Tema 04 – Programación Orientada a Objetos PHP

2º DAW – Desarrollo Web Entorno Servidor

Profesor Juan Carlos Moreno

CURSO 2024/2025

Tabla de Contenido

4 Programación Orientada a Objetos PHP

4.1 Introducción

La programación orientada a objetos es una metodología de programación avanzada y bastante extendida, en la que los sistemas se modelan creando clases, que son un conjunto de datos y funcionalidades. Las clases son definiciones, a partir de las que se crean objetos. Los objetos son ejemplares de una clase determinada y como tal, disponen de los datos y funcionalidades definidos en la clase.

La programación orientada a objetos permite concebir los programas de una manera bastante intuitiva y cercana a la realidad. La tendencia es que un mayor número de lenguajes de programación adopten la programación orientada a objetos como paradigma para modelizar los sistemas. PHP implanta la programación de objetos como metodología de desarrollo. También Microsoft ha dado un vuelco hacia la programación orientada a objetos, ya que .NET dispone de varios lenguajes para programar y todos orientados a objetos.

Así pues, la programación orientada a objetos es un tema de gran interés, pues es muy utilizada y cada vez resulta más esencial para poder desarrollar en casi cualquier lenguaje moderno. En este tema vamos ver algunas nociones sobre la programación orientada a objetos en PHP. Aunque es un tema bastante amplio, novedoso para muchos y en un principio, difícil de asimilar, vamos a tratar de explicar la sintaxis básica de PHP para utilizar objetos, sin meternos en mucha teoría de programación orientada a objetos en general.

Un **objeto** es una entidad independiente con sus propios datos y programación. Las ventanas, menús, carpetas de archivos pueden ser identificados como objetos; el motor de un auto también es considerado un objeto, en este caso, sus datos (atributos) describen sus características físicas y su programación (métodos) describen el funcionamiento interno y su interrelación con otras partes del automóvil (también objetos).

El concepto renovador de la tecnología Orientación a Objetos es la suma de funciones a elementos de datos, a esta unión se le llama **encapsulamiento**.

Por ejemplo, un objeto página contiene las dimensiones físicas de la página (ancho, alto), el color, el estilo del borde, etc, llamados atributos. Encapsulados con estos datos se encuentran los métodos para modificar el tamaño de la página, cambiar el color, mostrar texto, etc. La responsabilidad de un objeto pagina consiste en realizar las acciones apropiadas y mantener actualizados sus datos internos. Cuando otra parte del programa (otros objetos) necesitan que la pagina realice alguna de estas tareas (por ejemplo, cambiar de color) le envía un mensaje. A estos objetos que envían mensajes no les interesa la manera en que el objeto página lleva a cabo sus tareas ni las estructuras de datos que maneja, por ello, están ocultos.

El lenguaje PHP tiene la característica de permitir programar con las siguientes metodologías:

- Programación Lineal: Es cuando desarrollamos todo el código disponiendo instrucciones PHP alternando con el HTML de la página.

- Programación Estructurada: Es cuando planteamos funciones que agrupan actividades a desarrollar y luego dentro de la página llamamos a dichas funciones que pueden estar dentro del mismo archivo o en una librería separada.
- Programación Orientada a Objetos: Es cuando planteamos clases y definimos objetos de las mismas (Este es el objetivo del tutorial, aprender la metodología de programación orientada a objetos y la sintaxis particular de PHP 5 para la POO)

4.2 Clase

Una clase, es un «molde» a partir del cual podemos crear objetos. Cuando definimos una clase, debemos determinar también qué propiedades y métodos tendrán los objetos que crearemos a partir de ella, con ellos nos referimos a un concepto importante de la POO llamado abstracción.

Una clase está compuesta por un conjunto de:

- Atributos. Los atributos de una clase son sus características, y podemos pensar en ellas como variables que se definen con un valor inicial
- Métodos. Las acciones que es capaz de hacer la clase y normalmente se escriben con verbos en infinitivo, si nos fijamos en un ejemplo de la vida real, un gato, sus acciones serían correr, morder, comer, saltar, etc. Representan el conjunto de funciones del método.

