UNIDAD DIDÁCTICA 5: POO CON PHP

Introducción

Históricamente, PHP ha sido un lenguaje procedural. Esto causaba problemas de mantenimiento y escalabilidad.

A partir de la **versión 5**, PHP incorporó un modelo completo de Programación Orientada a Objetos (POO), similar al de otros lenguajes modernos.

- La POO es esencial para el mantenimiento, la escalabilidad y la legibilidad del código.
- Es la base de frameworks modernos como Symfony y Laravel.
- En esta unidad, aprenderemos a definir y utilizar clases y objetos en PHP.

Caso Introductorio

> Queremos definir clases que nos permitan datos de jugadores de un equipo de baloncesto y además que sea suficientemente mantenible y flexible para poder ampliarlo en un futuro.

1. Programación Orientada a Objetos

¿Qué es un Objeto?

1.1. ¿Qué es un Objeto?

Un objeto es la representación en software de un concepto del mundo real (un coche, un jugador, una factura).

Un objeto agrupa su información y su comportamiento en una sola entidad. Se compone de dos partes:

Propiedades (o Atributos)

Son las **características** del objeto. Son variables que pertenecen al objeto.

Ejemplo (para un Coche):

`color` = "Rojo"

`tipo` = "Turismo"

`uso` = "Particular"

Métodos (o Funciones)

Son las **acciones** que el objeto puede realizar. Son funciones que pertenecen al objeto.

Ejemplo (para un Coche):

`circularPorCiudad()`

`irMarchaAtrás()`

`cobrarPorViaje()`

1.3. Definición de una Clase

Clases: La "Plantilla"

Una **Clase** es la plantilla o molde para crear objetos.

Se define con la palabra reservada `class`, seguida del nombre de la clase (por convención, en *UpperCamelCase*).

Dentro de las llaves `{ }` definimos sus propiedades y métodos.

Ejemplo: `ClaseCoche.php`

```
color; // Usamos $this
    }

    public function getColor() {
        return $this->color;
    }
}
?>
```

1.4. `include` vs `require`

Reutilización de Código

Para mantener el código organizado, guardamos cada clase en su propio archivo (ej. `ClaseCoche.php`).

Luego, usamos `include` o `require` para "traer" ese archivo al script principal.

- `include 'archivo.php';`
  Si el archivo no existe, lanza un **Warning** (aviso) y el script **continúa**.
- `require 'archivo.php';`
  Si el archivo no existe, lanza un **Fatal Error** y el script **se detiene**.

Regla general: Usa `require` para clases, ya que sin ellas la aplicación no puede funcionar.

`include_once` y `require_once`

Son las versiones más seguras. PHP recuerda si ya ha incluido ese archivo y no lo volverá a incluir, evitando errores por "redeclaración de clase".

Siempre usaremos `require_once` para incluir clases.

```
// Archivo: index.php
require_once 'ClaseCoche.php';
require_once 'ClaseMoto.php';

// ... tu código ...
```

1.5. Creación de Objetos (Instanciación)

Crear un Objeto

Una vez definida la "plantilla" (Clase), podemos crear "instancias" (Objetos) usando la palabra reservada `new`.

Cada objeto es una copia independiente con sus propias propiedades.

```
// Incluimos la definición de la clase
require_once 'ClaseCoche.php';

// Creamos dos objetos (instancias)
$coche1 = new ClaseCoche();
$coche2 = new ClaseCoche();
```

Usar los Objetos

Usamos el operador "flecha" (->) para acceder a las propiedades y métodos de un objeto.

```
// Modificamos solo las propiedades del coche 1
$coche1->color = 'Rojo';

// Mostramos los colores de ambos
$coche1->mostrarColor(); // Imprime "Rojo"
$coche2->mostrarColor(); // Imprime "Verde"
```

2. Características de POO en PHP

Propiedades, Métodos, Visibilidad y Constructores

2.1. Propiedades (Atributos)

Las variables pertenecientes a una clase se llaman "propiedades".

Se declaran al principio de la clase usando una palabra clave de visibilidad (`public`, `private`, `protected`) seguida del nombre de la variable (con `$`).

```
class Jugador {
    // Declaración de propiedades
    public $nombre = "Sin nombre";
    public $posicion;
    public $dorsal = 0;

    // Válido a partir de PHP 7.4 (tipado)
    // public string $nombre;
    // public int $dorsal;
}
```

Definir un valor (ej. `= "Sin nombre"`) es opcional y sirve como valor por defecto.

