PHP orientado a objetos

https://fernando-gaitan.com.ar/

Fernando Gonzalo Gaitán

Índice

Al lector

Este documento ha sido creado con fines educativos. Podés copiar y utilizar cualquier parte de su contenido, sin embargo se deberá citar la fuente.

Además el mismo ha sido desarrollado el día 16/01/2016, por este motivo pudiesen existir problemas de incompatibilidad con versiones actuales.

Para seguir en contacto conmigo, podés hacerlo a través de las siguientes redes sociales:

Clases, propiedades y métodos

Bien, en los siguientes artículos que escribiré a continuación explicaré de qué se trata la programación orientada a objetos en PHP. Trataré de hacerlo lo más claro y fácil posible.

Bueno, comencemos. Si no sabés a qué se refiere el concepto de ‘objetos’ en programación, y aunque el término puede resultar confuso, y quizás al principio te cueste entenderlo. Simplemente te diré que un objeto en programación puede ser cualquier cosa.

Osea, un objeto es todo aquello que tiene propiedades (valores) y métodos (comportamientos). Las propiedades permiten guardar datos del objeto, cumplen la misma función que una variable; mientras que los métodos permiten realizar operaciones, cumplen la misma función que, valga la redundancia, una función.

Un ejemplo fácil puede ser por ejemplo una persona. Una persona puede ser un objeto porque tiene propiedades, valores. Una persona tiene un nombre, un apellido, una fecha de nacimiento, un número de documento, si es hombre o mujer, etc; y estas serían algunas de las propiedades de un objeto persona. Pero también tiene métodos o comportamientos, como dormir, comer, caminar, correr, etc.

También un auto podría ser un objeto porque tiene propiedades como el color, la marca, la patente, si está encendido o apagado el motor. Y tiene métodos, como acelerar, frenar, encender el motor o apagar el motor. A su vez estas propiedades y métodos trabajan juntas. Por ejemplo, como acabo de decir, el objeto auto puede tener una propiedad encendido que guarde un valor boolean, si es true es porque está encendido el motor y si es false es por está apagado; y también tener un método para encender el motor o apagarlo, lo cual alteraría el valor de dicha propiedad según el método que se llame.

Ahora bien, ¿cómo puedo hacer yo para crear un objeto en PHP? Para ello necesito una clase. Una clase es como un molde, un modelo. Algo que defina la estructura del objeto. Por ejemplo si yo tuviese un objeto edificio, la clase sería el plano de un arquitecto para crear ese edificio u otros edificios similares.

En primer lugar, y aunque en este lenguaje de programación no es obligatorio, una clase debería crearse en un archivo PHP que no comparta más código que la misma clase. Para ello, y para empezar a probar código, creá un archivo .php con un nombre, por ejemplo ‘test.php’. A la misma altura crear una carpeta llamada ‘clases’ y dentro de esta carpeta crear un archivo con el nombre ‘Persona.php’

Bueno, ahora editaremos ese último archivo y crearemos nuestra primer clase PHP.

Primero que nada las clases se escriben con la palabra reservada **class** y el nombre de la misma, éste debería ir con mayúsculas la primer letra, si bien esto último no es obligatorio es una buena práctica escribirla así, ya que de esta forma se acostumbra. Luego dentro de llaves {} irán las propiedades y métodos que definiremos dentro de la clase.

Crearemos una clase Persona, de esta forma (en nuestro archivo Persona.php):

```
<?php
 class Persona {
    //Acá dentro va el código
 }
?>
```

Y dentro de esta clase tendremos tres propiedades: nombre, apellido y edad:

```
<?php
 class Persona {
```

```
    public $nombre;
    public $apellido;
    public $edad;
 }
?>
```

Seguramente habrás notado que las propiedades en PHP se escriben igual que una variable, con el signo $ adelante y el nombre. Pero ¿qué significa la palabra **public**? Eso lo veremos más adelante, por ahora sólo ignorala.

Bueno, ahora ya que tenemos nuestra clase creada, iremos a nuestro archivos **test.php** y crearemos nuestro primer objeto. Primero importamos nuestra clase **Persona**, para tener el modelo del objeto a crear con require_once:

```
<?php
   require_once 'clases/Persona.php';
?>
```

Crearemos nuestro objeto de esta forma:

```
<?php
 require_once 'clases/Persona.php';
 //Creamos el objeto.
 $persona = new Persona();
 //Seteamos las propiedades.
 $persona->nombre = 'Fernando';
 $persona->apellido = 'Gaitan';
 $persona->edad = 26;
 //Mostramos el resultado de las propiedades.
 echo 'Nombre: ' . $persona->nombre . '<br />';
 echo 'Apellido: ' . $persona->apellido . '<br />';
 echo 'Edad: ' . $persona->edad . '<br />';
?>
```

Ok, primero incluimos la clase **Persona**, luego creamos el objeto con la palabra reservada **new**, de esta forma le aclaramos a nuestro programa que estamos creando un nuevo objeto basado en la clase **Persona**.

Seguramente habrán notado que un objeto tiene la misma forma que una variable (de hecho es una variable) con el signo $ y el nombre de la variable. Luego, desde el objeto llamamos a las propiedades $nombre, $apellido y $edad con una especie de flecha **->**, que nos permite acceder a los mismos; pero ojo, las propiedades se definen dentro de la clase con $ más nombre de la propiedad, sin embargo como verán, al ser invocados por el objeto no se usa el signo $, sino que directamente se pone el nombre. Y finalmente mostramos por pantalla los valores con **echo**. Ejecuta en tu navegador el archivo **test.php** para comprobarlo.

