Programación Orientada a Objetos en PHP

Antonio Espín Herranz

Objetos y Clases

- Introducción. Definición.
- Constructores / destructores.
- __toString()
- Ambito ::
- $this
- Visibilidad: private, protected, public.
- Propiedades y métodos: static.
- Constantes de clase / mágicas.
- Clonar objetos.
- Sobrecarga, herencia y polimorfismo.
- instanceof.
- final y abstract
- Interfaces y herencia de estos.
- Traits
- Clases Anónimas
- Excepciones
- Enumeraciones

Introducción

- Clase: descripción genérica o plantilla de un determinado tipo de objetos.

- Características de los objetos:
  - Estado: el valor de sus propiedades.
  - Comportamiento: funcionalidades que nos ofrecen (métodos).

- Métodos: función o subrutina asociada a un objeto.

Ejemplo de definición

```
class Persona {
    private $nombre;
    function asignaNombre($nuevoNombre){
        $this→nombre = $nuevoNombre;
    }
    function dameNombre(){
        return $this→nombre;
    }
}
```

Ejemplo de uso

- $unapersona = new Persona();
- $unapersona→asignaNombre('Pepe');
- echo $unapersona→dameNombre();

Constructores

- Cuando llamamos a new, creamos una instancia de la clase, con sus propios atributos o propiedades.
- El constructor se emplea para dar valores iniciales al nuevo objeto.
- Se denomina __construct(). Al constructor se le pueden definir parámetros como cualquier function o con parámetros por defecto.
- También se puede definir el constructor como un método que se llama igual que la clase. Para ser compatibles con versiones anteriores.
- PHP NO soporta la sobrecarga de constructores.

Ejemplo

```
class Persona {

    // Propiedades
    private $nombre;

    // Métodos
    function __construct() {
      $this->nombre = 'sin nombre';
    }
```

La llamada NO CAMBIA: $juan = new Persona();

Ejemplo

```
class Persona {

    // Propiedades
    private $nombre;

    // Métodos
    function __construct($nombre = 'sin nombre') {
      $this->nombre = $nombre;
    }
```

Llamadas válidas:

```
    $juan = new Persona(‘Juan’);
    $anonimo = new Persona();
```

Destructores

- La importancia de los destructores es limitada.
- PHP libera todos los recursos al finalizar la ejecución del script.
- El método se denomina __destruct().
- Y es llamado cuando desaparecen todas las referencias al objeto o se le llama explícitamente.

Ejemplo

```
class Persona {

  // Propiedades
  private $nombre;

  // Métodos
  function __construct($nombre = 'sin nombre') {
    $this->nombre = $nombre;
  }

  function __destruct() {
    echo "El objeto de nombre ", $this->nombre,
         " ha sido destruido<br />\n";
  }

  function asignaNombre($nombreAsignado) {
    $this->nombre = $nombreAsignado;
  }

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

} // class Persona
```

```
// Programa principal
  // ******************
  $fulanito = new Persona('Fulanito');

  $fulanito = NULL;
```

toString()

- Para imprimir un objeto con el valor de todos sus atributos.
- El método __toString() concatena el valor de todos los atributos del objeto.
- El método lo implementamos nosotros.

Ejemplo

```
class Persona {

    // Propiedades
    private $nombre;
    private $ape;

    // Métodos
    function __construct($nombre = 'sin nombre', $ape = 'sin ape') {
      $this->nombre = $nombre;
      $this->ape = $ape;
    }

function __toString() {
      return $this->nombre . " " . $this->ape;
}

// Uso:

 $fulanito = new Persona('Fulanito','Perez');

  // Escribimos directamente el objeto
  echo $fulanito, "<br />\n"; → SALTA EL MÉTODO __toString()
```

Ámbito ::

- Operador de resolución de ámbito.
- Es un elemento que permite acceder a algunas propiedades y métodos de una clase.
- En unión con el nombre de la clase.
- Podremos acceder a:
  - Métodos públicos.
  - Propiedades estáticas y sobrecargadas.
  - Constantes.

