ÍNDICE

1. Introducción a la Programación Orientada a Objetos

La programación orientada a objetos (POO, u OOP según sus siglas en inglés) es un paradigma de programación que usa objetos en sus interacciones, para diseñar aplicaciones y programas informáticos.

1.1. Elementos de la POO

Clase. Una clase es un modelo que se utiliza para crear objetos que comparten un mismo comportamiento, estado e identidad. Por ejemplo, la clase persona puede servir para crear seres de dos piernas inteligentes que tengan comportamientos comunes como comer, dormir…

```
class Persona{
    #propiedades
    #métodos
{
```

Objeto. Es una entidad provista de métodos o mensajes a los cuales responde (comportamiento); atributos con valores concretos (estado); y propiedades (identidad).

```
$persona=new persona();
```

Método. Es una función asociada al objeto que le indica qué puede hacer

```
function hablar(){
…
}
```

Eventos y mensajes. Un evento es un suceso en el sistema mientras que un mensaje es la comunicación del suceso dirigida al objeto.

Propiedades y atributos. Las propiedades y atributos, son variables que contienen datos asociados a un objeto.

```
$nombre = 'Carmen';
$edad = '4 años';
$altura = '1,06 mts';
```

1.2. Características conceptuales de la POO

La POO tiene ciertas características que la identifican y diferencian de otros paradigmas de programación, éstas son:

Abstracción. Define las características esenciales de un objeto aislándolo de su contexto.

Encapsulamiento. Reúne al mismo nivel de abstracción, a todos los elementos que puedan considerarse pertenecientes a una misma entidad.

Modularidad. Permite dividir una aplicación en varias partes más pequeñas (denominadas módulos), independientes unas de otras.

Ocultación (aislamiento). Los objetos están aislados del exterior, protegiendo a sus propiedades para no ser modificadas por aquellos que no tengan derecho a acceder a las mismas. ´

Polimorfismo. Es la capacidad que da a diferentes objetos, la posibilidad de contar con métodos, propiedades y atributos de igual nombre, sin que los de un objeto interfieran con el de otro.

Herencia. Es la relación existente entre dos o más clases, donde una es la principal (madre) y otras son secundarias (hijas) y dependen (heredan) de ellas, además, los objetos heredan las características de los objetos de los cuales heredan.

Recolección de basura. Es la técnica que consiste en destruir aquellos objetos cuando ya no son necesarios, liberándolos de la memoria.

2. Clases y objetos en PHP

2.1. Definición de clases

Una clase es una colección de variables y funciones que trabajan con estas variables. Las variables se definen utilizando la palabra reservada **var** y las funciones utilizando **function**.

La definición básica de clases comienza con la palabra clave **class**, seguido por un nombre de clase. El nombre de clase puede ser cualquier etiqueta válida que no sea una palabra reservada de PHP. Un nombre válido de clase comienza con una letra o un guión bajo, seguido de la cantidad de letras, números o guiones bajos que sea, a continuación de abrirán llaves.

```
class NombreClase {
  #...
}
```

3. Objetos en PHP

Una vez que las clases han sido declaradas, será necesario crear los objetos y utilizarlos, a menos que se trate de una clase abstracta que son solo modelos para otras, y por lo tanto no necesitan instanciar al objeto.

3.1. Instanciar una clase

Para instanciar una clase, solo es necesario utilizar la palabra clave **new**. El objeto será creado, asignando esta instancia a una variable (la cual, adoptará la forma de objeto). Lógicamente, la clase debe haber sido declarada antes de ser instanciada, como se muestra a continuación:

```
# declaro la clase
class Persona {
     #...
}
# creo el objeto instanciando la clase
$persona = new Persona();
```

3.2. Tipos de propiedades del objeto

Las propiedades representan ciertas características del objeto en sí mismo. Se definen anteponiendo la propiedad correspondiente al nombre de la variable:

```
class Persona {
    propiedad $nombre;
    propiedad $edad;
    propiedad $genero;
}
```

Las propiedades pueden tener diferentes características, como por ejemplo, la visibilidad: pueden ser públicas, privadas o protegidas. Como veremos más adelante, la visibilidad de las propiedades, es aplicable también a la visibilidad de los métodos.

