Desarrollo web en entorno servidor

Unidad 7: Programación orientada a objetos en los lenguajes script de servidor

ÍNDICE

INTRODUCCIÓN

La **Programación Orientada a Objetos (OOP, Object Oriented Programming)** es un estilo de organizar el código que permite a los desarrolladores agrupar tareas similares en clases. Esto ayuda a que el código sea más fácil de mantener y a no repetirse (**DRY, Don't Repeat Yourself**).

La POO (o OOP en inglés) es algo que ya hemos trabajado. Recuerda la técnica PDO en la unidad anterior de acceso a datos. Ahora veremos de dónde sale y entraremos más detalladamente en sus pilares fundamentales.

Aunque a priori parezca más complejo abordar problemas desde la visión de la orientación a objetos con respecto a la programación procedimental, cuando domines la técnica verás que es mucho más natural, ya que este paradigma de programación busca **resolver problemas complejos a partir de modelos cotidianos**.

OBJETIVOS / CAPACIDADES

En esta unidad de aprendizaje, las capacidades que más se van a trabajar son:

- ✔ **Utilizar herramientas y lenguajes específicos, cumpliendo las especificaciones, para desarrollar e integrar componentes software en el entorno del servidor web.**
- ✔ **Programar y realizar actividades para gestionar el mantenimiento de los recursos informáticos.**

PROYECTO DE LA UNIDAD

Antes de empezar a trabajar el contenido, te presentamos la **actividad** que está relacionada con esta unidad de aprendizaje. Se trata de un **caso práctico** basado en una **situación real** con la que te puedes encontrar en tu puesto de trabajo. Con esta actividad se evaluará la puesta en práctica de los **criterios de evaluación** vinculados al resultado de aprendizaje que se trabaja en esta unidad. Para realizarla deberás hacer lo siguiente: lee el enunciado que te presentamos a continuación, dirígete al área general del módulo profesional, concretamente a la actividad de evaluación que se encuentra dentro de esta unidad, allí encontrarás todos los detalles sobre fecha y forma de entrega, objetivos... A lo largo de la unidad irás adquiriendo los conocimientos necesarios para ir elaborando este proyecto.

Enunciado:

Almacenando datos de empleados.

Seguiremos la actividad de evaluación 6 justo donde la dejamos, es decir, tenemos un escenario donde podemos crear, listar, modificar y eliminar empleados de una base de datos de una forma muy visual.

Ahora deberemos hacer los cambios necesarios en la estructura de datos de nuestra aplicación para que los empleados que crees sean objetos de la clase Empleado siguiendo las siguientes directrices:

- Existirá la clase padre llamada Empleado que será abstracta. Esta clase tendrá los atributos nombre, apellidos, edad y sueldo bruto anual. Fíjate, que a diferencia de la UD6, desaparece el atributo cargo.
- Existirán 3 clases que heredan de Empleado: recursos humanos, marketing y atención al cliente. Las 3 clases tendrán un nuevo atributo llamado sueldo base.

- Crea un método (dónde y como quieras) que se llame aplicarBono() que incrementará un 10% el sueldo base.
- Crea los data accessors necesarios para recuperar los datos de los empleados y que se sigan pudiendo guardar, modificar y eliminar de la base de datos.

Para realizar el juego de pruebas crea un mínimo de 3 empleados con todos sus datos. Uno de ellos debe ser del departamento de RRHH, otro de marketing y otro de atención al cliente.

