



UD4 – POO con PHP

2º CFGS
Desarrollo de Aplicaciones Web
2022-23

1.- Introducción a POO

El concepto de **Programación Orientada a Objetos** apareció con el lenguaje de programación **Simula** en **1962**, lanzado oficialmente en 1967.

Su auge fue gracias a **Java** (1995) y **C#** (2000) que introdujeron estos conceptos y metodologías.

En la POO todos los elementos que actúan en un programa se tratan como elementos de la vida real con sus características y comportamientos.

Estos elementos son **objetos** que se ajustan a unas definiciones llamadas **clases**.

1.- Introducción a POO

Características de la POO:

- **Herencia:** cuando se crea una clase a partir de otra se hereda su comportamiento y características.
- **Abstracción:** de cara al exterior la clase sólo muestra los métodos no el cómo hace las cosas.
- **Polimorfismo y sobrecarga:** los métodos pueden tener comportamientos diferentes según la forma de utilizarlos.
- **Encapsulación:** Están juntos los datos y el código que los usa.

1.- Introducción a POO

Clases:

- **Propiedades** (atributos): almacenan información acerca del estado del objeto al que pertenecen. Su valor puede ser diferente para diferentes objetos de una misma clase.
- **Métodos**: contienen código ejecutable y definen las acciones que realiza el objeto. Son como una función y pueden recibir parámetros y devolver valores.
- **Instancia**: cuando se tiene una clase definida y se crea un objeto de dicha clase se dice que se tiene una instancia de la clase (un elemento "real").

1.- Introducción a POO

Ventajas que aporta la POO:

- **Modularidad:** permite dividir los programas en partes más pequeñas independientes entre sí. Se pueden usar esas partes en otros programas.
- **Extensibilidad:** para ampliar la funcionalidad de las clases solo hay que modificar su código mediante herencia.
- **Mantenimiento:** Gracias a la modularidad el mantenimiento es más sencillo. Es importante que cada clase esté en un archivo diferente.

[Más información acerca de la POO.](#)

2.- POO en PHP

En un principio PHP no se diseñó para la **POO**.

En PHP 3 se introdujeron algunos conceptos de POO.

En PHP 4 se potenció su uso debido al auge de la POO gracias a Java y C#.

En PHP 5 se reescribió el motor de PHP que incluía un nuevo modelo de objetos.

Actualmente PHP soporta todas las características de la POO **excepto la herencia múltiple y la sobrecarga de métodos**.

3.- Clases en PHP

En PHP no es necesario que cada clase esté en su propio archivo .php.

Aún así, es recomendable que cada clase que se declare esté en su archivo propio cuyo nombre seguirá el patrón: **NombreClase.inc.php**.

Para la declaración de una clase se usa la palabra reservada **class**.

Los nombres de las clases deben comenzar por **mayúscula**.

Como buena práctica como programador, el código dentro de la clase debe estar bien organizado, por ello primero se indicarán las propiedades, luego los constructores y finalmente los métodos propios.

3.- Clases en PHP

Ejemplo de una clase Producto → **Producto.inc.php**:

```
<?php
class Producto {
    public $codigo;
    public $nombre;
    public $precio;

    public function mostrarDatos() {
        return $nombre . '(' . $codigo . ')':'. $precio . ' €';
    }
}
```


3.- Clases en PHP

En las clases conviene tener un método que permita inicializar a los objetos, este método es el **constructor**.

Somo PHP no permite sobrecarga solo se puede definir un constructor.

Para definir un constructor se usa **__constructor** como se puede observar delante del nombre se usan dos caracteres **_**.

3.- Clases en PHP

Producto.inc.php

```
<?php
class Producto {
    private $codigo;
    private $nombre;
    private $precio;

    public function __construct($codigo, $nombre, $precio) {
        $this->codigo = $codigo;
        $this->nombre = $nombre;
        $this->precio = $precio;
    }
}
```

3.- Clases en PHP

Si se necesita usar constructores con diferente número de parámetros se puede declarar el constructor con parámetros con valor por defecto.

```
public function __construct($codigo="", $nombre="", $precio=0.0) {  
    $this->codigo = $codigo;  
    $this->nombre = $nombre;  
    $this->precio = $precio;  
}
```

```
$consola = new Producto("sony-ps5", "PS5", 499.99);  
$tele = new Producto("Samsung65"); // Se asigna solo el código  
$movil = new Producto(nombre:"apple-i14", precio: 1009.0);  
$tablet = new Producto(nombre:"Galaxy Tab");
```

3.- Clases en PHP

Para instanciar objetos de una clase se debe incluir el script php que la contiene:

```
require_once("Producto.inc.php");
```

Para **instanciar** un objeto se usa la palabra **new**:

```
$consola = new Producto("sony-ps5", "ps5", 499.99);
```

Se usa la notación -> para acceder a las propiedades y los métodos de los objetos:

```
$consola->nombre = "PS5";  
$consola->precio = 459.0;  
echo $consola->mostrarDatos();
```

4.- Propiedades

Se puede definir el tipo de acceso para las propiedades y los métodos de una clase.