4.2.1 Declaración Clase

La sintaxis básica para declarar una clase es:

```
class [Nombre de la Clase] {
  [atributos]
  [métodos]
}
```

Siempre conviene buscar un nombre de clase lo más descriptiva posible a lo que representa. La palabra clave para declarar la clase es *class*, seguidamente el nombre de la clase y luego encerramos entre llaves de apertura y cerrado todos sus atributos (variable) y métodos (funciones).

Actualmente existe una convención por el que la letra inicial del nombre de la clase va en mayúsculas y además empieza por Class_nombreclase. Ejemplo si quisiera definir la clase persona sería: Class_persona

Ejemplo definición de la **clase persona**:

```
class Class_persona {
      private $nombre;
      public function __construct()
      {
            $this->nombre=null;
      }
      Public function getNombre() {
            return $this->nombre;
      }
```

```
      Public function setNombre($pNombre) {
            $this->nombre = $pNombre;
      }
      publicfunction imprimir()
      {
      echo $this->nombre;
      echo '<br>';
      }
}
```

Ejemplo efinición de la **clase vehículo**

```
<?php
Class Class_vehiculo {

      public $matricula;
      public $velocidad;

      public function __construct() {

            $this->matricula = null;
            $this->velocidad = null;

      }

      Public functionget Matricula() {
      return $this->matricula;

      }

      Public function getVelocidad() {
            return $this->velocidad;
      }

      Public function setMatricula($pMatricula) {
            $this->matricula = $pMatricula;
      }

      Public function setVelocidad($pVelocidad) {
            $this->velocidad = (float) $pVelocidad;
      }

      Public function aumentarVelocidad() {
            $this->velocidad++;
      }

      Public function disminuirVelocidad() {
            $this->velocidad--;
      }

      publicfunction parar() {
            $this->velocidad = 0;
       }
```

```
    }
?>
```

4.2.2 Variable $this

Cuando en el cuerpo de los métodos nos referimos a las propiedades para asignar algún valor, lo hacemos mediante la palabra reservada $this-> y, a continuación, el nombre de la variable sin el signo $.

La palabra $this hace referencia a «este objeto», esta instancia, la que está ejecutando el método. Decir $this->number = 5 es lo mismo que decir: «asignar el valor 5 a la variable "number" de este objeto de la clase en el que estamos»

4.3 Objetos.

Una vez definida la clase que servirá de modelo para nuestro trabajo, ya podemos empezar a crear objetos a partir de ese modelo. Cada objeto que hagamos a partir de una clase se denomina «instancia».

4.3.1 Crear objeto

La sintaxis para crear un objeto a partir de una determinada clase es la siguiente:

```
$nomObjeto = new nomClase();
```

Aunque lo veremos más adelante, lo que en realidad hace la sentencia new es ejecutar el método __construct() asociada a la clase.

Ahora veamos cómo podríamos definir yusar un objeto de la clase Persona:

```
//Crea el objeto
$per1 = new Class_persona();

//Asigna al objeto el nombre Juan mediante el método setNombre
$per1->setNombre('Juan');

//Muestra el nombre del objeto persona
echo$per1->getNombre();

//Error: nombre es atributo privado y no puede usarse fuera de la
//clase
echo $per1->nombre;
```

4.3.2 Uso objetos

Desde esteinstante, $per1 ya es un objeto, a pesar de que parece una variable a simple vista, es una instancia de la clase persona. Por esta razón, ya posee todas las propiedades definidas dentro de la clase y también la capacidad de ejecutar cualquiera de sus métodos. Además en el mismo momento en que creamos la instancia —cuando se ejecuta new—, se ejecuta un método llamado constructor.

