



UP4. Programación Orientada a Objetos en PHP

Desarrollo Web en Entorno Servidor

Profesora: Silvia Vilar Pérez

curso 2025-2026

Contenidos

- Clases en PHP.
- Clases y métodos abstractos
- Propiedades y métodos estáticos
- Operador de Resolución de Ámbito
- Herencia
- Interfaces
- Rasgos (Traits)
- Métodos mágicos
- Gestión de Excepciones
- Patrones de diseño

Clases en PHP

La clase establece las características, métodos y comportamiento de un objeto. El objeto es la instancia de una clase. Los métodos y atributos del objeto se acceden con **->** (**sin \$!!!**)

```
<?php /*definición*/
class ClaseSencilla
{
    // Declaración de una propiedad
    public $var = 'valor1';

    // Declaración de un método
    public function mostrarVar() {
        echo $this->var;
    }
}
?>
```

```
<?php /*instanciación*/
$instancia = new ClaseSencilla();

// Esto también se puede hacer con una variable:
$nombreClase = 'ClaseSencilla';
$instancia = new $nombreClase();
?>
```

```
<?php /*invocación de métodos y propiedades*/
echo $instancia->var; //devuelve valor1, el atributo var del objeto
$instancia->mostrarVar(); //invocamos al método y devuelve valor1
?>
```

Clases Abstractas

Una clase abstracta (creada con la palabra reservada **abstract**) es aquella que contiene métodos abstractos que sólo se declaran pero que se codifican en las clases descendientes.

Las clases abstractas no se pueden instanciar y sus descendientes están obligadas a implementar los métodos abstractos o a volver a definirlos como abstractos.

Si una clase tiene un método abstracto, la clase debe ser declarada como abstracta. Por otra parte, una clase abstracta puede contener métodos no abstractos.

Los métodos abstractos tienen dos requerimientos:

- No se pueden definir como privados puesto que necesitan ser heredados
- No pueden ser definidos como final porque deben ser redefinidos en las clases descendientes

Clases Abstractas - Ejemplo

```
abstract class Database {
    abstract public function connect($server, $username, $password, $database);
    abstract public function query($sql);
    abstract public function fetch();
    abstract public function close();
}

class MySQL extends Database {
    protected $dbh; // manejador de la BD
    protected $query; // código SQL
    public function connect($server, $username, $password, $database) {
        $this->dbh = mysqli_connect($server, $username, $password, $database);
        // https://www.php.net/manual/es/function.mysqli-connect
    }
    public function query($sql) {
        $this->query = mysqli_query($this->dbh, $sql); // https://www.php.net/manual/es/mysqli.query.php
    }
    public function fetch() {
        return mysqli_fetch_row($this->dbh, $this->query); // https://www.php.net/manual/en/mysqli-result.fetch-row.php
    }
    public function close() {
        mysqli_close($this->dbh); // https://www.php.net/manual/es/mysqli.close.php
    }
}
```

Propiedades y métodos estáticos

Los métodos y propiedades **static** pueden ser invocados sin necesidad de instanciar objetos de la clase (muy útil para contadores, aplicar formatos, conversiones de medidas, etc.) Por ello, se invocan usando **::** en lugar de **->** y en la misma clase se referencian con **self** (se refiere siempre a la clase en la que se crea) en lugar de usar **\$this** (no existe referencia al objeto o instancia)

Nota: las propiedades static se invocan con **::** e incluyen el **\$** de la variable.

```
<?php
class Format {
    public static $decimal=',';    // indica el formato para indicador de decimales
    public static $thousands='.'; // indica el formato para separador de miles
    public static function number($number, $decimals) {
        return number_format($number, $decimals, self::$decimal, self::$thousands);
    }
    public static function integer($number) {
        return self::number($number, 0); // se usa self en lugar de $this
    }
}
```

```
/* resultado */
1.234,57
1.235
```

```
print Format::number(1234.567,2) . "\n"; // Los métodos static se invocan con :: y no usan ->
print Format::integer(1234.567) . "\n";
```

?> Ver: <https://www.php.net/manual/es/language.oop5.static.php>

Operador de Resolución de Ámbito ::

El operador de resolución de ámbito (Paamayim Nekudotayim) o el doble dos-puntos, es un token que permite acceder a elementos estáticos, constantes, y sobrescribir propiedades o métodos de una clase.