$this

- Proporciona una referencia al propio objeto.
- Junto con → nos permite acceder a propiedades y métodos del propio objeto.
- Me permite trabajar con el propio objeto, aplicar otros métodos o acceder a sus propiedades.

Ejemplo

```
class ClaseA {
    private $propA = 'Valor A';

    function accedePropA() {

        $cadena = 'valor(' . $this->propA . '), clase(' . get_class() . ')';
        // get_class() → Devuelve el nombre de la clase
        return $cadena;

    }
}

$a = new ClaseA();
echo $a->accedePropA(), "<br />";
```

Visibilidad

- Tenemos 3 tipos de visibilidad:
  - public : a una propiedad o método público se puede acceder desde cualquier parte del programa. (*Se asume por defecto*).
  - private: la propiedad o método sólo es accesible desde la clase donde se define.
  - protected: la propiedad o método es visible desde la propia clase o desde las clases que heredan de ella.

Ejemplo

```
class Clase1 {
   public    $pPublica1 = 'pPublica1';
   private   $pPrivada1 = 'pPrivada1';
   protected $pProtegida1 = 'pProtegida1';

   public function metodo1() {
     echo $this->pPublica1, "<br />\n";
     echo $this->pPrivada1, "<br />\n";
     echo $this->pProtegida1, "<br />\n";
   }
 }

 class Clase2 extends Clase1 {
   public    $pPublica2   = 'pPublica2';
   private   $pPrivada2   = 'pPrivada2';
   protected $pProtegida2 = 'pProtegida2';

   public function metodo2() {
     echo $this->pPublica1, "<br />\n";
     echo $this->pPrivada1, "<br />\n";
     echo $this->pProtegida1, "<br />\n";
     echo $this->pPublica2, "<br />\n";
     echo $this->pPrivada2, "<br />\n";
     echo $this->pProtegida2, "<br />\n";
   }
 }
```

```
// Instanciamos la clase Clase1
  $obj1 = new Clase1();
  $obj1->metodo1();

  echo "<hr />\n";

  // Instanciamos la clase Clase2 y llamamos al método
       heredado metodo1()
  $obj2 = new Clase2();
  $obj2->metodo1();

  // ahora llamamos a su propio método metodo2()
  echo "<hr />\n";
  $obj2->metodo2();
```

¿Son correctas todas las llamadas?

Propiedades y métodos static

- Una propiedad o método **static** pertenece a la clase en la que está definido, no a los objetos de la propia clase.
- La propiedad o método static puede ser llamado desde fuera del contexto de un objeto.
- OJO, pertenecen a la clase no al objeto.

Ejemplo

```
class Empleado {
   // Definimos el identificador de empleados como estático y privado
   private static $idEmpleados = 0;
   protected $id;     // Id. del empleado (protegido)
   public    $nombre; // Nombre del empleado (público)

   function __construct($nombreEmpleado) {
     self::$idEmpleados++;
     $this->id     = self::$idEmpleados;
     $this->nombre = $nombreEmpleado;
   }

   function __toString() {
     $cadena = 'id(' . $this->id . ') nombre(' . $this->nombre . ')';
     return $cadena;
   }
 } // class Empleado

 // Creamos 3 nuevos empleados
 $e1 = new Empleado('Empleado 1');
 $e2 = new Empleado('Empleado 2');
 $e3 = new Empleado('Empleado 3');

 // Mostramos sus valores
 echo $e1, "<br />\n";
 echo $e2, "<br />\n";
 echo $e3, "<br />\n";
```

self

referencia a la propia clase.

$this

al propio objeto.

Como un método static pertenece a la clase, se pueden llamar así:

Nombre_Clase::metodo(), dentro de estos métodos no se permite el uso de $this.

Constantes de clase

- Se pueden definir constantes dentro de una clase.
- Su valor debe ser escalar, es decir, tipos simples (números, booleanos o cadenas).
- Son sensibles a mayúsculas / minúsculas.
- Por convección se escriben en mayúsculas.
- OJO no pertenecen al objeto, si no a la clase. TENDREMOS QUE ACCEDER MEDIANTE **self**.