2.2. Métodos (Funciones)

Los métodos son funciones definidas dentro de una clase.

Se declaran con la palabra clave `function` y una visibilidad (generalmente `public`).

```
class Jugador {
    public $nombre = "Paco";

    // Un método simple
    public function saludar() {
        echo "¡Hola!";
    }

    // Un método que devuelve un valor
    public function getNombre() {
        return $this->nombre; // $this es esencial
    }

    // Un método con argumentos
    public function setNombre($nuevoNombre) {
        $this->nombre = $nuevoNombre;
    }
```

2.4. La Pseudovariable `$this`

¿Qué es `$this`?

La pseudovariable `$this` está disponible **dentro** de un método y se refiere al **objeto actual** (a la instancia que está llamando al método).

Se usa para acceder a las propiedades y métodos de ese mismo objeto.

Error común: Olvidar `$this->`.

Ejemplo de uso de `$this`

```
class ClaseSencilla {
    public $var = 3;

    public function mostrarVar() {
        // Correcto: accede a la propiedad
        echo $this->var; // Imprime 3
    }

    public function mostrarVar2() {
        // Incorrecto: busca una variable local
        echo $var; // No imprime nada (Warning)
    }
}
```

2.5. Ámbito (Scope) de Variables

El ámbito de una variable es el contexto donde está definida.

Ámbito Global

Una variable declarada fuera de cualquier función o clase. No es accesible directamente *dentro* de una función.

```
$area = 0; // Ámbito global
function calcularArea() {
    $area = 10; // Ámbito local (¡es OTRA variable!)
    echo "Área dentro: " . $area; // 10
}
calcularArea();
echo "Área fuera: " . $area; // 0
```

Nota: Esta programación (procedural) se debe evitar. En POO, las variables se encapsulan en las clases.

2.6. Visibilidad (Encapsulación)

La visibilidad define desde dónde se puede acceder a una propiedad o método. Es la base de la **encapsulación**.

🌐 `public` **(Público)**

Se puede acceder desde cualquier lugar: dentro de la clase, en clases heredadas y fuera de la clase (desde el objeto).

🔒 `private` **(Privado)**

Solo se puede acceder desde **dentro de la misma clase** que lo definió. Ni clases heredadas ni objetos externos pueden verlo.

🔑 `protected` **(Protegido)**

Se puede acceder desde la misma clase y desde **clases heredadas**, pero no desde fuera.

Visibilidad: `public` vs `private`

`public` (Acceso Directo)

Permite leer y escribir la propiedad directamente desde fuera.

```
class Coche {
    public $color = 'Verde';
}

$miCoche = new Coche();
$miCoche->color = 'Azul'; // FUNCIONA
echo $miCoche->color;     // FUNCIONA
```

Desventaja: No tenemos control. Cualquiera puede asignar un valor inválido (ej. `$miCoche->color = 12345;`).

`private` (Acceso Bloqueado)

Bloquea el acceso desde fuera de la clase. Intentarlo provoca un **Error Fatal**.

```
class Coche {
    private $color = 'Verde';
}

$miCoche = new Coche();
$miCoche->color = 'Azul'; // ¡ERROR FATAL!
echo $miCoche->color;     // ¡ERROR FATAL!
```

Pregunta: ¿Cómo accedemos entonces?

2.7. Getters y 2.8. Setters

Métodos de Acceso

Para controlar el acceso a propiedades `private`, usamos métodos `public`:

- Un **Getter** es un método público que **devuelve** (get) el valor de una propiedad privada.
- Un **Setter** es un método público que **establece** (set) el valor de una propiedad privada, usualmente recibiendo el nuevo valor como parámetro.

Ejemplo de Getter y Setter

```
class Coche {
    private $color = 'Verde';

    // Getter para $color
    public function getColor() {
        return $this->color;
    }

    // Setter para $color
    public function setColor($nuevoColor) {
        $this->color = $nuevoColor;
    }
}

$miCoche = new Coche();
$miCoche->setColor('Azul'); // FUNCIONA
echo $miCoche->getColor(); // Imprime "Azul"
```

2.9. Visibilidad y Setters (Validación)

El Poder del Setter: Validación

¿Por qué usar un setter si parece lo mismo que `public`? Porque el setter nos permite **validar** los datos antes de asignarlos.