Bueno, ahora sólo me falta mostrar cómo se crea un método dentro de una clase, por lo que iremos nuevamente a nuestro archivo **Persona.php** y los editaremos con estas líneas:

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

```
    public $apellido;
    public $edad;
    public function saludar(){
       return 'Hola, soy ' . $this->nombre . ' ' . $this->apellido . ' y
tengo ' . $this->edad . ' años ';
    }
 }
?>
```

Como ven, también creamos un método **saludar()** que tiene la misma estructura que una función, sólo que con la palabra reservada **public** (insisto, más adelante veremos qué significa esa palabra) adelante. Y dentro del mismo utilizamos **$this->** para llamar a las propiedades de la clase. Siempre que dentro de un método necesitemos acceder a propiedades u otros métodos de la misma clase, usaremos $this-> más el nombre de la propiedad o método.

Probaremos el comportamiento del método **saludar()** en **test.php** de la siguiente forma:

```
<?php
 require_once 'clases/Persona.php';
 //Creamos el objeto.
 $persona = new Persona();
 //Seteamos las propiedades.
 $persona->nombre = 'Fernando';
 $persona->apellido = 'Gaitan';
 $persona->edad = 26;
 //Mostramos el resultado de las propiedades.
 echo $persona->saludar();
?>
```

Ejecuta **test.php** en tu navegador y mirá el resultado.

Bueno, hasta acá termina la primer parte. Prometo que las próximas serán un poco más cortas.

Saludos a todos.

Método constructor y destructor

Bueno, en mi anterior posteo publiqué una introducción en donde mostraba como crear una clase y luego un objeto en base a la misma, Php orientado a objetos, parte 1: Clases, propiedades y métodos.

Ahora bien, en aquella ocasión yo tenía una clase Persona con este aspecto:

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

```
    public $apellido;
    public $edad;
    public function saludar(){
       return 'Hola, soy ' . $this->nombre . ' ' . $this->apellido . ' y
tengo ' . $this->edad . ' años ';
    }
 }
```

Luego, creaba un objeto Persona:

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

A su vez este objeto tenía un método **saludar()**, que era invocado por el objeto persona de esta forma:

```
$persona->saludar();
```

Ahora bien, en la programación orientada a objetos nosotros tenemos la posibilidad de crear un método llamado constructor, que a diferencia de los otros métodos no es invocado, sino que se dispara en el preciso momento en que nosotros creamos el objeto. Osea cuando hacemos:

```
$objeto = new Clase();
```

Esto es muy útil ya que permitirá en dicho método constructor definir todas aquellas características que tiene un objeto al ser creado.

Para ello haremos lo siguiente. Iremos a nuestro archivo **Persona.php**, donde tenemos nuestra clase y le haremos algunos cambios.

Probemos cómo funciona el método constructor de la siguiente forma:

```
<?php
 class Persona {
    public function __construct() {
       echo 'Se acaba de crear el objeto persona';
    }
 }
?>
```

Y ahora probemos su funcionamiento de la siguiente manera:

```
<?php
 require_once 'clases/Persona.php';
 $persona = new Persona();
?>
```

Ejecutá tu archivo **test.php** y comprobá su funcionamiento. Seguramente habrás notado que el objeto al ser creado llama al método **__construct()**, pero sin ser invocado con la flecha **->**.

El método constructor en PHP se escribe como un método mágico, por tanto nosotros no podremos ponerle cualquier nombre al método, sino que tendrá que ser **__construct()**.

Volvamos a modificar la clase **Persona** casi cómo estaba antes, pero con el método constructor y en dicho método definiremos el valor de las propiedades de la clase **$nombre**, **$apellido** y **$edad**. Debería quedar así:

```
<?php
 class Persona {
    public $nombre;
    public $apellido;
    public $edad;
    public function __construct($nombre, $apellido, $edad) {
       $this->nombre = $nombre;
       $this->apellido = $apellido;
       $this->edad = $edad;
    }
    public function saludar(){
       return 'Hola, soy ' . $this->nombre . ' ' . $this->apellido . ' y
tengo ' . $this->edad . ' años ';
    }
 }
?>
```

Cómo verán, en el anterior posteo nosotros teníamos que, luego de crear nuestro objeto **$persona**, acceder desde el mismo para definir los valores de las propiedades **$nombre**, **$apellido** y **$edad**. En este caso el constructor será el encargado de definir los valores de estas propiedades, por tanto, nosotros al crear el objeto estamos obligados a setear los valores de estas propiedades.

Ahora bien, te preguntarás cómo definir los valores de las propiedades del objeto **$persona**, ya que están en el constructor, y el constructor no se invoca desde el objeto, sino que se llama automáticamente. Bueno, seguramente habrás notado que cuando nosotros creamos un objeto con **new Clase()**, se utiliza paréntesis, bueno esos paréntesis dentro pueden recibir parámetros, y esos parámetros son los mismos que el método constructor. Por tanto, para crear el objeto y definir el valor de las propiedades lo haremos de la siguiente manera:

```
<?php
 //Importamos la clase Persona.
 require_once 'clases/Persona.php';
 //Creamos el objeto con los valores que se definen en el constructor.
 $persona = new Persona('Fernando', 'Gaitan', 26);
?>
```

Ahora solamente nos queda invocar el método **saludar()** para mostrar por pantalla los valores que acabamos de setear. Así que haremos lo siguiente:

```
<?php
 //Importamos la clase Persona.
 require_once 'clases/Persona.php';
 //Creamos el objeto con los valores que se definen en el constructor.
 $persona = new Persona('Fernando', 'Gaitan', 26);
```

```
 //Mostramos por pantalla los valores.
 echo $persona->saludar();
?>
```

Ok. Ahora que sabemos de que se trata un método constructor podemos aprender a usar un método destructor. Los métodos destructores, son lo contrario a un constructor. Los constructores se disparan cuando el objeto se crea, mientras que los destructores se disparan en cuanto se borra de memoria. Si bien estos métodos no suelen utilizarse a menudo como sí pasa con los constructores, no está de más entender su funcionamiento.