Ejemplo

- class Clase1 {

```
const CTE1 = “valor1”;

function metodo1(){
   echo self::CTE1;
}
```

Constantes mágicas

- PHP proporciona dos constantes que están disponibles dentro de cada clase.
- Son __CLASS__ y __METHOD__
- __CLASS__ almacena en nombre de la clase (se puede usar también get_class()).
- __METHOD__ contiene el nombre del método.
- También hay una función:
  - get_class_methods($objeto) : Devuelve un array indexado con los nombres de los métodos.

Ejemplo

```
<?php
  class Clase1 {

     function metodo(){
      echo  __CLASS__ . "<br />";
     }

     function metodo2(){
      echo  __METHOD__ . "<br />";
     }

  }

 // Principal:

     $ej = new Clase1();
     $ej->metodo();
     $ej->metodo2();
?>
```

- Como salida:
  - Clase1
  - Clase1::metodo2

Clonar objetos

- Clonar un objeto es obtener una copia del mismo.
- **ERROR** → $e2 = $e1 siendo e1 y e2 dos objetos.
- No obtengo una copia son dos referencias que apuntan al mismo objeto.
- Para clonar objeto usaremos:
  $e2 = clone $e1;
- **Problema**: con la clase empleado, si queremos asignar un identificador nuevo para el objeto clonado.
- Se soluciona implementando el método __clone() en la clase Empleado.

Ejemplo

```
class Empleado {
    private static $idEmpleado = 0;
    protected $id;

    function __clone(){
        self::$idempleados++;
        $this→id = self::$idempleados;
    }
}
```

Sobrecarga

- La sobrecarga se puede implementar a través de los métodos especiales:

  __set() y __get()

- Tiene la siguiente sintaxis:

  __set($nombrePropiedad, $valor); // Asigna.

  __get($nombrePropiedad); // Recupera.

- Y con el método __call.
  - Dos parámetros:
    - El nombre del método.
    - Y un array con los parámetros del método.

__get / __set

```
public function __set($var, $valor){
        // convierte a minúsculas toda una cadena la función strtolower
        $temporal = strtolower($var);
        // Verifica que la propiedad exista, en este caso el nombre es la cadena en "$temporal"
        if (property_exists('nombreDeLaClase',$temporal)) {
        $this->$temporal = $valor;
         } else {
        echo $var . " No existe.";
         }
 }

 public function __get($var) {
        $temporal = strtolower($var);
        // Verifica que exista

        if (property_exists('nombreDeLaClase', $temporal)) {
        return $this->$temporal;
         }

        // Retorna nulo si no existe
        return NULL;
}
```

Ejemplo

```
<?php
 class empleado{
 private nombre;
 private apellido;
 private dependencia;

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

public function __set($var, $valor)
{
// convierte a minúsculas toda una cadena la función
      strtolower
$temporal = strtolower($var);
// Verifica que la propiedad exista, en este caso el
      nombre es la cadena en "$temporal"
if (property_exists('empleado',$temporal))
 {
$this->$temporal = $valor;
 }
 else
 {
echo $var . " No existe.";
 }
 }

 public function __get($var)
 {
$temporal = strtolower($var);
// Verifica que exista

if (property_exists('empleado', $temporal))
 {
return $this->$temporal;
 }

// Retorna nulo si no existe
return NULL;
}
 }

// CODIGO PRINCIPAL
$empleado = new empleado('Andres','Lara');
$empleado → dependencia = 'Programador Junior';

echo $empleado → nombre . ' - '. $empleado → apellido .' Y la
      dependencia es: ' . $empleado → dependencia;
?>
```

Ejemplo

```
class SinPropiedades {

  public function __set($nombrePropiedad, $valor) {
    echo "Asignamos $valor a $nombrePropiedad\n";
    $this->$nombrePropiedad = $valor;
  }

  public function __get($nombrePropiedad) {
    echo "Acceso a la propiedad $nombrePropiedad (clase ",
         __CLASS__, ")\n";
    return $this->$nombrePropiedad;
  }

  public function __call($nombreMetodo, $argumentos) {
    echo "Acceso al método $nombreMetodo (clase ",
         __CLASS__, ")\nArgumentos:\n", var_dump($argumentos), "\n";

  }
} // class SinPropiedades
```