Propiedades públicas. Las propiedades públicas se definen anteponiendo la palabra clave public al nombre de la variable. Éstas, pueden ser accedidas desde cualquier parte de la aplicación, sin restricción.

```
class Persona {
    public $nombre;
    public $genero;
}
```

Propiedades privadas. Las propiedades privadas se definen anteponiendo la palabra clave private al nombre de la variable. Éstas solo pueden ser accedidas por la clase que las definió.

```
class Persona {
    public $nombre;
    public $genero;
    private $edad;
}
```

Propiedades protegidas. Las propiedades protegidas pueden ser accedidas por la propia clase que la definió, así como por las clases que la heredan, pero no, desde otras partes de la aplicación. Se definen anteponiendo la palabra clave protected al nombre de la variable:

```
class Persona {
    public $nombre;
    public $genero;
    private $edad;
    protected $pasaporte;
}
```

Propiedades estáticas. Las propiedades estáticas representan una característica de "variabilidad" de sus datos. Una propiedad declarada como estática, puede ser accedida sin necesidad de instanciar un objeto. y su valor es **estático** (es decir, no puede variar ni ser modificado). Se define anteponiendo la palabra clave **static** al nombre de la variable:

```
class Libro {
            private $titulo;
            private $autor;
            private $paginas;
            #genero una variable de clase estática, no hay que instanciarla
para acceder a ella
            public static $referencia = 1;
}
```

3.3. Acceso a las propiedades de un objeto

La forma de acceder a las propiedades de un objeto dependerá del ámbito desde el cual se las invoque, así como de su condición y visibilidad.

3.3.1. Acceso a variables desde el ámbito de la clase

Se accede a una propiedad no estática dentro de la clase, utilizando la pseudo-variable **$this** siendo esta pseudo-variable una referencia al objeto mismo:

```
return $this->nombre;
```

Cuando la variable es estática, se accede a ella mediante el operador de resolución de ámbito, doble dos-puntos **::** anteponiendo la palabra clave self o parent según si trata de una variable de la misma clase o de otra de la cual se ha heredado, respectivamente:

```
print self::$variable_estatica_de_esta_clase;
 print parent::$variable_estatica_de_clase_madre;
```

Se puede cambiar self:: por el propio nombre de la clase.

3.3.2. Acceso a variables desde el exterior de la clase

Se accede a una propiedad no estática con la siguiente sintaxis:

```
$objeto->variable
```

Nótese además, que este acceso dependerá de la visibilidad de la variable. Por lo tanto, solo variables públicas pueden ser accedidas desde cualquier ámbito fuera de la clase o clases heredadas.

```
# creo el objeto instanciando la clase
$libro1 = new Libro();
 # accedo a la variable NO estática
print $libro1->titulo;
```

Para acceder a una propiedad pública y estática el objeto no necesita ser instanciado, permitiendo así, el acceso a dicha variable mediante la siguiente sintáxis: **Clase::$variable_estática**

```
# accedo a la variable estática
print PersonaAPositivo::$tipo_sangre;
```

3.4. Constantes de clase

Otro tipo de “propiedad” de una clase, son las constantes, aquellas que mantienen su valor de forma permanente y sin cambios. A diferencia de las propiedades estáticas, las constantes solo pueden tener una visibilidad pública. Puede declararse una constante de clase como cualquier constante normal en PHP, igualmente por cuestión de estilo se recomiendo el uso de mayúsculas para designar constantes.

```
const MI_CONSTANTE = 'Este es el valor estático de mi constante';
```

3.5. Métodos

El método de una clase, es un algoritmo igual al de una función. La única diferencia entre método y función, es que llamamos método a las funciones de una clase (en la POO), mientras que llamamos funciones, a los algoritmos de la programación estructurada.

La forma de declarar un método es anteponiendo la palabra reservada **function** al nombre del método, seguido por un par paréntesis de apertura y cierre y llaves que encierren el algoritmo:

```
# declaro la clase
class Persona {
     #propiedades
     #métodos
      function donar_sangre($destinatario) {
```

```
            #...
        }
}
```

Al igual que cualquier otra función en PHP, los métodos recibirán los parámetros necesarios indicando aquellos requeridos, dentro de los paréntesis.