- **public:** se puede acceder a ellos de manera directa.
- **protected:** se puede acceder desde la propia clase y las que hereden de ella.
- **private:** solo se puede acceder a ellos desde dentro de la clase.

```
class Producto {  
    private $codigo;  
    private $nombre;  
    private $precio;  
}
```

```
// Error porque la propiedad es privada  
echo $consola->precio;
```

4.- Propiedades

Lo más habitual es que las propiedades se declaren **private** y en ese caso será necesario crear los métodos **get** y **set** para poder acceder a ellas.

```
class Producto {  
    private $codigo;  
    public $nombre;  
    public $precio;  
  
    public function getCodigo() {  
        return $this->codigo;  
    }  
    public function setCodigo($codigo) {  
        $this->codigo = $codigo;  
    }  
  
    public function mostrarDatos() {  
        return $this->nombre .'(' . $this->código  
            . '): ' . $this->precio . ' €';  
    }  
}
```

```
echo $consola->getCodigo();
```

5.- Métodos mágicos

En el modelo de objetos de PHP permite crear los denominados **métodos mágicos** que facilitan la creación de métodos que realizan tareas concretas.

Los métodos mágicos comienzan con dos caracteres `_` seguidos.

`__construct`

`__callStatic`

`__isset`

`__wakeup`

`__toString`

`__clone`

`__destruct`

`__get`

`__unset`

`__serialize`

`__invoke`

`__debugInfo`

`__call`

`__set`

`__sleep`

`__unserialize`

`__set_state`

5.- Métodos mágicos

La mejor forma de comprender los métodos mágicos es viendo un ejemplo de `__get` y `__set`:

```
class Producto {  
    private $nombre;  
    private $precio;  
  
    public function __set ($propiedad, $valor) {  
        $this->$propiedad = $valor;  
    }  
    public function __get ($propiedad) {  
        return $this->$propiedad;  
    }  
}
```


```
$consola = new Producto();  
$consola->nombre = "PS5";           // acceso al método mágico __set(nombre, "PS5")  
$consola->precio = 499.99;          // acceso al método mágico __set(precio, 499.99)  
echo $consola->nombre;               // acceso al método mágico __get(nombre)
```


5.- Métodos mágicos

Aunque parezca una buena práctica el uso de los métodos mágicos no siempre es recomendable su uso.

- Los métodos mágicos suelen ser mucho más lentos que los normales.
- Con `__get` y `__set` existe un problema de seguridad y no se recomienda su uso:

```
class Producto {  
    private $codigo;  
    private $nombre;  
    private $precio;  
  
    public function __set ($propiedad, $valor) {  
        $this->$propiedad = $valor;  
    }  
    public function __get ($propiedad) {  
        return $this->$propiedad;  
    }  
}
```



```
// Se crea una nueva propiedad  
$consola->nueva = "codigo malicioso";  
echo $consola->nueva;
```

6.- Propiedades y métodos estáticos

Las propiedades y los métodos estáticos, también llamados **de clase**, se pueden utilizar sin instanciar ningún objeto.

Para definirlos se usa la palabra **static**.

Se utiliza la notación **::** para acceder a las propiedades o usar los métodos.

Si son **public** se podrá acceder a ellos usando el nombre de la clase.

Si son **private** se podrá acceder desde dentro de la clase (declaración) usando la palabra **self**.

6.- Propiedades y métodos estáticos

```
class Producto {  
    private static $cantidadProductos = 0;  
    private $codigo;  
    public $nombre;  
    public $precio;  
  
    public static function nuevoProducto () {  
        self::$cantidadProductos++;  
    }  
}
```

```
Producto::nuevoProducto();
```

7.- Uso de objetos

Como se ha visto hasta ahora, el uso de clases y objetos en PHP es muy similar a su uso en Java.

Si se quiere comprobar si un objeto es de una clase determinada se puede usar la instrucción **instanceof** igual que en Java.

PHP también hay una serie de funciones útiles para el uso de objetos:

get_class class_exists get_class_methods
...