Ejemplo de uso

```
// Asignamos un valor a una propiedad que no está definida
$obj = new SinPropiedades();
$obj->nombre = 'Sergio';

// Mostramos un "volcado" del objeto
echo var_dump($obj), "\n";

// Accedemos a la propiedad sobrecargada y a otra inexistente
echo 'Nombre = ', $obj->nombre, "\n";
echo 'Otra   = ', @$obj->suNombre, "\n";

// Intentamos ejecutar un método inexistente
echo $obj->dameNombre('Sergio', 805, 2004);
```

Con la @ ignoramos el error, porque no está definida la propiedad. Saldría un mensaje del intérprete.

Herencia

- Definir clases a partir de otras.
- Las subclases pueden añadir nuevas propiedades, nuevas funcionalidades o sobrescribir funcionalidades de la clase padre.
- Podemos considerar las clases Padre mas generales y clases hijas como especializaciones.
- La **herencia en PHP es simple**, solo se puede heredar de una clase.
- No hay límite en los niveles de herencia.

Herencia

- Desde la clase hija nos podemos referir a la clase padre, mediante la palabra reservada **parent**.

```
class Persona {
        …
}
class Empleado extends Persona {
        …
}
```

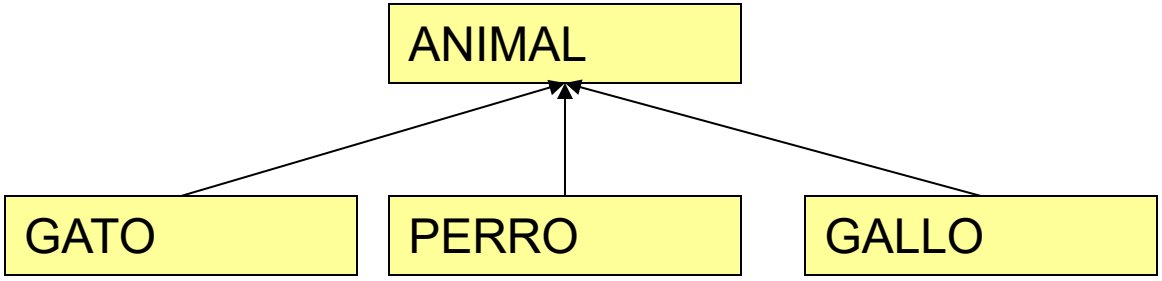
instanceof

- Es un operador que permite determinar si un objeto es una instancia de una determinada clase.

- Podemos resolver preguntas del tipo:
  - ¿Este objeto pertenece a esta clase?
  - if ($objeto instanceof $clase)
    ….

Polimorfismo

- Es la cualidad que tienen los objetos de comportarse de múltiples formas.
- Todas las subclases podemos decir que son animales.
- Cada una emite un sonido distinto.



Ejemplo

```
class Animal {
        function hacerSonido() { echo "Error:
            Este método debería ser
            implementado<BR>";
        }
}
class Gato extends Animal {
        function hacerSonido() {
            echo "miauuu<BR>";
        }
}
class Perro extends Animal {
        function hacerSonido(){
            echo "guauuu<BR>";
        }
}
class Gallo extends Animal {
        function hacerSonido(){
            echo "quiquiriqui<BR>";
        }
}
```

```
function imprimirSonido($obj) {
        if($obj instanceof Animal) {
            $obj->hacerSonido();
        } else {
            echo "Error: en el tipo de objeto";
        }
}

imprimirSonido(new Gato());
imprimirSonido(new Perro());
imprimirSonido(new Gallo());

// Salta el método correspondiente de
     cada clase.
```

final

- Se aplica a clases y a métodos
- Su cometido es impedir la redefinición del elemento sobre el que se aplica.
- La clase o método ya no se puede ni heredar ni sobrescribir.

abstract

- Podemos definir clases y métodos abstractos.
- Es decir, en un nivel superior de nuestra jerarquía a lo mejor no sabemos como se tiene que resolver un método y nos interesa que lo resuelvan las clases hijas.
- Por ejemplo: en la clase Animal, el método hacer sonido (es un método genérico).

abstract

- Si en una clase definimos un método como abstract estamos obligados a redefinirlo en las clases hija.
- Al definir un método como abstract esta clase se considera abstract (por definición).
- Los métodos abstractos se pueden propagar por el árbol de herencia.
- En un método abstracto sólo se indica la cabecera del método.