Primero que nada para destruir un objeto tenemos que usar la función de PHP **unset()** que sirve para borrar cualquier tipo de variable de memoria, incluso un objeto. Esto puede resultar últil para liberar memoria en nuestro script.

Así que para destruir nuestro objeto agregaremos lo siguiente:

```
<?php
 //Importamos la clase Persona.
 require_once 'clases/Persona.php';
 //Creamos el objeto con los valores que se definen en el constructor.
 $persona = new Persona('Fernando', 'Gaitan', 26);
 //Mostramos por pantalla los valores.
 echo $persona->saludar();
 //Destruimos el objeto.
 unset($persona);
?>
```

Y para comprobar que el objeto ha sido destruido modificaremos nuestra clase Persona.php agregando otro método mágico **__destruct()**:

```
<?php
 class Persona {
    public $nombre;
    public $apellido;
    public $edad;
    public function __construct($nombre, $apellido, $edad) {
       $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 ';
    }
 }
?>
```

Si comprueban su script verán que el método destructor se dispara en cuanto el objeto desparece de la memoria.

Bueno, hasta acá todo. Cualquier duda o sugerencia por acá la hacen.

Saludos!

Niveles de acceso

Si leíste mis dos anteriores posteos habrás notado que al definir propiedades y métodos en una clase utilice la palabra **public** al principio, y dije que lo explicaría más adelante. Bueno, esta vez trataré eso.

En la programación orientada a objetos existe algo llamado niveles de acceso. En el primer posteo, Php orientado a objetos, parte 1: Clases, propiedades y métodos, yo creaba una clase con este aspecto:

```
<?php
 class Persona {
    public $nombre;
    public $apellido;
    public $edad;
    public function saludar(){
       return 'Hola, soy ' . $this->nombre . ' ' . $this->apellido . ' y
tengo ' . $this->edad . ' años ';
    }
 }
```

Y luego al crear el objeto basado en la clase **Persona**, yo podía acceder desde dicho objeto a las propiedades **$nombre**, **$apellido** y **$edad**, pero también al método **saludar()**.

Ahora bien, yo podía hacerlo debido a que al definir la clase utilice **public** delante tanto de las propiedades como de los métodos. Sin embargo esto tienen sus desventajas, no tanto en cuanto al método **saludar()**, que es un método al cual yo necesito utilizar una vez creado el objeto, sino más bien en las propiedades.

Normalmente en la programación orientada a objetos las propiedades no suelen ser accedidas desde el objeto, ya que esto presenta un problema de seguridad, por tanto para proteger nuestras propiedades, para que solamente sean accedidas desde la clase y nunca desde el objeto se utiliza **private**.

Luego, en el siguiente posteo, Php orientado a objetos, parte 2: Método constructor y destructor, yo utilicé un método constructor para setear los valores de las propiedades al crear el objeto y no accediendo desde las mismas propiedades. La clase tenía este aspecto:

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

```
    public function __construct($nombre, $apellido, $edad) {
       $this->nombre = $nombre;
       $this->apellido = $apellido;
       $this->edad = $edad;
    }
    public function saludar(){
       return 'Hola, soy ' . $this->nombre . ' ' . $this->apellido . ' y
tengo ' . $this->edad . ' años ';
    }
 }
?>
```

Entonces para setear las propiedades yo lo hacía desde el constructor de esta forma:

```
$persona = new Persona('Fernando', 'Gaitan', 26);
```

Y me evitaba hacer esto:

```
$persona->nombre = 'Fernando';
$persona->apellido = 'Gaitan';
$persona->edad = 26;
```

Ahora bien, copiaremos la misma clase que usamos en el anterior posteo, pero cambiaremos los niveles de acceso de las propiedades de **public** a **private** de esta forma:

```
<?php
 class Persona {
    private $nombre;
    private $apellido;
    private $edad;
    public function __construct($nombre, $apellido, $edad) {
       $this->nombre = $nombre;
       $this->apellido = $apellido;
       $this->edad = $edad;
    }
    public function saludar(){
       return 'Hola, soy ' . $this->nombre . ' ' . $this->apellido . ' y
tengo ' . $this->edad . ' años ';
    }
 }
?>
```

Para probar que todo ha salido bien volveremos a hacer lo mismo de siempre:

```
<?php
 require_once 'clases/Persona.php';
 $persona = new Persona('Fernando', 'Gaitan', 26);
 echo $persona->saludar();
?>
```

Nada nuevo, ahora buscaremos problemas y romperemos nuestro script tratando de setear el valor de una propiedad desde el objeto de esta forma:

```
<?php
```

```
 require_once 'clases/Persona.php';
 $persona = new Persona('Fernando', 'Gaitan', 26);
 //Intentamos setear el valor de la propiedad privada para comprobar que
se rompe nuestro script.
 $persona->nombre = 'Cosme Fulanito';
 echo $persona->saludar();
?>
```

Si te tiró un mensaje de error como: "**Fatal error**: Cannot access private property Persona::$nombre in…" está perfecto. Las propiedades son privadas y no deben ser accedidas desde el objeto, para eso definimos su valor desde el constructor. Acordate que las propiedades deberían ser casi siempre **private**. En muy pocos casos las propiedades son **public** (sinceramente ahora no se me ocurre ningún caso)

Ahora, ¿un método también puede ser **private**? Sí, por ejemplo algún método que se dispare dentro de otros métodos, pero nunca sea necesario llamarlo desde el objeto. Recordar que tanto las propiedades como los métodos al ser **private** no pueden ser accedidos desde el objeto, sólo dentro de la clase, sino PHP nos devolverá por pantalla un error como el de recién.

En realidad hay más niveles de acceso, **protected** y **final**, pero esos los veremos más adelante.

Bueno, hasta acá termino.

Saludos!

Métodos getters y setters

Ok, en mi posteo anterior, Php orientado a objetos, parte 3: Niveles de acceso, hablamos sobre los niveles de acceso **public** y **private**, y mencioné que las propiedades deberían ser de tipo **private** para evitar que accidentalmente se acceda las mismas ya sea recuperando su valor o modificándolas.