3.5.1. Métodos públicos, privados, protegidos y estáticos

Los métodos, al igual que las propiedades, pueden ser públicos, privados, protegidos o estáticos. La forma de declarar su visibilidad tanto como las características de ésta, es exactamente la misma que para las propiedades.

```
static function a() { }
protected function b() { }
private function c() { }
# etc...
```

3.5.2. Métodos abstractos

Los métodos definidos como abstractos simplemente declaran la estructura del método, pero no pueden definir la implementación. Cuando se hereda de una clase abstracta, todos los métodos definidos como abstract en la definición de la clase padre deben ser redefinidos en la clase hija; adicionalmente, estos métodos deben ser definidos con la misma visibilidad (o con una menos restrictiva). Por ejemplo, si el método abstracto está definido como protected, la implementación de la función puede ser redefinida como protected o public, pero nunca como private.

Podríamos decir que, a grandes rasgos, los métodos abstractos son aquellos que se declaran inicialmente en una clase abstracta, sin especificar el algoritmo que implementarán, es decir, que solo son declarados pero no contienen un “código” que especifique qué harán y cómo lo harán.

3.5.3. Métodos mágicos

Estos métodos, otorgan una funcionalidad pre-definida por PHP, que pueden aportar valor a nuestras clases y ahorrarnos grandes cantidades de código.

__construct

El método __construct() es aquel que será invocado de manera automática, al instanciar un objeto. Su función es la de ejecutar cualquier inicialización que el objeto necesite antes de ser utilizado.

```
# declaro la clase
class Producto {
      #defino algunas propiedades
       public $nombre;
       public $precio;
       protected $estado;
       #defino el método set_estado_producto()
       protected function set_estado_producto($estado) {
                  $this->estado = $estado;
        }
       # constructor de la clase
```

```
      function __construct() {
                  $this->set_estado_producto('en uso');
          }
}
```

En el ejemplo anterior, el constructor de la clase se encarga de definir el estado del producto como “en uso”, antes de que el objeto (Producto) comience a utilizarse. Si se agregaran otros métodos, éstos, podrán hacer referencia al estado del producto, para determinar si ejecutar o no determinada función.

__destruct

El método __destruct() es el encargado de liberar de la memoria, al objeto cuando ya no es referenciado. Se puede aprovechar este método, para realizar otras tareas que se estimen necesarias al momento de destruir un objeto.

```
# declaro la clase
class Producto {
     #defino algunas propiedades
      public $nombre;
      public $precio;
      protected $estado;
     #defino el método set_estado_producto()
     protected function set_estado_producto($estado) {
                 $this->estado = $estado;
      }
    # constructor de la clase
      function__construct() {
                   $this->set_estado_producto('en uso');
      }
     # destructor de la clase
      function__destruct() {
                     $this->set_estado_producto('liberado');
                      print 'El objeto ha sido destruido';
       }
}
```

PHP nos ofrece otros métodos como __call, __callStatic, __get, __set, __isset, __unset, __sleep, __wakeup, __toString, __invoke, __set_state y __clone.

Puede verse una descripción y ejemplo de su uso, en el sitio Web oficial de PHP:
http://www.php.net/manual/es/language.oop5.magic.php

4. Modificadores de acceso a métodos y propiedades

4.1. Modificador Public

Es el nivel de acceso más permisivo. Sirve para indicar que el método o atributo de la clase es público. En este caso se puede acceder a ese atributo, para visualizarlo o editarlo, por cualquier otro elemento de nuestro programa. Es el modificador que se aplica si no se indica otra cosa.

Veamos un ejemplo de clase donde hemos declarado como public sus elementos, un método y una propiedad. Se trata de la clase "dado", que tiene un atributo con su puntuación y un método para tirar el dado y obtener una nueva puntuación aleatoria.

```php
class dado{
    public $puntos;

    function __construct(){
        srand((double)microtime()*1000000);
    }

    public function tirate(){
        $this->puntos=$randval = rand(1,6);
    }
}

$mi_dado = new dado();

for ($i=0;$i<30;$i++){
    $mi_dado->tirate();
    echo "<br>Han salido " . $mi_dado->puntos . " puntos";
}
```

En el ejemplo se realiza un bucle 30 veces, en las cuales se tira el dado y se muestra la puntuación que se ha obtenido.