Ejemplo

```
abstract class Animal {
    abstract function hacerSonido(); // No tiene cuerpo.
}

class Gato extends Animal {
    function hacerSonido(){   // Redefine el método.
        echo ….
    }
}
```

interfaces

- Son similares a las clases abstractas.
- Permiten definir protocolos de comportamiento para los objetos en cualquier punto de la jerarquía.
- Una interface permite una definición de constantes y de métodos.
- Si una clase implementa una interface debe implementar todos los métodos de esta o definirlos como abstract.
- Una clase puede implementar mas de una interface.

Ejemplo

```
interface Leer {
    function tieneLibro($tituloLibro);
    function leeLibro();
}

class Empleado extends Persona implements Leer {
}

// La clase Empleado tiene que implementar los
    dos métodos de la interface.
```

Herencia vs Interfaces

- Una interface no puede tener propiedades ni implementar ningún método, mientras que esto si es posible en una clase abstracta.
- Una clase puede implementar varios interfaces pero, solo puede tener una clase padre.
- Una clase puede implementar mas de una interface.
- Una interface puede extender de varios interfaces.
- Las interfaces no forman parte de la jerarquía de clases.

Herencia de interfaces

- Permite herencia múltiple.
- Hay que tener cuidado con la colisión de constantes o definición de métodos entre los distintos interfaces.
- Ejemplo:

```
interface Perfil_1 implements Interface1, Interface2 {
}
```

Clases con tipos

- Se recomienda utilizar el tipado fuerte:

```
class Usuario {
    private string $nombre;
    private int $edad;

    public function saludar(string $mensaje): string {
        return "Hola $this->nombre, $mensaje";
    }
}
```

Atributos opcionales

```
class Usuario {
    private string $nombre;
    private ?string $email;

    public function __construct(string $nombre, ?string $email = null) {
        $this->nombre = $nombre;
        $this->email = $email;
    }
}

$u1 = new Usuario("Ana");
$u2 = new Usuario("Luis", "luis@example.com");
```

- La interrogación se coloca antes del tipo.

Definición de atributos en el constructor

```
class Usuario {
    public function __construct(
        private string $nombre,
        private ?string $email = null
    ) {}
}
```

Traits

- Desde PHP 8.2

- Un trait es como una **plantilla de métodos** que puedes incluir en una clase. No es una clase ni una interfaz, pero puede contener:
  - Métodos
  - Propiedades (desde PHP 8.2)
  - Métodos estáticos

Ejemplo

```
trait Logger {
    public function log(string $mensaje) {
        echo "[LOG]: $mensaje\n";
    }
}

class Usuario {
    use Logger;
}

class Producto {
    use Logger;
}

$u = new Usuario();
$u->log("Usuario creado");

$p = new Producto();
$p->log("Producto agregado");
```

Usuario y Producto comparten el método log() sin heredar de una clase común.

Posibles conflictos

```
trait A {
    public function saludar() {
        echo "Hola desde A";
    }
}

trait B {
    public function saludar() {
        echo "Hola desde B";
    }
}

class Demo {
    use A, B {
        B::saludar insteadof A;
        A::saludar as saludarDesdeA;
    }
}

$d = new Demo();
$d->saludar();         // Hola desde B
$d->saludarDesdeA();   // Hola desde A
```

Ventajas

- Reutilización de código sin herencia múltiple.
- Más modular y mantenible
- Compatible con clases externas

Clases Anónimas

- Se introdujeron a partir de **PHP 7**:

```
$saludo = new class {
    public function decirHola() {
        return "¡Hola desde una clase anónima!";
    }
};
// ¡Hola desde una clase anónima!
echo $saludo->decirHola();
```