Ahora bien, ¿qué pasa si nosotros tenemos la necesidad de recuperar el valor de una propiedad o cambiar su valor? ¿La convertimos en **public**? Podría ser una solución a simple vista, pero no, las propiedades deberían ser siempre **private**.

Para solucionar dicho problema existen los métodos **getters** y **setters**.

Los métodos **getters** son aquellos que nos devuelven los valores de las propiedades. Por lo general estos métodos no reciben parámetros y deben devolver algo, osea tener un **return**. Si bien cada uno de estos métodos suele crearse para devolver el valor de una sola propiedad hay programadores que utilizan un método que devuelve un array con los valores de las propiedades. En mi caso no hago esto último.

Cambiaremos un poco de clase, ya que hace tres posteos que vengo con la misma **Persona**. En esta ocasión crearemos una clase llamada **Producto** que tendrá cuatro propiedades: un id, un nombre, una descripción y el precio:

```
<?php
 class Producto {
    private $id;
    private $nombre;
    private $descripcion;
    private $precio;
    public function __construct($id, $nombre, $descripcion, $precio) {
       $this->id = $id;
       $this->nombre = $nombre;
       $this->descripcion = $descripcion;
       $this->precio = $precio;
    }
 }
?>
```

Nada, nuevo, propiedades y un constructor en donde se definen los mismos.

Ahora supongamos que yo necesito recuperar el nombre del producto desde el objeto, para eso dentro de la clase crearé un nuevo método con este aspecto:

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

Facíl, muy fácil. Ahora ¿qué es un método setters?

Los métodos **setters** son aquellos que permiten modificar el valor de una propiedad. Por lo general estos reciben un parámetro con el nuevo valor de la propiedad y no devuelven nada. Al igual que los **getters**, estos métodos suelen crearse un por cada propiedad a la cual es posible cambiarle el valor, sin embargo también hay programadores usan uno para modificar varias propiedades con un array de parámetro. Yo tampoco hago esto último.

Veremos cómo se utliza:

```
public function set_nombre($nombre){
   $this->nombre = $nombre;
}
```

Muy fácil. Como vemos el método recibe un parámetro que es el nuevo valor de la propiedad. Entonces, no sólo podemos recuperar el valor de nuestra propiedad, sino también modificarla.

Ahora teniendo en cuenta que ya entendemos la lógica de esto crearé los métodos **getters** y **setters** para las demás propiedades:

```
<?php
 class Producto {
    private $id;
```

```
    private $nombre;
    private $descripcion;
    private $precio;
    public function __construct($id, $nombre, $descripcion, $precio) {
       $this->id = $id;
       $this->nombre = $nombre;
       $this->descripcion = $descripcion;
       $this->precio = $precio;
    }
    public function get_nombre(){
       return $this->nombre;
    }
    public function get_descripcion(){
       return $this->descripcion;
    }
    public function get_precio(){
       return $this->precio;
    }
    public function get_id(){
       return $this->id;
    }
    public function set_nombre($nombre){
       $this->nombre = $nombre;
    }
    public function set_descripcion($descripcion){
       $this->descripcion = $descripcion;
    }
    public function set_precio($precio){
       $this->precio = $precio;
    }
 }
?>
```

Si es la primera vez que ves esto tal vez te preguntes lo mismo que yo la primera vez que lo vi: ¿Para qué c… esto? ¿No es más fácil que las propiedades sean **public**?

Bueno, la respuesta es muy simple:

Existe la necesidad de recuperar el valor del nombre, la descripción, el precio y el id del producto. También, el nombre del producto, la descripción y el precio pueden cambiar, por ejemplo si tenemos una aplicación para un negocio que vende artículos de computación y la persona que ingresa el registro se equivoca y debe modificar cosas o simplemente quiere cambiarle el precio al producto. Pero suponiendo el id del producto sea la clave primaria de una base de datos, ¿existiría la posibilidad de modificarlo? NO. Por tanto la propiedad $id tendrá un método **getters**, mas no un **setters**.

Si no usáramos métodos **getters** y **setters** y dejamos las propiedades como **public**, entonces esto estaría bien:

```
echo $producto->id;
```

Pero esto no:

```
 $producto->id = 2000;
```

Por ese motivo, en la clase que creé arriba, si te fijaste bien no hay un método **setters**, por el simple hecho de que el id no debería ser modificado.

También existe la posibilidad de que una propiedad tenga un método **setters** y no un **getters**. De todas formas, ahora sabemos que es recomendable proteger nuestras propiedades con private, y luego de acuerdo a la lógica de cada una crear métodos **getters** y **setters** según lo amerite.

Para probar el funcionamiento copiá la clase creada de Producto en un archivo con nombre **Producto.php** y luego probar esto en tu navegador:

```
<?php
 require_once 'clases/Producto.php';
 $producto = new Producto(111, 'Pendrive 8', 'Pendrive marca Kingston de
8GB', 150);
 echo 'Id: ' . $producto->get_id() . '<br />';
 echo 'Nombre: ' . $producto->get_nombre() . '<br />';
 echo 'Descripción: ' . $producto->get_descripcion() . '<br />';
 echo 'Precio: $' . $producto->get_precio() . '<br />';
 $producto->set_nombre('Pendrive 16');
 $producto->set_descripcion('Pendrive marca Kingston de 16GB');
 $producto->set_precio(300);
 echo '<hr />';
 echo 'Id: ' . $producto->get_id() . '<br />';
 echo 'Nombre: ' . $producto->get_nombre() . '<br />';
 echo 'Descripción: ' . $producto->get_descripcion() . '<br />';
 echo 'Precio: $' . $producto->get_precio() . '<br />';
?>
```

Saludos!