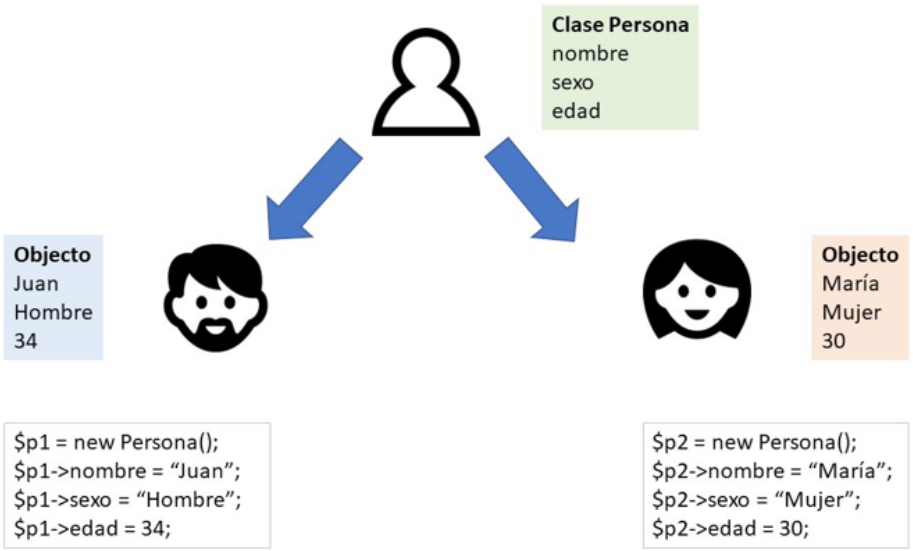
1. LAS CLASES. USO DE CLASES

DESTACADO *Una **clase** es una plantilla para objetos y un **objeto** es una instancia de clase.*

Vamos a ver un **ejemplo** para entender mejor este apartado.

- **I**. Supongamos que tenemos una **clase llamada Persona**. Una persona puede tener propiedades como nombre, sexo, edad, etc. Podemos definir variables como $nombre, $sexo y $edad para contener los valores de estas propiedades.

  Cuando se crean los objetos individuales (Juan, María, etc.), **heredan todas las propiedades y comportamientos** de la clase, pero cada objeto tendrá diferentes valores para las propiedades.



- **II**. Una clase se define usando la palabra reservada `class`, seguida del nombre de la clase y un par de llaves (`{}`). Todas sus propiedades y métodos van dentro de las llaves.

➔ **III**. A continuación, declaramos una clase llamada Persona que consta de tres propiedades ($nombre, $sexo y $edad) y dos métodos `set_name()` y `get_name()` para establecer y obtener la propiedad $nombre:

```
<?php
class Persona {
  // Propiedades
  public $nombre;
  public $sexo;
  public $edad;

  // Métodos
  function set_name($nombre) {
    $this->nombre = $nombre;
  }
  function get_name() {
    return $this->nombre;
  }
}
?>
```

*En una clase, las **variables** se llaman propiedades y las **funciones** se llaman métodos.*

Vídeo: PHP orientado a objetivos

Visualiza el siguiente vídeo que trata sobre la POO.
https://www.youtube.com/embed/9cb6zgw68n4

2. OBJETOS. DECLARACIÓN Y USO

Las **clases** definen un molde, y los **objetos** es aquello que sacamos de ese molde. Podemos crear múltiples objetos de una clase. Cada objeto tiene todas las propiedades y métodos definidos en la clase, pero tendrán diferentes valores de propiedad.

Los **objetos de una clase** se crean utilizando la palabra reservada `new`.

Ejemplo: creación de objetos de una clase

En el siguiente ejemplo, **$persona1** y **$persona2** son instancias de la clase Persona:

```
<?php
class Persona {
  // Propiedades
  public $nombre;
  public $sexo;
  public $edad;

  // Métodos
  function set_name($nombre) {
    $this->nombre = $nombre;
  }
  function get_name() {
    return $this->nombre;
  }
}

$persona1 = new Persona();
$persona2 = new Persona();
$persona1->set_name('Juan');
$persona2->set_name('María');

echo $persona1->get_name();
```

```
echo "<br>";
echo $persona2->get_name();
?>
```

Tanto los **atributos** como los **métodos** pueden ser de **tres tipos**:

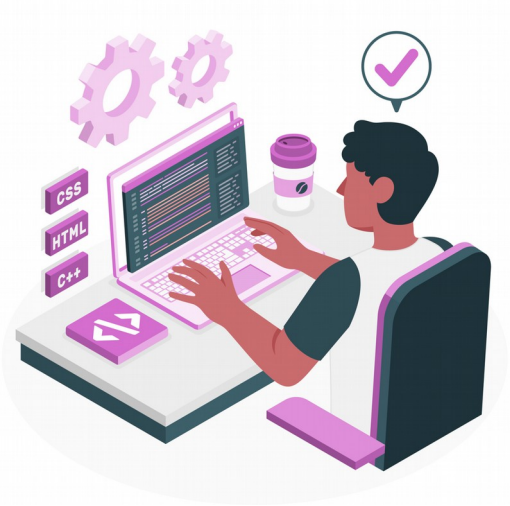
- **Public**: se puede acceder a esta variable desde cualquier sitio.
- **Private**: sólo accesible desde dentro de la clase donde se define (self).
- **Protected**: accesible desde la clase donde se define y las que heredan de ésta.
- **Final**: la clase o método no puede ser sobreescrito en clases descendientes.
- **Abstract**: la clase o método no puede ser usado directamente, ha de ser heredado primero para usarse.

TOMA NOTA

La palabra clave `$this` se refiere al objeto actual y solo está disponible dentro de los métodos.

3. CONSTRUCTORES. DEFINICIÓN DE CONSTRUCTORES

Los **constructores se encargan de realizar acciones de inicialización** de los objetos. Cuando se instancia un objeto, se deben realizar varios pasos en su inicialización como, por ejemplo, dar valor a sus atributos y de ello es de lo que se encargan los constructores. Los constructores pueden recibir datos para inicializar sus objetos.



El método `construct()` comienza con dos guiones bajos (__).

Siguiendo el ejemplo de la clase Persona, ahora crearemos un **constructor** para poder asignar los valores de nombre, sexo y edad en el momento de crear el objeto. Eso nos ahorra el método `set_name()`. También crearemos un **método** llamado `mostrarDatos()` para ver la información de la persona.

Código: creación de constructor y método

```
<?php
class Persona {
  public $nombre;
  public $sexo;
  public $edad;

  function __construct($nombre, $sexo, $edad) {
    $this->nombre = $nombre;
    $this->sexo = $sexo;
    $this->edad = $edad;
  }
```

```
  function mostrarDatos() {
    return "El nombre es " . $this->nombre . ", es ". $this->sexo . " y
tiene " . $this->edad . " años.";
  }
}

$persona1 = new Persona('Juan', 'hombre', 34);
$persona2 = new Persona('María', 'mujer', 30);



echo $persona1->mostrarDatos();
echo "<br>";
echo $persona2->mostrarDatos();
?>
```

4. LOS PILARES FUNDAMENTALES DE LA POO

Existen muchos conceptos en programación orientada a objetos (POO a partir de ahora), como clases y objetos, sin embargo, en el desarrollo de software con POO, hay un conjunto de **ideas fundamentales** que forman los cimientos del desarrollo de software. Estas **ideas o pilares fundamentales** son:

- Abstracción.
- Encapsulamiento.
- Herencia.
- Polimorfismo.

Veamos a continuación un **resumen** de cada uno de ellos:

➔ **Abstracción**.

Agrupar **funcionalidades comunes** de distintos objetos para tener un código más limpio y fácil de trabajar. Por ejemplo, en una tienda de animales se venden perros y gatos porque una de las características comunes es que tienen precio.

➔ **Encapsulación**.

Permite **ocultar propiedades o métodos** del objeto. Por ejemplo, puede que no podamos cambiar la variable `$peso` del gato, pero sí que podremos engordarlo llamando al método `alimentar()`.

➔ **Herencia**.

Hay objetos que **derivan de otros**. Normalmente esto se estructura en clases, pero no tiene por qué ser así (ej. prototipos de JavaScript). Una clase hija hereda propiedades y métodos de una clase padre.