Para acceder a las constantes de una clase, se debe utilizar el nombre de la clase (o **self** si estamos dentro de la clase) y el operador de ámbito :: como se muestra en el ejemplo:

```
class DB {  
    const USUARIO = 'php';  
    public function muestraUsuario() {  
        echo self::USUARIO; // desde dentro de la clase }  
}
```

```
echo DB::USUARIO; // desde fuera de la clase
```

Herencia

Para indicar que una subclase hereda desde la clase padre (herencia simple multinivel) se usa **extends**. Es necesario que la clase padre se defina antes que la subclase. Se heredan todos los métodos públicos y protegidos de la clase padre y se pueden sobrescribir los métodos heredados.

```
class forma {  
    function dibujar() { // pintar en pantalla }  
}  
class circulo extends forma {  
    function dibujar($inicio, $radio) {  
        // validar datos  
        if ($radio > 0) {  
            parent::dibujar();  
            return true;  
        }  
        return false;  
    }  
}
```

Interfaces

Las interfaces permiten definir el comportamiento de los objetos mediante la definición de métodos que serán implementados en dichos objetos. Se utiliza la palabra reservada **implements**. Con `class_implements()` podemos saber si una clase implementa una interface concreta

```
interface NombreInterface {  
    public function getNombre();    // Sólo se definen los métodos, no se implementan  
    public function setNombre($nombre);  
}
```

```
class Libro implements NombreInterface {  
    private $nombre;  
    public function getNombre() {    // Aquí es donde se implementan los métodos  
        return $this->nombre;  
    }  
    public function setNombre($nombre) {  
        return $this->nombre = $nombre;  
    }  
}
```

Rasgos (Traits)

Los rasgos o **traits** permiten reutilizar código entre objetos sin herencia. No se pueden instanciar y sus funciones se utilizan desde otras clases. La palabra reservada para incluir traits es **use**. Combinado los trait con interface se obtiene el máximo potencial de ambos. Una clase puede implementar la interface directamente o usar el trait que la implementa.

```
trait NombreTrait {  
    private $nombre;  
    public function getNombre() {  
        return $this->nombre;  
    }  
    public function setNombre($nombre)  
    {  
        return $this->nombre = $nombre;  
    }  
}
```

```
/* instancia la clase y usa los métodos */  
$l = new Libro();  
$l->getNombre();  
$l->setNombre("Mi libro");
```

```
/* usando únicamente trait en la clase */  
class Libro {  
    use NombreTrait;  
    // Tiene los métodos getNombre y setNombre  
}  
  
/* combinando interface y trait en la clase */  
class Libro implements NombreInterface {  
    use NombreTrait;  
}
```

Rasgos (Traits) Múltiples

Ejemplo con múltiples traits:

```
trait Hola {  
    public function saludo() {  
        echo "Hola";  
    }  
}  
trait Adios {  
    public function saludo() {  
        echo "Adiós";  
    }  
}
```

```
/* usamos los trait en la clase */  
class MiSaludo {  
    use Hola, Adios {  
        // Ambos tienen el método saludo()  
        // indicamos cuál queremos mostrar  
        Hola::saludo insteadof Adios;  
        Adios::saludo as despedida;  
    }  
}
```

Cuando varios traits comparten el nombre de los métodos, podemos elegir de qué trait vamos a usar el método con **insteadof**. También podemos usar los métodos de ambos traits indicando un alias con **as**

```
/* instanciamos la clase y usamos los métodos */  
$misaludo = new MiSaludo();  
$misaludo->saludo(); // imprime Hola  
$misaludo->despedida(); // imprime Adiós
```

Métodos mágicos

Los métodos mágicos se invocan cuando ocurren determinados sucesos o eventos que los activan. Con nombre **__metodo()**, determinan cómo reaccionará el objeto ante dichos eventos o sucesos. A continuación veremos algunos métodos mágicos:

__construct(): permite inicializar el objeto al crearlo con **new**

__destruct(): libera objetos sin referencias (unset) u otra finalización (exit, fin del script, etc.)

```
class usuario {  
    public $nombreusuario;  
    function __construct($nombreusuario, $password) {  
        if ($this->validar_usuario($nombreusuario, $password)) {  
            $this->nombreusuario = $nombreusuario;  
        }  
    }  
    function __destruct() {  
        echo "Ha finalizado su sesión";  
    }  
}
```

```
// usamos constructor para crearlo con datos  
$usuario = new usuario('Silvia', 'Dwes');  
//El final del script invoca al destructor
```