Constantes

Así como podemos definir propiedades que funcionan igual que variables, también podemos definir constantes dentro de nuestras clases. Vale aclarar que la diferencia de una propiedad (o variable) de una constante, es que la primera puede modificar su contenido, mientras que las constantes, como su nombre lo indica, tienen un valor constante, un valor que no puede modificarse.

Un ejemplo perfecto de una constante puede ser el valor de PI, que siempre será 3,14. En esta ocasión crearé una clase llamada Validador que tendrá un método que reciba la edad del visitante y devuelva true si es mayor de edad y false si es menor.

Para comprobar si es mayor o menor de edad debemos tener una edad como referencia que jamás cambiará su valor ni en la clase, ni en el objeto. En este caso será 18. Bueno ese valor lo guardaremos en una constante:

```
<?php
 class Validador {
    const MAYOR_DE_EDAD = 18;
    public function __construct() {
       //Constructor vacío.
    }
    public function comprobarAcceso($edad){
       if($edad < self::MAYOR_DE_EDAD){
          return false;
       }else{
          return true;
       }
    }
 }
?>
```

Dentro de la clase creamos una constante con el valor 18. Comprobá que la constante se escribe con la palabra reservada **const**, luego el nombre de la variable igual al valor. Luego creamos un método que recibe un parámetro con una edad y verifica si es mayor de edad devolviendo true o false.

Sin embargo algo importante a tener en cuenta es que las constantes dentro de la clase no se llaman con **$this->**, sino con **self**. La palabra reservada **self** hacen referencia al nombre de la clase. Hacer **self::MAYOR_DE_EDAD** sería lo mismo que hacer **Validador::MAYOR_DE_EDAD**.

Probemos su funcionamiento:

```
<?php
 require_once 'clases/Validador.php';
 $edad = 26;
 $validador = new Validador();
    if($validador->comprobarAcceso($edad)){
       echo 'Bienvenido al sitio';
    }else{
       echo 'Usted es menor de edad.';
    }
?>
```

Ahora para recuperar el valor de la constante fuera de la clase haríamos esto:

```
<?php
 require_once 'clases/Validador.php';
 echo Validador::MAYOR_DE_EDAD;
?>
```

Saludos!

Herencia

Una de las ventajas que nos da la programación orientada a objetos es, por un lado, la forma ordenada de escribir código, y por otro la reutilizacion del mismo.

Supongamos que tenemos una clase basada en un auto:

```
<?php
 class Auto {
    private $motor_encendido = false;
    private $cantidad_de_puertas;
    private $cantidad_de_ruedas;
    private $marca;
    public           function           __construct($cantidad_de_puertas,
$cantidad_de_ruedas, $marca) {
       $this->cantidad_de_puertas = $cantidad_de_puertas;
       $this->cantidad_de_ruedas = $cantidad_de_ruedas;
       $this->marca = $marca;
    }
    public function encenderMotor(){
       $this->motor_encendido = true;
    }
    public function apagarMotor(){
       $this->motor_encendido = false;
    }
    //Verifica si el motor está encendido para saber si el auto puede
arrancar o no.
    public function arrancar(){
       return $this->motor_encendido;
    }
 }
?>
```

Una clase común como las que venimos haciendo. Con cuatro propiedades que definen si el motor está encendido, la cantidad de puertas, la cantidad de ruedas y la marca del auto. En el constructor definimos las propiedades excepto si el motor está encendido o apagado, y tres métodos: uno para encender el motor, otro para apagarlo y por último un método para arrancar el auto que devuelve si el motor está apagado o encendido.

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

```
<?php
 class Moto {
    private $motor_encendido = false;
    private $cantidad_de_ruedas;
    private $marca;
    public function __construct($cantidad_de_ruedas, $marca) {
       $this->cantidad_de_ruedas = $cantidad_de_ruedas;
       $this->marca = $marca;
    }
    public function encenderMotor(){
```

```
        $this->motor_encendido = true;
    }
    public function apagarMotor(){
        $this->motor_encendido = false;
    }
    //Verifica si el motor está encendido para saber si la moto puede
arrancar o no.
    public function arrancar(){
        return $this->motor_encendido;
    }
 }
?>
```

La clase **Moto** es prácticamente idéntica a la clase **Auto**, sólo que con una mínima diferencia: la clase **Moto** no tiene una propiedad **$cantidad_de_puertas**, 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 auto y una moto? Que ambos son vehículos.

Así que entonces podríamos tener una clase **Vehiculo** también:

```
<?php
 class Vehiculo {
    protected $motor_encendido = false;
    protected $cantidad_de_puertas;
    protected $cantidad_de_ruedas;
    protected $marca;
    protected          function          __construct($cantidad_de_puertas,
$cantidad_de_ruedas, $marca) {
        $this->cantidad_de_puertas = $cantidad_de_puertas;
        $this->cantidad_de_ruedas = $cantidad_de_ruedas;
        $this->marca = $marca;
    }
    public function encenderMotor(){
        $this->motor_encendido = true;
    }
    public function apagarMotor(){
        $this->motor_encendido = false;
    }
    //Verifica si el motor está encendido para saber si el auto puede
arrancar o no.
    public function arrancar(){
        return $this->motor_encendido;
    }
 }
?>
```

Y ahora volveremos a crear nuestras clases Auto y Moto, pero heredando de esta última clase así:

```
<?php
 //Ante de definir la clase incluimos la clase padre Vehiculo.
```

```
 require_once 'Vehiculo.php';
 class Auto extends Vehiculo {
    public           function           __construct($cantidad_de_puertas,
$cantidad_de_ruedas, $marca) {
       parent::__construct($cantidad_de_puertas,     $cantidad_de_ruedas,
$marca);
    }
 }
?>

<?php
 //Ante de definir la clase incluimos la clase padre Vehiculo.
 require_once 'Vehiculo.php';
 class Moto extends Vehiculo {
    public function __construct($cantidad_de_ruedas, $marca) {
       parent::__construct(0, $cantidad_de_ruedas, $marca);
    }
 }
?>
```

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

La herencia permite que clases como **Auto** 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. Además cuando definimos la clase debemos utilizar la palabra reservada **extends** más el nombre de la clase padre:

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

Pueden verlo como lo definimos tanto en la clase **Auto**, como en **Moto**.