Si la propiedad fuera pública, no podríamos controlar los valores que se le asignan.

Ejemplo de Validación

```
class Jugador {
    private $nombre;
    private $edad;

    // El setter valida la edad
    public function setEdad($nuevaEdad) {
        if ($nuevaEdad > 0 && $nuevaEdad < 100) {
            $this->edad = $nuevaEdad;
        } else {
            // No hacemos nada o lanzamos un error
            echo "Error: Edad no válida.";
        }
    }
}
$j = new Jugador();
$j->setEdad(25);  // Funciona
$j->setEdad(150); // Imprime "Error: Edad no válida."
```

2.10. Visibilidad y Métodos Privados

Métodos Privados (Helpers)

Un método `private` (o `protected`) no puede ser llamado desde fuera del objeto.

Se usan como "funciones ayudantes" (helpers) internas para organizar el código y evitar duplicación, siendo llamadas por otros métodos públicos.

Ejemplo de Helper

```
class Factura {
    // Método público (API)
    public function calcularTotal($subtotal) {
        $impuestos = $this->calcularIVA($subtotal);
        $total = $subtotal + $impuestos;
        return $total;
    }

    // Método privado (Helper)
    private function calcularIVA($cantidad) {
        // Lógica compleja de impuestos
        return $cantidad * 0.21;
    }
}

$f = new Factura();
echo $f->calcularTotal(100); // Imprime 121
// $f->calcularIVA(100); // ¡ERROR FATAL!
```

2.11. Diagrama de Clases y Visibilidad (UML)

En los diagramas de clase UML (Lenguaje Unificado de Modelado), la visibilidad se representa con símbolos:

+ (Signo Más) = **public**

- (Signo Menos) = **private**

(Almohadilla) = **protected**

Ejemplo de Clase (Representación textual)

```
Clase: Jugador
---------------------------------
- nombre: String
- posicion: String
---------------------------------
+ setPosicion(nueva: String): void
+ getNombre(): String
```

2.12. El Constructor

Método Mágico: `__construct()`

PHP tiene "métodos mágicos" que empiezan con doble guion bajo (__).

El **constructor** (`__construct`) es un método mágico que se llama **automáticamente** cada vez que se crea un objeto con `new`.

Es el lugar ideal para inicializar las propiedades obligatorias del objeto.

Ejemplo de Constructor

```
class Jugador {
    private $nombre;

    // El constructor se ejecuta con "new"
    public function __construct($nombreInicial) {
        echo "Creando jugador... ";
        $this->nombre = $nombreInicial;
    }

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

// Pasamos "Rudy" al constructor
$j1 = new Jugador("Rudy");
// Imprime "Creando jugador... "

echo $j1->getNombre(); // Imprime "Rudy"
```

Solución al Caso: Clase `Equipo`

Aplicando todo lo aprendido (clase, propiedades, visibilidad, getter/setter y constructor) al caso introductorio del equipo de baloncesto:

Archivo: `Equipo.php`

```
nombre = $nombre;
        $this->posicionLiga = 0; // Valor por defecto
    }

    // Setter para la posición
    public function setPosicion($pos) {
        if ($pos > 0) {
            $this->posicionLiga = $pos;
        }
    }

    // Getter para el nombre
    public function getNombre() {
        return $this->nombre;
    }
```

Solución al Caso: Archivo `liga.php`

Creamos los objetos (equipos) y los utilizamos:

Archivo: `liga.php`

```
setPosicion(1);
$RMDBasket->setPosicion(3);

// 4. Usamos los getters para mostrar la info
echo "Equipo: " . $VLCBasket->getNombre();
echo "";
echo "Equipo: " . $RMDBasket->getNombre();

?>
```

Resumen Final

> Hemos introducido la Programación Orientada a Objetos en PHP, un paradigma esencial para el desarrollo moderno. A partir de la versión 5 (y mejorado en la 7+), PHP permite crear código flexible y mantenible. Hemos aprendido a definir Clases (plantillas) y a crear Objetos (instancias), encapsulando propiedades (`private`) y exponiendo métodos (`public`) como getters, setters y constructores (`__construct`).

¿Preguntas?