Luego para llamar a los métodos debemos anteceder el nombre del objeto el operador -> y por último el nombre del método. Para poder llamar al método, éste debe ser definido público (con la palabra clave *public*). En el caso que tenga parámetros se los enviamos:

```
$per1->setNombre('Juan');
```

También podemos ver que podemos definir tantos objetos de la clase Persona como sean necesarios para nuestro algoritmo:

```
$per2 = new Class_persona();
$per2->setNombre('Ana');
$per2->imprimir();
```

4.3.3 Ejemplo Definición Clases y Objetos

Definición y uso de la clase persona:

```
<?php
      //Creamos el objeto ejectuando __construct()
      $per1= new Class_persona();
      $per1->setNombre('Juan');
      $per1->imprimir();
      $per2 = new Class_persona();
      $per2->setNombre('Ana');
      $per2->imprimir();
?>
```

Definición y uso de la clase vehículo

```
<?php

      $coche1 = new Class_vehiculo();
      $coche1->setMatricula('0101 BFH');
      $coche1->setVelocidad(10)
      $coche1->aumentarVelocidad();
      // velocidad = 11

      $coche2 = new Class_vehiculo();
```

4.4 Método constructor y destructor.

4.4.1 Método __construct()

PHP permite a los desarrolladores declarar métodos constructores para las clases. Aquellas que tengan un método constructor lo invocarán en cada nuevo objeto creado, lo que lo hace idóneo para cualquier inicialización que el objeto pueda necesitar antes de ser usado.

Algunas de las características básicas de este método son:

- El constructor es el primer método que se ejecuta cuando se crea un objeto.
- El constructor se llama automáticamente.
- Objetivo básico dentro de la programación orientada a objetos (POO).

Otras características de los constructores son:

- Un constructor no puede retornar dato.
- Un constructor puede recibir parámetros que se utilizan normalmente para inicializar atributos.
- Es un método opcional, no es obligatorio su definición.

Veamos la sintaxis del constructor:

```
publicfunction __construct([parámetros])
  {
    [algoritmo]
  }
```

Veamos el uso del método __construct en la clase clientes

```
class Class_cliente{

      $nombre;
      $numero;
      $peliculas_alquiladas;

      function __construct(
            $nombre=null,
            $numero=null,
            $peliculas_alquiladas = []
            )
      {
      $this->nombre=$nombre;
      $this->numero=$numero;
      $this->peliculas_alquiladas = $peliculas_alquiladas;
      }

      Function dame_numero(){
            return $this->numero;
      }
}

//instanciamos un par de objetos cliente
$cliente1 = new Class_cliente("Pepe", 1);
$cliente2 = new Class_cliente("Roberto", 564);

//mostramos el numero de cada cliente creado
echo "El identificador del cliente 1 es: " . $cliente1->dame_numero();
echo "El identificador del cliente 2 es: " . $cliente2->dame_numero();

//El identificador del cliente 1 es: 1
//El identificador del cliente 2 es: 564
```

4.4.2 Método _destruct().

Los destructores son funciones que se encargan de realizar las tareas que se necesita ejecutar cuando un objeto deja de existir. Cuando un objeto ya no está referenciado por ninguna

variable, deja de tener sentido que esté almacenado en la memoria, por tanto, el objeto se debe destruir para liberar su espacio. En el momento de su destrucción se llama a la función destructor, que puede realizar las tareas que el programador estime oportuno realizar.

La creación del destructor es opcional. Sólo debemos crearlo si deseamos hacer alguna cosa cuando un objeto se elimine de la memoria.

Las características de este método son:

- El objetivo principal es liberar recursos que solicitó el objeto (conexión a la base de datos, creación de imágenes dinámicas etc.)
- Es el último método que se ejecuta de la clase.
- Se ejecuta en forma automática, es decir no tenemos que llamarlo.
- Debe llamarse __destruct.
- No retorna datos.
- Es menos común su uso que el constructor, ya que PHP gestiona bastante bien la liberación de recursos en forma automática.

En el código siguiente vamos a ver un destructor en funcionamiento. Aunque la acción que realiza al destruirse el objeto no es muy útil, nos puede servir bien para ver cómo trabaja.

```
<?php

class Class_persona {
      public $nombre;
      public $apellido;
      public $edad;

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

      Public function __destruct() {
            echo 'Objeto destruido';
      }
      Public function saludar(){
            return 'Hola, soy ' . $this->nombre . ' ' .
                    $this->apellido .' y tengo ' .
                    $this->edad .' años ';
    }
 }

$persona = new Class_Persona('Fernando', 'Gaitan', 26);
echo $persona->saludar();
```

```
//Destruimos el objeto __destruct() mostrará Objeto Destruido
unset($persona);
```