Otra cosa que también es interesante es que si bien una clase hija hereda los métodos de su clase madre también se pueden sobrescribir.

Por ejemplo en la clase Vehiculo nosotros tenemos un constructor cómo este:

```
protected function __construct($cantidad_de_puertas, $cantidad_de_ruedas,
$marca) {
    $this->cantidad_de_puertas = $cantidad_de_puertas;
    $this->cantidad_de_ruedas = $cantidad_de_ruedas;
    $this->marca = $marca;
 }
```

Pero en la clase Moto lo que hacemos es sobrescribir el método constructor de la clase madre:

```
public function __construct($cantidad_de_ruedas, $marca) {
    parent::__construct(0, $cantidad_de_ruedas, $marca);
 }
```

Osea lo único que estamos haciendo es que cuando creemos un objeto basado en Moto el constructor cambie y omita el parámetro de $cantidad_de_puertas, que tomo esa propiedad como 0, ya que una moto no tiene puertas.

Además 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, fijate 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 Auto y Moto no podrían acceder a las mismas.

Acá les dejo un ejemplo de cómo serían los objetos basados en ambas clases:

objeto **Auto**:

```
<?php
 require_once 'clases/Auto.php';
 $auto = new Auto(4, 4, 'Ford');
 $auto->encenderMotor();
 if($auto->arrancar()){
    echo 'El auto esta andando';
 }else{
    echo 'No se puede arrancar el auto si el motor no esta encendido';
 }
?>
```

objeto **Moto**:

```
<?php
 require_once 'clases/Moto.php';
 $moto = new Moto(2, 'Yamaha');
 $moto->encenderMotor();
 if($moto->arrancar()){
    echo 'La moto esta andando';
 }else{
    echo 'No se puede arrancar la moto si el motor no esta encendido';
 }
?>
```

Saludos!

Clases abstractas y finales

En el post pasado Php orientado a objetos, parte 6: Herencia vimos cómo hereda una clase de otra, lo que permite a la clase hija obtener las mismas propiedades y métodos de la clase madre, y así evitar escribir nuevamente las mismas. Pero hay algo que no expliqué con respecto a la herencia.

Nosotros teníamos una clase con este aspecto:

```
<?php
 class Vehiculo {
    protected $motor_encendido = false;
    protected $cantidad_de_puertas;
    protected $cantidad_de_ruedas;
    protected $marca;
    protected          function          __construct($cantidad_de_puertas,
$cantidad_de_ruedas, $marca) {
       $this->cantidad_de_puertas = $cantidad_de_puertas;
       $this->cantidad_de_ruedas = $cantidad_de_ruedas;
       $this->marca = $marca;
    }
    public function encenderMotor(){
       $this->motor_encendido = true;
    }
    public function apagarMotor(){
       $this->motor_encendido = false;
    }
    //Verifica si el motor está encendido para saber si el auto puede
arrancar o no.
    public function arrancar(){
       return $this->motor_encendido;
    }
 }
?>
```

De donde otras dos clases, **Auto** y **Moto** heredaban y obtenían todas las propiedades y métodos de esta clase **Vechiculo**. Sin embargo si nos ponemos a pensar, tener clases para crear objetos basados en un auto o una moto tiene lógica, porque estamos hablando de cosas concretas, que existen. Yo puedo decir: “tengo un auto” o “tengo una moto”, pero no puedo decir: “tengo un vehículo” o mejor dicho, debo aclarar qué vehículo es. Por tanto si bien la clase **Vehiculo** debería crearse para que otras clases hereden de ahí, no debería darse la posibilidad de crearse objetos basados en una clase **Vehiculo**.

Para ello existen las clases abstractas. Una clase abstracta no permite crear instancias de objetos basados en la misma. Para lograr esto sólo tenemos que agregar la palabra **abstract** antes de **class**, de esta forma:

```
<?php
 abstract class Vehiculo {
    protected $motor_encendido = false;
    protected $cantidad_de_puertas;
    protected $cantidad_de_ruedas;
    protected $marca;
    protected          function          __construct($cantidad_de_puertas,
$cantidad_de_ruedas, $marca) {
       $this->cantidad_de_puertas = $cantidad_de_puertas;
       $this->cantidad_de_ruedas = $cantidad_de_ruedas;
       $this->marca = $marca;
    }
    public function encenderMotor(){
       $this->motor_encendido = true;
    }
    public function apagarMotor(){
       $this->motor_encendido = false;
    }
    //Verifica si el motor está encendido para saber si el auto puede
arrancar o no.
    public function arrancar(){
       return $this->motor_encendido;
    }
 }
?>
```

De esta forma no podrán crearse objetos basados en la clase **Vehiculo**, pero sí basados en la clase **Auto** y **Moto**, debido a que estas clases no son abstractas. Si intentamos crear un objeto nos tirará un error.

En el caso de que lo que queramos es que una clase no tenga herencia, osea que ninguna clase pueda heredar sus métodos debemos agregar la palabra reservada **final**.

```
final class MiClase {}
```

Saludos!

Propiedades y métodos estáticos

En este posteo explicaré unos de los temas más importantes que tienen que ver con las clases en PHP.

Como hemos visto hasta ahora las clases nos sirven como modelos con un fin: crear objetos. Las clases tienen métodos y estos pueden invocarse desde el objeto que ha sido creado. Sin embargo, las clases no se limitan solamente a eso, servir de modelo para la creación de un objeto.