➔ **Polimosfismo**.

Capacidad de que varios objetos de clases diferentes reconozcan un mismo método, pero reaccionen distinto.

- ✔ Método hablar() de un gato = "miau".
- ✔ Método hablar() de un perro = "guau".

Cuestionario

Lee el enunciado e indica la opción correcta:

El método construct() de una clase empieza con:

a. El símbolo $.

b. #.

c. __ (Doble guión de subrayado).

Lee el enunciado e indica la opción correcta:

Un atributo declarado como public:

a. Sólo accesible desde dentro de la clase donde se define (self).

b. Accesible desde la clase donde se define y las que heredan de ésta.

c. Se puede acceder a esta variable desde cualquier sitio.

Lee el enunciado e indica la opción correcta:

Agrupar funcionalidades comunes de distintos objetos para tener un código más limpio y fácil de trabajar, es la definición de:

a. Polimorfismo.

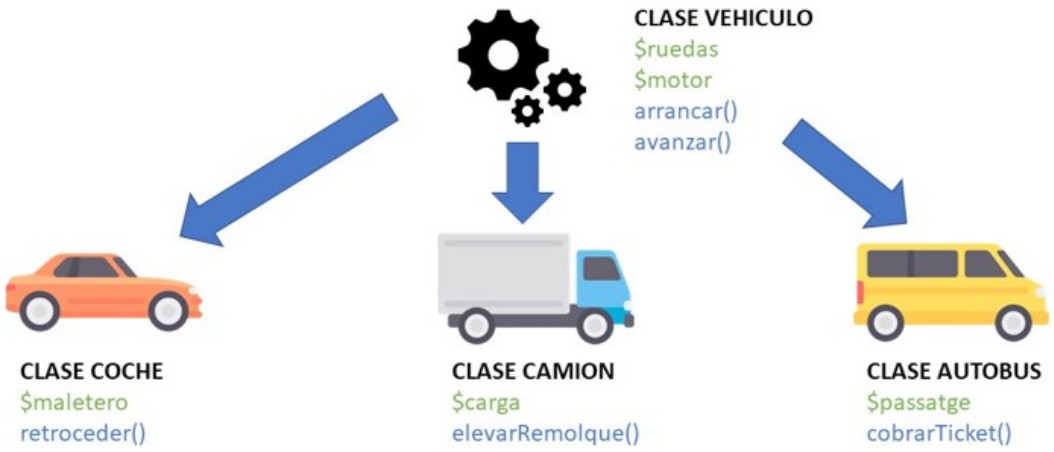
b. Abstracción.

c. Herencia.

5. HERENCIA. REDEFINICIÓN DE MÉTODOS

La **herencia** es uno de los mecanismos fundamentales de la POO. Mediante la herencia, pueden definirse clases a partir de la declaración de otras clases. Las clases que heredan **incluyen tanto los métodos como las propiedades** de la clase a partir de la que están definidos.

Una **clase heredada** se define utilizando la palabra reservada `extends`.



Ejemplo visual de clases y clases heredadas

Veamos un **ejemplo** donde tendremos la clase Vehiculo y la clase Coche para ver como una hereda de la otra.

Código: herencia entre clases

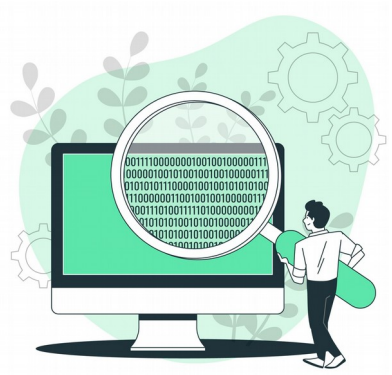
```
<?php
class Vehiculo {
      public $ruedas;
      public $motor;
      public function __construct($ruedas, $motor) {
            $this->ruedas = $ruedas;
            $this-> motor = $motor;
```

```
	}
	public function arrancar() {
		echo "Brrrrr";
	}
	public function avanzar() {
		echo "Rum-rum-rum";
	}
}

class Coche extends Vehiculo {
	public $maletero;
	public function __construct($ruedas, $motor, $maletero) {
		parent::__construct($ruedas, $motor);
		$this->maletero = $maletero;
	}
	public function retroceder() {
		echo "pip-pip-pip";
	}
}

$opel_corsa = new Coche(4, "1.6hdi", 325);
$opel_corsa->arrancar();
$opel_corsa->retroceder();

?>
```