Enumeraciones

- Representan constantes que están relacionadas para definir posibles situaciones o estados finitos:

```
enum EstadoPedido {
    case Pendiente;
    case Enviado;
    case Entregado;
    case Cancelado;
}
```

Enumeraciones

```
$estado = EstadoPedido::Enviado;

if ($estado === EstadoPedido::Enviado) {
    echo "Tu pedido ha sido enviado.";
}
```

Ejemplo completo

```
enum EstadoPedido {
    case Pendiente;
    case Enviado;
    case Entregado;
    case Cancelado;

    public function mensaje(): string {
        return match($this) {
            self::Pendiente => "Tu pedido está pendiente.",
            self::Enviado => "Tu pedido ha sido enviado.",
            self::Entregado => "Tu pedido fue entregado.",
            self::Cancelado => "Tu pedido fue cancelado.",
        };
    }
}

echo EstadoPedido::Entregado->mensaje(); // Tu pedido fue entregado.
```

self hace referencia a la propia clase

Apéndice

Palabra clave	Se refiere a...	Uso típico
`self`	La **clase actual** (estática)	Constantes, métodos estáticos
`$this`	La **instancia actual** (objeto)	Propiedades y métodos no estáticos
`static`	Contexto de **herencia dinámica**	Métodos estáticos heredables
`parent`	La **clase padre**	Acceder a métodos o constantes heredadas

namespace

- Un **namespace** en PHP es una forma de **organizar y agrupar clases, funciones y constantes** bajo un nombre lógico, para evitar conflictos entre nombres y facilitar la estructura del código.

- Los **namespaces** nos permiten tener la misma clase, pero en distintos ámbitos.

Ejemplo

```
// archivo: src/Modelo/Producto.php
namespace App\Modelo;

class Producto {
    public string $nombre;
}
```

Uso de una clase

- **Con use**
  - use App\Modelo\Producto;
  - $producto = new Producto();

- **Sin use:**
  - $producto = new \App\Modelo\Producto();

Organizar el proyecto

- **mi_proyecto**
  - **composer.json**
  - **doc**
    - pdfs
    - txt
    - docx
  - **web**
    - **backend**
      - **src**
        - » **data**
          - » **producto.php**
          - » **categoria.php**
        - » dao
          - » Acceso a la BD clases con PDO
        - » controller
          - » Controladores de los servicios REST
    - **frontend**
      - index.php

*El desacoplamiento entre el **backend** (enfocado a la persistencia de datos, Servicios, el modelo)*
*Y el **frontend** (enfocado a la capa de Presentación, de cara al cliente)*

*Favorece la actualización del proyecto y la sustitución de un **framework** por otro.*

Separación de perfiles en el desarrollo

Uso de composer

composer.json

```
{
  "autoload": {
    "psr-4": {
      "App\\": "web/backend/src/"
    }
  }
}
```

psr-4 es un estándar de PHP llamado:
PHP Standard Recommendation 4
Para el autoloading de clases y namespaces basado en Las rutas.

- Desde la raíz del proyecto, ejecutamos en un cmd: ***composer dump-autoload***

Uso de composer

- Al lanzar el comando saldrá un mensaje:

```
D:\apache\Apache24\htdocs\php8_avanzado\proyecto_empresa>composer dump-autoload
Generating autoload files
Generated autoload files
```

- Crea una carpeta **vendor** donde se irá dejando las librerías que descarguemos con **composer**.

Uso de las clases Data

- Utilizaremos, para importar las clases:

```
require __DIR__ . '/../../vendor/autoload.php';

// Ojo son las clases no el fichero:
use App\data\Producto
use App\data\Categoria
// Ojo con las rutas de las carpetas: respetar mayúsculas y minúsculas
$categoria = new Categoria(…);
```

Clases del mismo namespace

- Las clases que se encuentran en el mismo namespace no es necesario indicar la claúsula use.

Añadir librerías al proyecto

- **composer require nombre/libreria**