Dentro de las clases nosotros podemos definir métodos llamados métodos estáticos con la palabra reservada **static** antes del nombre del método:

```
class NombreDeLaClase {
    public static function nombreDelMetodo(){
       //
    }
}
```

Los métodos estáticos a diferencia de los otros se invocan sin crear un objeto, osea desde la misma clase a la que pertenece. De una forma similar a ésta:

```
NombreDeLaClase::nombreDelMetodo();
```

Como verán primero se llama a la clase, luego se utliza los dos puntos (**::**) y finalmente se invoca al método estático creado. Todo esto sin la necesidad de tener que instanciar un objeto basado en esa clase.

Crearé un ejemplo de una clase llamada **Date_f**, que tendrá dos métodos. Uno que nos devolverá la fecha del día, y otra que nos devolverá la hora.

```
<?php
 class Date_f {
    public static function getFecha(){
       $anio = date('Y');
       $mes = date('m');
       $dia = date('d');
       return $dia . '/' . $mes . '/' . $anio;
    }
    public static function getHora(){
       $hora = date('H');
       $minutos = date('i');
       $segundos = date('s');
       return $hora . ':' . $minutos . ':' . $segundos;
    }
 }
?>
```

Y para probarla crearemos un script donde incluiremos la clase y llamaremos a ambos métodos desde la misma. Primero probaremos con el método que devuelve la fecha:

```
<?php
 require_once 'clases/Date_f.php';
 echo 'La fecha de hoy es: ' . Date_f::getFecha();
?>
```

Luego probaremos el método que devuelve la hora:

```
<?php
 require_once 'clases/Date_f.php';
 echo 'Son las: ' . Date_f::getHora();
?>
```

A su vez dentro de un método estático puede llamarse a otro método, siempre y cuando comparte la misma característica: sea estático.

Por ejemplo yo podría crear otro método estático que devuelve la fecha y la hora juntas invocando a los otros dos:

```
<?php
 class Date_f {
    public static function getFecha(){
       $anio = date('Y');
       $mes = date('m');
       $dia = date('d');
       return $dia . '/' . $mes . '/' . $anio;
    }
    public static function getHora(){
       $hora = date('H');
       $minutos = date('i');
       $segundos = date('s');
       return $hora . ':' . $minutos . ':' . $segundos;
    }
    public static function getFechaHora(){
       $fecha = self::getFecha();
       $hora = self::getHora();
       return $fecha . ' | ' . $hora;
    }
 }
?>
```

Y luego probarlo con:

```
<?php
 require_once 'clases/Date_f.php';
 echo 'Fecha y hora: ' . Date_f::getFechaHora();
?>
```

Para invocar métodos estáticos dentro de la clase no se usa **$this->metodo()**, sino que lo hago con **self::metodo()**. Esto se debe a que **$this** sirve para hacer llamadas a propiedades y métodos que están dentro del objeto (no estáticos), y yo acá no tengo que crear ningún objeto. En realidad **self** es una palabra reservada que hace referencia al nombre de la clase donde estoy parado, como lo dije anteriormente en mi antiguo posteo, de la misma forma en que se llama a una constante: Php orientado a objetos, parte 5: Constantes.

Osea que hacer:

```
$fecha = self::getFecha();
$hora = self::getHora();
```

Es lo mismo que hacer:

```
$fecha = Date_f::getFecha();
$hora = Date_f::getHora();
```

Sólo que **self** es más recomendable por si más adelante decidimos cambiarle de nombre a la clase.

Para llamar a propiedades estáticas también debería hacerse con **self**. Para eso crearemos una propiedad estática con los días de la semana y un método que los muestre:

```
<?php
 class Date_f {
    private   static   $dias_de_la_semana   =   array('Lunes',   'Martes',
'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo');
    public static function getFecha(){
       $anio = date('Y');
       $mes = date('m');
       $dia = date('d');
       return $dia . '/' . $mes . '/' . $anio;
    }
    public static function getHora(){
       $hora = date('H');
       $minutos = date('i');
       $segundos = date('s');
       return $hora . ':' . $minutos . ':' . $segundos;
    }
    public static function getFechaHora(){
       $fecha = Date_f::getFecha();
       $hora = Date_f::getHora();
       return $fecha . ' | ' . $hora;
    }
    public static function mostrarDíasDeLaSemana(){
       $cadena_con_dias = implode(', ', self::$dias_de_la_semana);
       return $cadena_con_dias;
    }
 }
?>
```

Cómo ven creamos un nuevo método que devolverá una cadena con los días de la semana y los recupera llamando a la propiedad estática **$dias_de_la_semana**:

```
<?php
 require_once 'clases/Date_f.php';
 echo 'Los días de la semana son: ' . Date_f::mostrarDíasDeLaSemana();
?>
```

Otra cosa que vale aclarar es que dentro de un método estático no se puede invocar un método que no lo es, ya que estos sólo están disponibles en el momento en que se crea un objeto. Sí en cambio podemos retornar desde un método estático un objeto basado en la clase. Por ejemplo nosotros podemos tener una clase con este aspecto:

```
<?php
 class Persona {
    private $nombre;
    private $apellido;
    private $edad;
    public function __construct($nombre, $apellido, $edad) {
       $this->nombre = $nombre;
```

```
        $this->apellido = $apellido;
        $this->edad = $edad;
    }
    public function saludar(){
        return 'Hola, soy ' . $this->nombre . ' ' . $this->apellido . ' y
tengo ' . $this->edad . ' años ';
    }
    public static function getUsuarioPorDefecto(){
        return new self('Fernando', 'Gaitan', 26);
    }
 }
?>
```

Pero en lugar de utilizar el constructor para crear el objeto, creamos instancia con un método estático de esta forma:

```
<?php
 require_once 'clases/Persona.php';
 $persona = Persona::getUsuarioPorDefecto();
 echo $persona->saludar();
?>
```

Al trabajar en forma independiente de un objeto los métodos estáticos se caracterizan por ser similares a las funciones, pero la ventaja que tiene sobre ésta es que cada método estáticos está encapsulada en una clase. Esto nos permite trabajar en forma más ordenada que la programación estructurada, en donde a medida que el proyecto iba haciéndose más grande y terminábamos teniendo millones de funciones esperando ser invocadas.