6. ENCAPSULACIÓN

La **encapsulación** se encarga de mantener ocultos los procesos internos, dándole al programador acceso sólo a lo necesario.

*El **aislamiento** protege a las propiedades de un objeto contra su modificación para quienes no tienen derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el **estado interno** de un objeto de manera inesperada. A esto se le conoce como principio de **ocultación**.*



Veamos un **ejemplo**.

➔ **Recuperar y modificar peso**.

Supongamos que tenemos la **clase Perro** que tiene una **propiedad llamada $peso.** Cuando construimos un objeto Perro podemos recuperar y modificar su peso.

```
<?php
class Perro {
      public $peso;
      public function __construct() {
            $this->peso = 1;
      }
}

$toby = new Perro();
$toby->peso = 5;
echo $toby->peso;
?>
```

➔ **Aplicar encapsulación**.

Pero imaginemos que quizás quisiéramos más **control** sobre la forma de cambiar de peso de un perro, por ejemplo, haciendo que se incrementase en una unidad. Pero si, desde fuera de la clase podemos hacer cosas del estilo `$toby->peso = 5`; perdemos ese control.

Una solución es aplicar **encapsulación**, declarando la propiedad `$peso` como privada y haciendo un método público para alimentar. También podemos crear una función para mostrar el peso actual, ya que si la propiedad `$peso` es privada no podremos acceder a ella como hasta ahora. Veamos esa solución:

```
<?php
class Perro {
      private $peso;
      public function __construct() {
            $this->peso = 1;
      }
      public function alimentar() {
            $this->peso++;
      }
       public function get_peso() {
            return $this->peso;
      }
}

$toby = new Perro();
$toby->alimentar();
echo $toby->get_peso();
?>
```

Vídeo: herencia y encapsulación

Visualiza el siguiente vídeo en el que verás un ejemplo de herencia y encapsulación.

Actividad de aprendizaje 1: POO – herencia y encapsulación

Actividad en parejas.

Un miembro de la pareja creará la clase Animal. Después la clase Perro y la clase Gato que heredan de la clase Animal.

Cualquier animal tendrá un constructor con un nombre y un color de pelo. También tendrán el método info() que debe devolver sus atributos.

El otro miembro de la pareja instanciará varios objetos de perros y gatos y probará que el método info() devuelve correctamente la información. También deberá comprobar que no puede cambiar ni el nombre ni el color del animal (excepto en el momento de instanciarlo).

Entrega el resultado final en el espacio habilitado para ello en el foro.

7. POLIMORFISMO

*En programación cuando hablamos de **polimorfismo** nos referimos a la capacidad de acceder a múltiples funciones a través del mismo interfaz. Es decir, que un mismo identificador o función puede tener **diferentes comportamientos** en función del contexto en el que sea ejecutado.*

Vamos a **crear una clase** Animal con el método `hablar()`. Después, crearemos 2 clases hijas, Perro y Gato, que heredan el método hablar, pero cada una de ellas lo sobrescriben según sus necesidades.