4.5 Getters y Setters

Con la incorporación de atributos de tipo *public* estaríamos violando una de las reglas básicas de la POO, la **encapsulación** de los datos internos, que solo deberían poder accederse mediante la ejecución de los métodos de dicha clase. Desde el código de nuestra implementación no se debería dar un valor a una propiedad directamente, y tampoco leer datos directamente, para eso existen los **getter** y **setter**, para realizar estas tareas tan comunes:

La estructura de un método **getter** por lo general es sumamente sencilla, no recibe ningún parámetro, y se limita a devolver mediante el *return* el valor de una propiedad.

Sintaxis básica de un **getter**.

```
Public function getNumber(){
      return $this->number;
}
```

Por el contrario, la estructura de un método **setter** sí debe recibir un parámetro obligatoriamente, que será el valor que se le asignará a la propiedad. No devolverá nada con return, simplemente le asignará un valor.

```
Public function setNumber($value){
    $this->number = $value;
}
```

Veamos el ejemplo completo de square.php

```
Class Class_square {

      private $number;

      public function setNumber($value){
        $this->number = $value;
      }

      Public function getNumber(){
            return $this->number;
      }

      Public function area(){
            return $this->number * $this->number;
      }
}
```

Que si se incluye en un index.php quedaría

```
Require_once("Square.php");

$instance = new Class_quare();
$instance->setNumber(12);

echo $instance->area();
```

4.6 Herencia

La herencia indica que se pueden crear nuevas clases partiendo de clases existentes, heredando todos los atributos y los métodos de su 'superclase' o 'clase padre' y además se le podrán incorporar nuevos atributos y métodos propios.

En PHP, a diferencia de otros lenguajes orientados a objetos (C++), una clase sólo puede derivar de una única clase, es decir, PHP no permite herencia múltiple.

4.6.1 Superclase o clase padre

Clase de la que desciende o deriva una clase. Las clases hijas (descendientes) heredan (incorporan) automáticamente los atributos y métodos de la clase padre.

4.6.2 Subclase

Clase descendiente de otra. Hereda automáticamente los atributos y métodos de su superclase. Es una especialización de otra clase. Admiten la definición de nuevos atributos y métodos para aumentar la especialización de la clase.

La sintaxis para crear clases hijas o subclases sería:

```
Class nomClaseHija extends nombClasePadre{
      [atributos]
      [métodos]
    }
```

En el siguiente ejemplo vemos cómo a partir de la superclase Vehículo, creo la subclase deportivo, con la incorporación de nuevos atributos y métodos.

```
<?php
class Class_deportivo extends Class_vehiculo {

      private $aleron_deportivo;
      private $cristales_tintados;
      private $musica;

      public function __construct() {

            $this->aleron_deportivo = true;
            $this->cristales_tintados = true;
            $this->musica = 0;
       }

      Public function ponerMusica() {
            $this->musica = 1;
```

```
      }

      Public function apagarMusica() {
            $this->musica = 0;
      }

}

    $deportivo = new Class_deportivo();

    $deportivo->aumentarVelocidad();
    // velocidad = 1
    $deportivo->aumentarVelocidad();
    // velocidad = 2
    $deportivo->ponerMusica();
    // musica = 10
```

4.6.3 Herencia y constructores

Los constructores padres no son llamados implícitamente si la clase hija define un constructor. Para ejecutar un constructor padre, se requiere invocar a parent::__construct() desde el constructor hijo. Si el hijo no define un constructor, entonces se puede heredar de la clase madre como un método de clase normal (si no fue declarada como privada).

```
class Class_producto {
      protected $id;
      protected $titulo;
      protected $precio;
      protected $nombreAutor;
      protected $apellidosAutor;

      function __construct(
            $id=null,
            $titulo=null,
            $precio=null,
            $nombreAutor=null,
            $apellidosAutor=null
      ) {
            $this->id = $id;
            $this->titulo= $titulo;
            $this->nombreAutor = $nombreAutor;
            $this->apellidosAutor = $apellidosAutor;
            $this->precio = $precio;
      }
      Public function getNombreAutor() {
            return $this->nombreAutor
      }
      Public function getApellidosAutor() {
            return $this->apellidosAutor
      }

      Public function getTitulo(){
            return $this->titulo;
      }
```

```
      Public function getPrecio(){
            return $this->precio;
      }
 }

class Class_libro extends Class_producto {
      public function getResumen() {
            $resumen = "Titulo: " . $this->getTitulo() . ", Precio: " .
            $this->getPrecio();
            $resumen .= ", Autor: " .$this->getAutor() . ", Núm.
            páginas: " . $this->getNumPaginas();
            return $resumen;
      }
 }
```