En la documentación se pueden ver todas [todas](#).

7.- Uso de objetos

Desde PHP 5 en la definición de las funciones se puede indicar el tipo de dato de los parámetros y el tipo de dato de retorno.

Esto es extrapolable a la definición de las clases y en sus constructores y sus métodos.

Se aconseja que a la hora de definir clases se utilicen estas técnicas ya que así los scripts son más estrictos y se evitan posibles errores.

7.- Uso de objetos

Las variables que guardan objetos realmente almacenan la dirección de memoria donde se guardan los datos del objeto.

Por esa razón cuando se realizan las siguientes instrucciones no se está realizando una copia del objeto si no que las dos variables son lo mismo.

```
$consola = new Producto("sony-ps5", "PS5", 499.99);  
$copiaConsola = $consola;  
$copiaConsola->nombre = "PS4";  
echo $consola->nombre .' - ' . $copiaConsola->nombre;  
// Se mostrará: PS4 - PS4
```

Para realizar una copia de un objeto es necesario utilizar la función **clone**.

```
$copiaConsola = clone($consola);
```

7.- Uso de objetos

Para comparar objetos se usan el operador `==` para saber si sus propiedades tienen los mismos valores y `===` para saber si son el mismo objeto.

```
$consola = new Producto("sony-ps5", "PS5", 499.99);  
$copiaConsola = clone($consola);
```

El resultado de `$consola == $copiaConsola` es **true** porque son dos copias idénticas.

El resultado de `$consola === $copiaConsola` es **false** porque son dos objetos diferentes.

Práctica

Actividad 1: Agenda de contactos.

8.- Herencia

La herencia permite definir clases en base a otras ya existentes creando así una jerarquía de clases.

Al definir clases nuevas en base a otras se puede ampliar su funcionalidad.

A las nuevas clases se les llama **subclases** y a las clases que sirven e base se les llama **superclases**.

Al utilizar la herencia, además de los modificadores de visibilidad **public** y **private**, se puede hacer uso del modificador **protected**.

8.- Herencia

Si todos los productos tuvieran solo las propiedades código, nombre y precio la siguiente clase sería suficiente.

Al incorporar productos como televisores, altavoces, móviles... estas propiedades son comunes pero cada tipo tiene además otras propiedades interesantes.

```
class Producto {  
    protected $codigo;  
    protected $nombre;  
    protected $precio;  
  
    public function __construct($codigo, $nombre, $precio) {  
        $this->codigo = $codigo;  
        $this->nombre = $nombre;  
        $this->precio = $precio;  
    }  
  
    public function __toString() {  
        return $this->nombre . ' (' . $this->codigo . '): ' .  
            $this->precio . ' €';  
    }  
}
```

8.- Herencia

Mediante la palabra reservada **extends** se definen subclases.

Si se quiere llamar a un método de la superclase se debe usar la palabra reservada **parent** seguida del operador **::**

```
class Television extends Producto {
    public $pulgadas;
    public $tecnologia;

    public function __construct($codigo, $nombre, $precio, $pulgadas, $tecnologia) {
        parent::__construct($codigo, $nombre, $precio);
        $this->pulgadas = $pulgadas;
        $this->tecnologia = $tecnologia;
    }

    public function __toString() {
        return parent::__toString() . ' '.
            $this->pulgadas . ' '.
            $this->tecnologia;
    }
}
```

```
class Altavoz extends Producto {
    public $potencia;
    public $canales;

    public function __construct($codigo, $nombre, $precio, $potencia, $canales) {
        parent::__construct($codigo, $nombre, $precio);
        $this->potencia = $potencia;
        $this->canales = $canales;
    }

    public function __toString() {
        return parent::__toString() . ' '.
            $this->potencia . ' '.
            $this->canales . ' canales';
    }
}
```

8.- Herencia

Como se ha indicado anteriormente en la documentación se pueden consultar las funciones para trabajar con objetos.

Entre esas funciones se encuentran las siguientes dos que facilitan el trabajo cuando se define herencia:

`get_parent_class`

`is_subclass_of`

9.- Sobrecarga

La **sobrecarga** en POO permite definir **métodos con el mismo nombre** pero con diferente cantidad de parámetros y tipos de los mismos.

Al realizar la llamada a un método sobrecargado dependiendo de los parámetros se ejecutara una "versión" u otra del método.

Aunque en PHP no se permite la sobrecarga, se puede hacer uso de una de las características de PHP para simular la sobrecarga.

Como se estudió en su momento, **se puede llamar a una función con un número variable de parámetros**.