Código: herencia y sobreescritura

```
<?php
class Animal {
	public function hablar() {
		return "Los animales emiten sonidos<br/>";
	}
}

class Perro extends Animal {
	public function hablar() {
	return parent::hablar() . "Concretamente los perros hacen GUAU<br/>";
	}
}

class Gato extends Animal {
	public function hablar() {
```

```
     return parent::hablar() . "Concretamente los gatos hacen
MIAU<br/>";
     }
}

$toby = new Perro();
echo $toby->hablar();
$puskas = new Gato();
echo $puskas->hablar();
?>
```

8. ABSTRACCIÓN

*Las **clases abstractas** son clases que no se instancian y sólo pueden ser heredadas, trasladando así un funcionamiento obligatorio a clases hijas. Mejoran la calidad del código y ayudan a reducir la cantidad de código duplicado.*

Veamos en el mismo ejemplo de perros y gatos. Ahora la clase Animal será **abstracta**, no se podrá instanciar. Y su método, `hablar()`, estará vacío esperando que sea implementado por sus clases hijas:

Código: abstracción

```
<?php
abstract class Animal {
      abstract public function hablar();
}

class Perro extends Animal {
      public function hablar() {
      return "Los perros hacen GUAU<br/>";
      }
}

class Gato extends Animal {
      public function hablar() {
      return "Los gatos hacen MIAU<br/>";
      }
}
```

```
$toby = new Perro();
echo $toby->hablar();
$puskas = new Gato();
echo $puskas->hablar();
?>
```

Vídeo: polimorfismo y abstracción

Visualiza el siguiente vídeo en el que verás un ejemplo de polimorfismo y abstracción.

Actividad de aprendizaje 2: POO - abstracción y polimorfismo

De la actividad 1, añade el método hablar para que los gatos digan MIAU y los perros GUAU. También se debe cambiar la clase Animal para que todos sus métodos sean abstractos.

Con tu compañero/a comprueba, a partir de la creación de nuevos animales, que se cumplen los requisitos de este enunciado.

Entrega el resultado final en el espacio habilitado para ello en el foro.

Cuestionario

Lee el enunciado e indica la opción correcta:

Permite ocultar propiedades o métodos del objeto, es la definición de:

a. Polimorfismo.

b. Encapsulación.

c. Herencia.

Lee el enunciado e indica la opción correcta:

Capacidad de que varios objetos de clases diferentes reconozcan un mismo método, pero reaccionen distinto, es la definición de:

a. Encapsulación.

b. Herencia.

c. Polimorfismo.

Lee el enunciado e indica la opción correcta:

¿Se puede instanciar una clase abstracta?:

a. No.

b. Sí, pero solo si tienen métodos no abstractos.

c. Sí.

RESUMEN

En esta unidad didáctica hemos aprendido a trabajar con **PHP** desde el punto de vista de la **POO (Programación Orientada a Objetos)**.

Hemos repasado el concepto de **clases y objetos** y hemos visto los pilares de la POO: **herencia, encapsulación, polimorfismo y abstracción**.

En el futuro podrás decidir si quieres resolver un problema desde el paradigma de la programación orientada a objetos o desde la resolución basada en métodos y procedimientos.

Quizás para resolver pequeños scripts la metodología procedimental será más rápida, y cuando trabajes en proyectos más grandes, con demandas de escalabilidad, la programación orientada a objetos será tu aliada.

RECURSOS PARA AMPLIAR

PÁGINAS WEB

- Tutorial de POO en PHP de w3schools: *https://www.w3schools.com/php/php_oop_what_is.asp* [Consulta noviembre 2022].

BIBLIOGRAFÍA

PÁGINAS WEB

- Referencia oficial de PHP: *https://www.php.net/* [Consulta noviembre 2022].

GLOSARIO

- **Abstracción**: en POO, técnica para agrupar funcionalidades comunas de distintos objetos.
- **Clase**: en POO, una clase es una plantilla para crear objetos.
- **Encapsulación**: en POO, técnica para ocultar propiedades o métodos de una clase.
- **Herencia**: en POO, capacidad que tienen las clases para derivar de otras.
- **Objeto**: en POO, un objeto es una instancia de una clase.
- **Polimorfismo**: en POO, capacidad que tienen objetos distintos de reconocer el mismo método.
- **POO**: siglas de Programación Orientada a Objetos.