Como no hemos definido ninguno constructor específico en esta clase, se heredará el de la clase base. Por lo tanto cuando creamos un objeto instanciado de la clase Libro deberemos de añadir los parámetros que necesitaba el constructor de la clase base, aunque tal y como está declarado si no pasamos los argumentos le asigna valor nulo a cada uno de los atributos del nuevo objeto de la clase libro.

```
$libro1 = new Class_libro(1,"título",20,"Autor", "Apellido1
Apellido2");
      echo $libro1->getResumen();
```

Pero veamos ahora el siguiente ejemplo:

```
class Class_libro extends Class_producto {
      private $numPaginas;
      function __construct(
            $id=null,
            $titulo=null,
            $precio=null,
            $nombreAutor=null,
            $apellidosAutor=null,
            $numPaginas=null){

            Parent::__construct($id, $titulo, $precio, $nombreAutor,
                  $apellidosAutor);
            $this->numPaginas = $numPaginas;
      }

      Public function getNumPaginas() {
            return $this->numPaginas;
      }

      Public function getResumen() {
            $resumen = "Titulo: " . $this->getTitulo() . ", Precio: " .
            $this->getPrecio();
            $resumen .= ", Autor: " .$this->getAutor() . ", Núm.
            páginas: " . $this->getNumPaginas();
            return $resumen;
      }
}
```

Hemos creado un constructor que necesita los argumentos de la clase base y el nuevo argumento ($numPaginas). Y como nuestro constructor es un añadido al constructor base, hemos llamado a este con los argumentos necesarios y posteriormente hemos añadido la nueva funcionalidad.

```
$libro1 = new Class_libro(1,"título",20,"Autor", "Apellido1
Apellido2",440);
echo $libro1->getResumen();
```

4.7 Tipos Atributos y Métodos

Cuando creamos un objeto de una clase determinada, los atributos declarados por la clase son localizadas en memoria y pueden ser modificados mediante los métodos.

Los tipos de atributos de una clase coinciden con los tipos de variables ya vistos en PHP. Ahora bien según su visibilidad se pueden declarar 4 tipos de atributos:

- Público
- Privado
- Protegido

4.7.1 Atributos Públicos. *Public*

Significa que se puede interactuar con ellos desde el exterior. Por defecto un atributo es declarado de tipo *public*.

```
<?php
class Class_coche {
      $velocidad;

      function __construct() {
            $this->velocidad = 0;
      }
      Function aumentarVelocidad() {
            $this->velocidad++;
      }

      function parar() {
            $this->velocidad = 0;
      }
}
$coche = new Class_coche();
$coche->velocidad = 1000; // velocidad = 1000
```

4.7.2 Atributos Privados. *Private*

Como norma de la filosofía de la programación orientada a objeto las propiedades o los atributos deben permanecer ocultos al exterior, por ello se suele usar el modificador *private*.

Sólo los métodos de la clase pueden acceder a dichos atributos y no está permitido acceder a ellos desde el exterior, aunque es común permitir el acceso dichos de forma controlada a través de "setters" y "getters" como veremos a continuación.Las clases hijas tampoco tienen permitido el acceso a los atributos *private*.

Veamos ejemplo con *private*

```
<?php
class Class_coche {
      private $velocidad;

      public fucntion __construct() {
            $this->velocidad = 0;
      }

      Public Function setVelocidad($velocidad) {

            $this->velocidad = (int) $velocidad;
      }

      Public Function getVelocidad() {
            return $this->velocidad;
      }

}
$coche = new Class_coche();
$coche->setVelocidad(1000);
echo$coche->getVelocidad(); // 1000
echo $coche->velocidad; //Error: el atributo velocidad es privado
```

4.7.3 Atributos protegidos. *Protected*

Hace que al atributo se puede acceder desde la clase que las define y también desde cualquier otra clase hija.