Métodos mágicos II

__toString(): permite controlar cómo se muestra un objeto cuando se imprime. **Nota:** Si hay valores no string, hacer casting a string al devolverlo

```
class Persona {  
    protected $nombre;  
    protected $email;  
    public function setNombre($nombre) {  
        $this->nombre = $nombre;  
    }  
    public function setEmail($email) {  
        $this->email = $email;  
    }  
    public function __toString() {  
        return "$this->nombre <$this->email>";  
    }  
}
```

Ejecutamos:

```
$silvia = new Persona;  
$silvia->setNombre('Silvia Vilar');  
$silvia->setEmail('silvia@php.net');  
print $silvia; // Muestra Silvia Vilar <silvia@php.net>
```

Métodos mágicos III

__get(): permite obtener propiedades private y protected del objeto

__set(): permite establecer propiedades private y protected del objeto

__isset(): invocado por isset() comprueba propiedad no public

__unset(): se invoca al usar la función unset() con propiedad no public

```
class Persona {  
    private $vdatos = array();           //almacenaremos sus datos en un array  
    public function __get($propiedad) {   //con get() no se puede acceder si no es public, __get sí puede  
        if (isset($this->vdatos[$propiedad])) {  
            return $this->vdatos[$propiedad];  
        } else {  
            return false;  
        }  
    }  
    public function __set($propiedad, $valor) { //con set() no se puede acceder si no es public, __set sí  
        $this->vdatos[$propiedad] = $valor;  
    }  
    public function __isset($property) { //permite comprobar propiedades private o protected  
        return isset($this->data[$property]);  
    }  
    public function __unset($property) {  
        if (isset($this->data[$property])) {  
            unset($this->data[$property]);  
        }  
    }  
}
```

```
$silvia = new Persona;  
$silvia->email = 'silvia@php.net'; // llama a  
__set(email, 'silvia@php.net')  
print $silvia->email; // llama a __get(email) y muestra  
silvia@php.net
```

```
}
```

Métodos mágicos III-bis

Hay quien recomienda no usar los métodos mágicos `__get` y `__set` y usar en su lugar funciones para implementar `get` y `set` de las propiedades del objeto. Algunas razones son:

- Estos métodos sólo se ejecutan cuando falla el acceso a propiedades no existentes o inaccesibles (acceden a `private` o `protected`).
- Sin validación (`property_exists` o `isset`), se puede modificar la estructura del objeto en tiempo de ejecución agregando propiedades.
- Son más lentos que los métodos `getX()/setX()`.
- Se requiere documentar explícitamente el uso de dichas funciones ya que imposibilita la documentación automática con herramientas como `phpDocumentor`. Además los IDE tienen dificultades para identificarlos y sugerir código (`phpIntellisense`).
- No se pueden utilizar con propiedades `static`.

Métodos mágicos IV

__call(): Permite que varios objetos se comporten como uno solo

__callStatic(): Realiza la función de __call() con métodos estáticos

```
class Persona {
    protected $nombre;
    protected $direccion;
    public function __construct() {
        $this->direccion = new Direccion;
    }
    public function setNombre($nombre) {
        $this->nombre = $nombre;
    }
    public function getNombre() {
        return $this->nombre;
    }
    public function __call($metodo, $argumentos) {
        if (method_exists($this->direccion, $metodo)) { //method_exists($object,$method_name):bool
            return call_user_func_array(array($this->direccion, $metodo), $argumentos);
        }
    }
}

class Direccion {
    protected $ciudad;
    public function setCiudad($ciudad) {
        $this->ciudad = $ciudad;
    }
    public function getCiudad() {
        return $this->ciudad;
    }
}

$silvia = new Persona;
$silvia->setNombre('Silvia Vilar');
$silvia->setCiudad('Valencia'); //llama a Direccion::setCiudad('Valencia') con __call
print $silvia->getNombre() . ' vive en ' . $silvia->getCiudad() . '.';
```

Métodos mágicos V

__sleep(): se invoca al serializar un objeto para almacenarlo como string. Se almacenan variables y nombre de clase pero no métodos