Como el atributo $puntos y el método tirate() son públicos, se puede acceder a ellos desde fuera del objeto, o lo que es lo mismo, desde fuera del código de la clase.

4.2. Modificador Private

Es el nivel de acceso más restrictivo. Sirve para indicar que esa variable sólo se va a poder acceder desde el propio objeto, nunca desde fuera. Si intentamos acceder a un método o atributo declarado private desde fuera del propio objeto, obtendremos un mensaje de error indicando que no es posible a ese elemento.

Si en el ejemplo anterior hubiéramos declarado private el método y la propiedad de la clase dado, hubiéramos recibido un mensaje de error.

Aquí tenemos otra posible implementación de la clase dado, declarando como private el atributo puntos y el método tirate().

```php
class dado{
    private $puntos;

    function __construct(){
        srand((double)microtime()*1000000);
    }

    private function tirate(){
        $this->puntos=$randval = rand(1,6);
    }

    public function dame_nueva_puntuacion(){
        $this->tirate();
        return $this->puntos;
    }
}

$mi_dado = new dado();

for ($i=0;$i<30;$i++){
    echo "<br>Han salido " . $mi_dado->dame_nueva_puntuacion() . " puntos";
}
```

Hemos tenido que crear un nuevo método público para operar con el dado, porque si es todo privado no hay manera de hacer uso de él. El mencionado método es dame_nueva_puntuación(), que realiza la acción de tirar el dado y devolver el valor que ha salido.

4.3. Modificador Static

Ejemplos de situaciones de la vida real donde tendríamos miembros estáticos, pensemos en los autobuses de una ciudad. Generalmente todos los autobuses metropolitanos tienen la misma tarifa. Yo podría definir como un atributo de la clase AutobúsMetropolitano su precio. En condiciones normales, para acceder al precio de un autobús necesitaría instanciar un objeto autobús y luego consultar su precio. ¿Es esto práctico? quizás solo quiero saber su precio para salir de casa con dinero suficiente para pagarlo, pero en el caso de un atributo normal necesariamente debería tener instanciado un autobús para preguntar su precio.

```php
<?php
class AutobusMetropolitano {

    static $precio = 1.20;
    public function aceptarPago($dinero) {
        if ($dinero < self::$precio) {
            return FALSE;
        } else{
            return TRUE;
        }
    }
}

$ahorrado=1;
```

```
if (AutobusMetropolitano::aceptarPago($ahorrado)) {
    print('Puedes coger un autobús');
} else {
    echo 'necesitas ahorrar ', AutobusMetropolitano::$precio-$ahorrado;
}
```

4.4. Modificador Protected

Este indica un nivel de acceso medio y un poco más especial que los anteriores. Sirve para que el método o atributo sea **público dentro del código de la propia clase y de cualquier clase que herede de aquella donde está el método o propiedad protected.**

Es privado y no accesible desde cualquier otra parte. Es decir, un elemento protected es público dentro de la propia clase y en sus heredadas.

5. Herencia

Suponemos que tenemos una clase Coche como la que se define a continuación

```
<?
class Coche {
    protected $marca;
    protected $matricula;
    protected $numIdentificador;
    protected $numPuertas;


    public function __construct($marca, $matricula, $numIdentificador,
$numPuertas) {
       $this->marca = $marca;
       $this->matricula = $matricula;
       $this->numIdentificador = $numIdentificador;
       $this->numPuertas = numPuertas;
    }?>
```

Una clase con tres propiedades que definen la marca, la matrícula y el identificador del coche.

Ahora supongamos que tenemos una clase para definir el prototipo de una moto. Sería más o menos así:

```
<?
class Moto {
    protected $marca;
    protected $matricula;
    protected $numIdentificador;



    public function __construct($marca, $matricula, $numIdentificador) {
       $this->marca = $marca;
       $this->matricula = $matricula;
       $this->numIdentificador = $numIdentificador;
    }?>
```

La clase Moto es prácticamente idéntica a la clase Coche, sólo que con una mínima diferencia: la clase Moto no tiene una propiedad $numPuertas, porque sencillamente una moto no tiene puertas.