9.- Sobrecarga

La **sobrecarga** en POO permite definir **métodos con el mismo nombre** pero con diferente cantidad de parámetros y tipos de los mismos.

Al realizar la llamada a un método sobrecargado dependiendo de los parámetros se ejecutara una "versión" u otra del método.

Aunque en PHP no se permite la sobrecarga, se puede hacer uso de una de las características de PHP para simular la sobrecarga.

9.- Sobrecarga

Para simular la sobrecarga se pueden usar las siguientes técnicas:

- Argumentos con nombre

```
public function precioFinal($precio, $iva=21, $descuento=0.0) {  
    // acciones a realizar  
}
```

- Cantidad de argumentos variables

```
public function precioFinal() {  
    if (func_num_args() == 1) {  
        // acciones a realizar  
    } else if (func_num_args() == 2) {  
        // acciones a realizar  
    } else if (func_num_args() == 3) {  
        // acciones a realizar  
    }  
}
```

Práctica

Actividad 2: codificando un UML.

10.- Clases y métodos abstractos

Las clases **abstractas** son aquellas que no permiten instanciar objetos de ellas, así que su función es la de servir de superclase.

Los métodos de una clase también se pueden definir como abstractos, en ese caso el método no contendrá código y será obligado declararlos y definirlos en las subclases.

```
abstract class Figura {  
    protected $color;  
  
    public function __construct($color) {  
        $this->color = $color;  
    }  
  
    public function getColor() {  
        return $this->color;  
    }  
  
    public function setColor($color) {  
        $this->color = $color;  
    }  
  
    abstract public function dibuja();  
    abstract public function calcularArea();  
}
```

```
class Cuadrado extends Figura {  
    private $lado;  
  
    public function __construct($color, $lado) {  
        parent::__construct($color);  
        $this->lado = $lado;  
    }  
  
    public function dibuja() {  
        // instrucciones  
    }  
  
    public function calcularArea() {  
        return pow($this->lado, 2);  
    }  
}
```

10.- Clases y métodos abstractos

Si no se quiere permitir la herencia se debe usar la palabra **final** que sirve tanto para clases como para métodos.

```
final class Vehiculo {  
    ...  
}
```

```
public final function asignarIva($iva=21) {  
    ...  
}
```

10.- Interfaces

Las **interfaces** son clases que solo contienen declaraciones de métodos sin definir su código.

Las clases definidas como interfaces se utilizan como plantillas para otras clases, de esta manera estas clases deberán definir todos los métodos declarados en la interfaz.

```
interface MostrarDatos {  
    public function mostrar();  
}  
  
class Consola extends Producto implements MostrarDatos {  
    public function mostrar ( ) {  
        // Instrucciones  
    }  
}
```

11.- Traits

Una de las características típicas de la herencia es poder reutilizar código ya codificado en la superclase.

En PHP no se permite la herencia múltiple (pocos lenguajes lo permiten).

PHP dispone de los **trait** que permiten reutilizar código reduciendo las limitaciones de la herencia simple.

Los **traits** son similares a las clases pero no se permite instanciar objetos y solo permiten agrupar funcionalidades.

11.- Traits

```
trait Saludos {  
    public function holaMundo() {  
        echo 'Hola mundo!';  
    }  
  
    public function saludoObiWan() {  
        echo 'Hello there!';  
    }  
}  
  
trait Despedida {  
    public function despedida() {  
        echo "Hasta la vista baby!";  
    }  
}
```

```
class Persona {  
    use Saludos;  
    use Despedida;  
}  
  
$obiWan = new Persona();  
echo $obiWan->saludoObiWan();  
echo '<br>';  
echo $obiWan->despedida();
```

12.- Excepciones

PHP dispone de un modelo de excepciones similar al de otros lenguajes de programación:

try – catch – finally

Un bloque **try** debe tener un bloque **catch** y/o un bloque **finally**.

Un bloque **try** puede tener varios bloques **catch**.

Se pueden lanzar excepciones con **throw new TIPO_EXCEPCION**. ([Tipos](#))

También se pueden crear subclases de las excepciones para crear excepciones propias.

12.- Excepciones

```
$num1 = 13;  
$num2 = 0;  
  
try {  
    $res = $num1 .'/' . $num2 .'=' . $num1/$num2;  
} catch (DivisionByZeroError $e) {  
    echo 'Se ha producido un error: ' . $e->getMessage();  
}
```

Práctica

Actividad 3:
El zoológico.

Actividad 4:
Pilotos MotoGP.