__wakeup(): se invoca al recuperar un objeto serializado y recupera conexiones de BD u otras tareas de reinicialización

```
class LogFile {  
    protected $filename;  
    protected $handle; // contiene puntero al fichero  
    public function __construct($filename) {  
        $this->filename = $filename; // Nombramos el fichero  
        $this->open(); // Lo abrimos para trabajar con él  
    }  
    private function open() {  
        $this->handle = fopen($this->filename, 'a'); // apertura para añadir al final, w sobrescribe  
    }  
    public function __destruct() {  
        if ($this->handle) { fclose($this->handle); }  
    }  
    public function __sleep() { // invocado con serialized(), obtendría array con propiedades a  
        return array('filename'); // serializar ($filename)  
    }  
    public function __wakeup() { // Invocado tras unserialize(): usa la propiedad $filename ya  
        $this->open(); // restaurada para reabrir el fichero.  
    }  
}
```

```
$logFile = new LogFile('mi_log.txt');  
$cadena_serializada = serialize($logFile);  
$file_recuperado = unserialize($cadena_serializada);  
$file_recuperado->write("¡Hola de nuevo!");
```

Gestión de Excepciones

Cuando se genera un error o un comportamiento inesperado en un script de PHP, se produce un **Error** o una **Exception**. Los errores pueden ser lanzados por múltiples funciones y clases de PHP y las excepciones son lanzadas por las funciones y clases definidas por el usuario y las excepciones de PHP.

La gestión de dichos errores y excepciones es el modo adecuado de tratar los errores, asegurar la consistencia de los datos y mostrar información útil al usuario para que pueda evitar volver a generar el error. Por otra parte, si no se maneja el error lanzado por PHP, éste provoca que el script se detenga

Para tratar Error y Exception, recurrimos al bloque **try... catch** con la posibilidad de lanzar la excepción a otro gestor con **throw**

IMPORTANTE: La jerarquía de Error no hereda de Exception con lo que deberemos dedicar un bloque `catch(Exception $e)` para las excepciones y un bloque `catch(Error $e)` para manejar los errores. Sin embargo, podemos acceder a ambas mediante la interfaz predefinida **Throwable**

Ejemplos Throwable

```
/* Ejemplo con Throwable */
class Mailer {
    private $transport;
    public function __construct(Transport $transport)
    {
        $this->transport = $transport;
    }
}

$transport = new stdClass(); //generará error porque
stdCalss no implementa/extiende Transport

/* Usamos Throwable, interfaz para capturar Error y
Exception en el mismo bloque Catch */
Try {
    //lanza TypeError porque $transport no es del tipo
    Transport
    $mailer = new Mailer($transport);
} catch (Throwable $e) {
    echo 'Caught!';
} finally {
    echo 'Cleanup!';
}
```

```
Try { /* Ejemplo con Exception */
    throw new Exception('Hello world');
} catch (Throwable $e) {
    echo 'Exception: '. $e->getMessage();
}
```

Podemos comparar la ejecución con Error y Throwable:

```
/*con clase Error*/
try {
    $resultado = 4 / 0;
} catch (DivisionByZeroError $e) {
    echo $e->getMessage(), "\n";
}
```

```
/* con Throwable */
try {
    $resultado = 4 / 0;
} catch (Throwable $e) {
    echo $e->getMessage(), "\n";
}
```

Patrones de diseño

Cabe destacar el MVC como el patrón de diseño basado en POO por excelencia en PHP.

En el **modelo**, se usan ORM (Object-Relational Mapping) de modo que las tablas relacionales se traducen en objetos con sus propiedades que interaccionan entre sí usando métodos para manipular los datos. Gracias a los modelos, se aplican **reglas de integridad** en cuanto a **validaciones** y **cálculos** (por ejemplo edad, o cómo debe ser el nombre en cuanto a tipo y longitud). Además **ejecutan** operaciones **CRUD** sobre la BD.

Para la **Vista** se usan plantillas para **renderizar** la información que le pasa el controlador para el usuario.

El **Controlador** recibe la petición del usuario (GET/POST), gestiona el **flujo de la aplicación** y contiene la **lógica de negocio y de aplicación** (reglas de autorización, procesos de negocio, coordinación de servicios, etc. **Gestiona** las operaciones **CRUD**