.

```
class Coche{
    protected $gasolina = 50;
    public function setGasolina($gas){
        $this->gasolina=$gas;
    }
    public function getGasolina () {
        return $this->gasolina;
    }
}
class Ferrari extends Coche{
    public function arrancar(){
        $this->gasolina -= 5;
    }
}
class Seat extends Coche{
    public function arrancar(){
        $this->gasolina -= 2;
    }
}
```

Actividad 19

- *¿Podemos aplicar polimorfismo utilizando clases abstractas? ¿Por qué?*
- *Comprueba si en nuestra aplicación de gestión de un equipo de fútbol podríamos aplicar polimorfismo*
- *Si se puede, aplícalo. Si no, crea una clase nueva a tu elección y aplica el polimorfismo*

POO. Interfaces

- **Las interfaces permiten establecer las características que debe cumplir una clase.**
- La **principal diferencia con una clase abstracta es que no se puede heredar de un interfaz. Un interfaz se implementa.**
- No permite definir atributos, solo métodos, y estos deberán ser siempre públicos.

POO. Interfaces

- **El interfaz es el recurso ideal para la implementación del polimorfismo, ya que los interfaces únicamente declaran funciones o métodos que deben ser codificadas en las clases que los implementan.**
- La manera de declarar una interfaz es similar a la declaración de una clase, sustituyendo la palabra **class** por **interface**.

POO. Interfaces

- Por ejemplo, un coche y un cartel luminoso no tienen nada en común, pero ambos han de ser encendidos y apagados en un momento dado.
- Por lo tanto, podríamos definir una interfaz que obligue a las clases que lo implementen a tener estos dos métodos.

```
interface Acciones {
        public function arrancar();
        public function apagar();
}
```

POO. Interfaces

- Para trabajar con una interfaz **se usa la palabra reservada implements.**

  Todos los métodos que están definidos en la interfaz deberán estar en las clases que la implementan, sino esto producirá un error.

```
class Seat extends Coche implements Acciones{
    public function arrancar(){
        ...
    public function avanzar($velocidad){
        ...
    }
```

POO. Interfaces

- Cada clase puede implementar más de una interfaz y para eso basta con separarlas por comas.
- Una clase no puede implementar interfaces que tengan funciones con el mismo nombre, ya que esto produciría una colisión.

POO. Interfaces. Funciones

FUNCIÓN		EJEMPLO
get_declared_interfaces	Devuelve un array con los nombres de los interfaces declarados	print_r(get_declared_interfaces());

interface_exists	Comprueba si una interfaz está definida	if (interface_exists('Acciones')) {
		class Clase implements Acciones
		{ ... }
		}

Actividad 20

Comprueba si alguno de los métodos que tenemos en nuestra aplicación se puede pasar a una interfaz.

En caso de que se pueda, asegúrate de que todas las clases implementen esa interfaz.

¿Puede una clase abstracta implementar a una interfaz? ¿Por qué?

POO. Traits

- **PHP no permite la herencia múltiple,** con el cual una clase no puede heredar de más de una clase a la vez.
- Los **Traits** permiten simular la herencia múltiple, pero no pueden ser instanciados.
- **Se usan con la palabra reservada use.**

POO. Traits

- Por ejemplo:

```
trait Saludos{
    private $buenosdias = "Buenos días!!!";
    public function getBuenosDias(){
      return $this->buenosdias;
    }
}
```

- Para usar este trait simplemente tenemos que usar la palabra reservada **use** después de declarar la clase:

```
class Persona{
   use Saludos;
   ...
}
$p = new Persona();
echo $p->getBuenosDias();
```

POO. Traits

- Los traits pueden ser utilizados en cualquier clase independientemente de la estructura jerárquica de herencia existente.
- Se pueden crear clases que utilicen dos o más traits al mismo tiempo, separando sus nombres por una coma.

```
class Persona {
    use Saludos, Funciones;
    …
}
```

POO. Espacio de nombres

- En PHP **no podemos tener dos clases o funciones con el mismo nombre**, ya que PHP no sabría cuál de las dos funciones tendría que ejecutar.
- Sin embargo, en ocasiones podemos necesitar tener dos elementos con el mismo nombre.

POO. Espacio de nombres

- Para declarar un espacio de nombres **debemos usar la instrucción namespace seguido del nombre que le queremos dar.**

- Esta instrucción se puede emplear de dos modos, aunque el primero es el recomendado.

POO. Espacio de nombres

- La instrucción **namespace debe ser la línea siguiente al tag de apertura de PHP**, con el cual afecta a todo el fichero.

```php
<?php
  namespace Prueba;
  class Persona {
    ...
}
```

POO. Espacio de nombres

- **Cerrando entre llaves el código perteneciente al espacio de nombres**.

  Esto permite dos o más espacios de nombres en un mismo fichero, aunque es aconsejable tener un único espacio de nombres por fichero.

```
namespace Prueba{
  class Persona {
    ...
  }
}
```

POO. Excepciones

- Una excepción puede ser lanzada y atrapada.
- **El control de excepciones es MUY IMPORTANTE**, ya que podremos controlar qué acciones realizar y qué mostrarle al usuario en caso de un error

POO. Excepciones

- El procedimiento a seguir es:
    - Meter el código susceptible de producir algún error para facilitar la captura de posibles excepciones dentro de un bloque **try**
    - Cuando se produce algún error, se lanza una excepción utilizando la instrucción **throw**.
    - Cada bloque **try** debe poseer como mínimo un bloque **catch**, que será el encargado de procesar el error en caso de que una excepción sea lanzada.
    - El bloque **finally** se ejecutará después de los bloques try y catch

POO. Excepciones

- PHP ofrece una **clase base Exception para utilizar como manejador de excepciones.**
- Para lanzar una excepción no es necesario indicar ningún parámetro, aunque de forma opcional se puede pasar un mensaje de error y un código de error.

POO. Excepciones

- Entre los métodos que se pueden usar con los objetos de la clase Exception están:
    - **getMessage**. Devuelve el mensaje, en caso de que se pusiera alguno.
    - **getCode**. Devuelve el código de error se existe.

POO. Excepciones

```
function division($dividendo, $divisor) {
   try {
     if ($divisor == 0) {
         throw new Exception('No se puede dividir por 0');
     }
     else return $dividendo/$divisor;
  } catch (Exception $y) {
       echo 'Excepción capturada: ', $e->getMessage(), "\n";
  }
}
```

Actividad 21

En esta actividad deberemos:

- *¿Tendría sentido crear un namespace para nuestra aplicación? ¿Por qué?*
- *Controlar las excepciones que se puedan dar en nuestra aplicación, ofreciéndole siempre información al usuario del error que se ha producido*

Tema 5 : POO

Ciclo Superior DAW

Asignatura: Desarrollo web en entorno servidor

Curso 20/21