Ahora, ¿por qué estás clases son tan parecidas? Si lo pensamos como personas y no como programadores (ésa es una de las cosas que nos permite la programación orientada a objetos) ¿Qué tienen en común un coche y una moto? Que ambos son vehículos.

Así que entonces podríamos tener una clase Vehiculo también:

```
<?php
 class Vehiculo {
    protected $marca;
    protected $matricula;
    protected $numIdentificador;



    public function __construct($marca, $matricula, $numIdentificador) {
       $this->marca = $marca;
       $this->matricula = $matricula;
       $this->numIdentificador = $numIdentificador;
    }

    function muestraCaracteristicas(){
        print $this->marca .'<br>';
        print $this->matricula .'<br>';
        print $this->numIdentificador .'<br>';
    }


 }
```

Y ahora volveremos a crear nuestras clases Coche y Moto, pero heredando de esta última clase. En el caso de la clase Coche añadimos el campo $numeroPuertas que no contenía la clase padre.

```
class Coche extends Vehiculo {
    private $numeroPuertas;
    public function __construct($marca, $matricula, $numIdentificador,
$numPuertas) {
       parent::__construct($marca, $matricula, $numIdentificador);
       $this->numeroPuertas=$numPuertas;


    }
       function muestraCaracteristicas(){
        parent::muestraCaracteristicas();
        print $this->numeroPuertas. '<br>';
    }
 }
```

La clase Moto

```
class Moto extends Vehiculo {
    public function __construct($marca, $matricula, $numIdentificador) {
       parent::__construct($marca, $matricula, $numIdentificador);
    }
 }
```

Ambas clases tienen ahora mucho menos código. Ahora vamos a lo importante que permite la herencia:

La herencia permite que clases como Coche y Moto hereden de una clase madre, en nuestro caso la clase Vehiculo. ¿Y qué es lo que heredan? Las clases hijas heredan de las clases madres todas las propiedades y métodos, lo que permite ahorrar líneas de código innecesarias.

Ahora varios puntos para repasar.

Primero, dentro las clases hijas debe incluirse con require_once, u otra función de PHP para importar archivos el .php donde se encuentra la clase madre, a menos que estén en el mismo documento como en el ejemplo.

Además cuando definimos la clase debemos utilizar la palabra reservada extends más el nombre de la clase padre:

```
class ClaseHija extends ClasePadre {}
```

Para llamar a métodos de una clase hija a una clase madre no lo hacemos con $this->metodo(), sino con parent::metodo(). En el caso de las propiedades sí se llaman con $this->propiedad.

Y por último, hay que fijarse que en la clase madre no usamos private para las propiedades sino que usamos otra palabra reservada, protected.

Repasemos, public era para que las propiedades y métodos se puedan acceder desde la clase y el objeto, mientras que private era sólo para que se acceda desde la clase. En el caso de protected es un intermedio, las propiedades y métodos no podrán accederse desde el objeto, pero sí se podrán acceder desde las clases y las clases que hereden esos métodos. Si yo le pusiera private a las propiedades de la clase Vehiculo, la clase Coche y Moto no podrían acceder a las mismas.

Unos ejemplos de objetos basados en ambas clases serían

objeto Coche:

```
$coche1=new Coche('Mazda', '4563GHZ', 'e3', '4');
$coche1->muestraCaracteristicas();
```

objeto Moto:

```
$moto1=new Moto('Vespa', '5566ttt', '2m');
$moto1->muestraCaracteristicas();
```

5.1. Métodos y clases abstractos

Las clases abstractas se utilizan como patrones de diseño ya que obligan a que las clases hijas tengan definidos todos los métodos declarados como abstractos en la clase padre.

Un método abstracto se caracteriza por dos detalles

1. Utiliza como prefijo la palabra clave abstract
2. No tiene cuero del método, en lugar de ello su cabecera se termina con un ;

No solo los métodos pueden declararse como abstractos, sino que también pueden declararse como abstractas las clases. Las clases de declaran como abstractas insertando la palabra abstract en la cabecera de la clase.