Podemos crear a partir de una clase una subclase o clase hija, esta nueva clase hereda todos los atributos y métodos de la clase madre declarados con el modificador *public* y *protected*.

Veamos el siguiente ejemplo:

```
<?php
class Class_coche {
      private $velocidad;

      public fucntion __construct() {
            $this->velocidad = 0;
      }

      Public Function setVelocidad($velocidad) {
            $this->velocidad = (int) $velocidad;
      }
}

Class Class_deportivo extends Class_coche{

      Public Function aumentarVelocidad() {
            $this->velocidad  = $this->velocidad * 2;
      }
```

```
}

$deportivo = new Class_deportivo();
$deportivo->setVelocidad(100);
$deportivo->aumentarVelocidad(); // PHP Notice:
      Undefinedproperty//Deportivo::$velocidad
```

Los métodos que heredan de Coche, pueden acceder a la propiedad declarada como private, sin embargo cualquier método nuevo no puede. Para ello existe un nuevo tipo de ámbito, protected, que permite el acceso completo desde la propia clases y todas las clases heredadas.

```
<?php
class Class_coche {
      protected $velocidad = 0;

      public function setVelocidad($velocidad) {
            $this->velocidad = (int) $velocidad;
   }
}

Class Class_deportivo extends Class_coche{

      Function aumentarVelocidad() {
            $this->velocidad  = $this->velocidad * 2;
      }
}

$deportivo = new Class_deportivo();
$deportivo->setVelocidad(100);
$deportivo->aumentarVelocidad(); // velocidad = 200
```

Una buena práctica es declarar siempre los atributos como *private*, en caso de que la clase sea susceptible de ser clase padre o superclase y quieras delegar el control del atributo a las clases hijas debes usar *pretected*. Por otro lado los métodos tanto de las clases padres como hijas los debes declarar como públicos.

4.7.4 Elementos Estáticos

Existe un tipo especial de métodos y atributos denominados estáticos, que se caracterizan porque no se necesita crear una instancia, un objeto, para usarse, sino que pueden emplearse directamente con el operador PaamayimNekudotayim (los cuatro puntos ::). Dicho de otra forma, sirven para definir métodos y propiedades cuyo comportamiento no depende de la creación de un objeto, de ahí que se conozcan como «elementos de clase».

Para crear un elemento estático en una clase se usa la palabra reservada *static*

```
Class nombClase{
      Static [atributos]
      Static [métodos]
    };
```

Los métodos estáticos no pueden acceder a ningún atributo normal de la clase. Debido a que los métodos estáticos se pueden invocar sin tener creada una instancia del objeto, la variable

$this no está disponible dentro de los métodos declarados como estáticos. Pero si que pueden acceder a atributos definidos como estáticos.

Para acceder a atributos o métodos estáticos, desde el interior de otro método estático, no se necesita una variable que haga referencia a un objeto ($this). En vez de ello se usa la palabra reservada *self* en conjunción de los caracteres "::" y el nombre del atributo.

Veamos el siguiente ejemplo

```
class Class_prueba {
      static public $nombre = "José";
      static public function saludo() {
            echo "Hola " . self::$nombre . " os saluda";
      }
}
```

Un atributo o método declarado como *static* no puede ser accedido desde un objeto de clase instanciado como lo veníamos haciendo hasta ahora, usamos el nombre de la clase seguido de :: y a continuación el nombre del método o el atributo.Veamos el siguiente ejemplo:

```
class Class_prueba {
      public static $nombre = "José";
      public static function saludo() {
            echo "Hola " . self::$nombre . " os saluda";
      }
}

 $p = new Class_prueba();
 $p2 = new Class_prueba();
 $p->saludo();  //error
 $p2->saludo();  //error
 $p->$nombre; //error
 Echo Class_prueba::nombre; //Correcto
 Echo Class_prueba::saludo(); //Correcto
```