La declaración de una clase como abstracta tiene varios objetivos:

- No pueden crearse instancias de las clases abstractas. Tratar de utilizar la palabra clave new con una clase abstracta es un error y el compilador no lo permitirá.
- Solo las clases abstractas pueden disponer de métodos abstractos.
- Las clases abstractas con métodos abstractos obligan a las subclases a sustituir e implementar esos métodos declarados como abstractos.

Si tenemos este código

```
abstract class Vehiculo {
    protected $marca;
    protected $matricula;
    protected $numIdentificador;

    public function __construct($marca, $matricula, $numIdentificador) {
        $this->marca = $marca;
        $this->matricula = $matricula;
        $this->numIdentificador = $numIdentificador;
    }
     abstract function muestraCaracteristicas();
 }
```

La clase hija Coche no funcionará a menos que definamos el método muestraCaracteristicas() dentro de ella

```
class Coche extends Vehiculo {
    private $numeroPuertas;
    public function __construct($marca, $matricula, $numIdentificador,
$numPuertas) {
        parent::__construct($marca, $matricula, $numIdentificador);
        $this->numeroPuertas=$numPuertas;

    }
     //Es necesario definir muestraCaracteristicas porque existe en la clase
padre como abstracto.
```

```
        public function muestraCaracteristicas(){
        print $this->marca .'<br>';
        print $this->matricula .'<br>';
        print $this->numIdentificador .'<br>';
        print $this->numeroPuertas. '<br>';
    }
```

5.2. Interfaces

Las interfaces de objetos permiten crear código con el cual especificar qué métodos deben ser implementados por una clase, sin tener que definir cómo estos métodos son manipulados.

La herencia múltiple es el fenómeno que se produce en aquellos casos en que la clase tiene más de una superclase inmediata. La subclase tiene entonces todas las características de ambas superclases además de las definidas en la propia subclase. PHP no permite la herencia múltiple de clases, pero proporciona otra estructura, denominada interface, que permite una forma limitada de herencia múltiple.

```
<?php

class Vehiculo {
    protected $marca;
    protected $matricula;
    protected $numIdentificador;

    public function __construct($marca, $matricula, $numIdentificador) {
        $this->marca = $marca;
        $this->matricula = $matricula;
        $this->numIdentificador = $numIdentificador;
    }

}

interface CaracteristicasVehiculo{
    public function muestraCaracteristicas();
}

class Coche extends Vehiculo implements CaracteristicasVehiculo {

    private $numeroPuertas;

    public function __construct($marca, $matricula, $numIdentificador,
$numPuertas) {
        parent::__construct($marca, $matricula, $numIdentificador);
```

```
        $this->numeroPuertas = $numPuertas;
    }



    //Es necesario definir muestraCaracteristicas porque existe en la clase padre como abstracto.
     public function muestraCaracteristicas() {
        print $this->marca . '<br>';
        print $this->matricula . '<br>';
        print $this->numIdentificador . '<br>';
        print $this->numeroPuertas . '<br>';
    }

}
class Moto extends Vehiculo {

    public function __construct($marca, $matricula, $numIdentificador) {
        parent::__construct($marca, $matricula, $numIdentificador);
    }

   /* Aquí no es necesario incluir la función muestraCaracterísticas() porque la clase Moto no implementa esa Interfaz    */
}
```

Como se ve en el código en la clase Coche es necesario definir el método muestraCaracteristicas() porque se implementa la interfaz CaracterísticasVehículo, no siendo necesario en la clase Moto

5.3. Declaraciones de clases finales.

PHP 5 incorpora clases finales que no pueden ser heredadas por otra. Se definen anteponiendo la palabra clave **final**.

```
final class NombreClaseFinal {
  #esta clase no podrá ser heredada
}
```

6. Fuentes

https://desarrolloweb.com/articulos/1788.php

http://www1.herrera.unt.edu.ar/biblcet/wp-content/uploads/2014/12/eugeniabahitpooymvcenphp.pdf

http://www.php.net/manual/es/language.oop5.magic.php

http://php.net/