Excepciones

En otros lenguajes de programación suele utilizarse mucho las excepciones, algo interesante y bastante útil a la hora de controlar el flujo de nuestros proyectos que puede llegar a romperse por causa de errores.

PHP 5 nos da la posibilidad de crear nuestras propias excepciones con la clase **Exception** y las sentencias **try** y **catch**. Si nunca has visto las sentencias **try catch**, la sintáxis es más o menos así:

```
try{
   //Ejecuta código que puede llegar a tener error.
 }catch(Exception $e){
   //Recupera un error de haberlo en lo anterior.
 }
```

Vamos a ver un ejemplo bastante sencillo. Supongamos que tenemos una función que recibe dos números y nos devuelve el valor de la suma de ambos:

```
<?php
```

```
function sumar($numero1, $numero2){
   return $numero1 + $numero2;
}
?>
```

Luego la llamaríamos con:

```
<?php
echo sumar(15, 5);
?>
```

Ahora bien, acá surge un posible problema y es que los valores ingresados podrían llegar a no ser numéricos, lo que provocaría que PHP nos lance un mensaje de error.

Para solucionar esto deberíamos abrir el paraguas de antemano y crear dentro de la función lo que se llama una excepción. La función entonces quedaría de la siguiente forma:

```
<?php
 function sumar($numero1, $numero2){
   //Verifico si ambos valores ingresados son numéricos.
   if(is_numeric($numero1) and is_numeric($numero2)){
      return $numero1 + $numero2;
   }else{
      //Crea una excepción.
      throw new Exception('Los valores ingresados no son numéricos');
      return 0;
   }
 }
?>
```

Cómo verán dentro de la función yo utilizo un **if** que verifica si ambos valores ingresados son numéricos con la función de PHP **is_numeric()**, que recibirá un parámetro y de ser númerico me devolverá true. Si ambos dan true, osea que son numéricos se realizará la suma así retornando el valor de la misma, pero de lo contrario creará una nueva excepción como se ve en el ejemplo:

```
throw new Exception('Mensaje de error');
```

Ahora probaremos ejecutar el código:

```
<?php
 function sumar($numero1, $numero2){
    //Verifico si ambos valores ingresados son numéricos.
    if(is_numeric($numero1) and is_numeric($numero2)){
       return $numero1 + $numero2;
    }else{
       //Crea una excepción.
       throw new Exception('Los valores ingresados no son numéricos');
       return 0;
    }
 }
 try{
    //Ejecutamos la función con números.
```

```
    echo sumar(15, 5);
 }catch(Exception $e){
    echo $e->getMessage();
 }
?>
```

El bloque de código devuelve por pantalla: “20”. Lo que significa que la función se ejecutó sin problemas y por eso no capturó ningún error. Pero probemos con esto:

```
<?php
 function sumar($numero1, $numero2){
   //Verifico si ambos valores ingresados son numéricos.
   if(is_numeric($numero1) and is_numeric($numero2)){
      return $numero1 + $numero2;
   }else{
      //Crea una excepción.
      throw new Exception('Los valores ingresados no son numéricos');
      return 0;
   }
 }
 try{
   //Ejecutamos la función con parámetros incorrectos
   echo sumar('Saraza', 'EAEA');
 }catch(Exception $e){
   echo $e->getMessage();
 }
?>
```

En este últimos caso llamamos a la función **sumar()**, pero con parámetros que son incorrectos. Lo ejecutamos desde el bloque **try**, pero la misma dispara un error que es capturado y mostrado por pantalla desde el bloque **catch**. **$e** en realidad es un objeto que se crea dentro de **catch** (si se produce el error)

Los métodos disponibles para dicho objetos son:

getMessage() — Obtiene el mensaje de Excepción
getPrevious() — Devuelve la excepción anterior
getCode() — Obtiene el código de Excepción
getFile() — Obtiene el fichero en el que ocurrió la excepción
getLine() — Obtiene la línea en donde ocurrió la excepción
getTrace() — Obtiene el seguimiento de la pila
getTraceAsString() — Obtiene el stack trace como cadena

(Fuente: https://php.net/manual/es/class.exception.php)

El ejemplo mencionado en este post es muy útil para saber cómo trabajan las excepciones, sin embargo este tipo de cosas como recibir parámetros erróneos no es la forma correcta que requiera un manejo de errores. Lo correcto en este caso hubiese sido validar previamente que los valores ingresados sean numéricos antes de llamar a la función y de ser incorrectos redireccionar al usuario a una página de error de validación.

Otra cosa que es conveniente aclarar es que el manejo de errores tampoco debería ser utilizado para controlar errores del programador, sino errores posibles que el programador no puede controlar. Un ejemplo perfecto es intentarse conectar a una base de datos que no está funcionando. El programador escribe el código para conectarse a la base de datos, el código no contendrá errores, pero si la base de datos en un futuro no funciona el script se romperá.

En el próximo posteo veremos esto:

Saludos!

PDO, Conectarse a una base de datos

Un tema muy importante que aun no he tocado en estos posteos son las conexiones y consultas a una base de datos. Me he reservado este posteo para explicar cómo conectarse a una base de datos mediante **PDO**, una clase de PHP que permite crear objetos de conexión a una base de datos.

Para empezar crearemos una base de datos llamada 'batman', yo la crearé con código pero ustedes pueden hacerlo como quieran:

```
CREATE DATABASE batman;
```

Ahora que ya tenemos nuestra base de datos podemos comenzar intentando conectar PHP con la misma mediante un objeto PDO. Para ello necesitaremos 5 datos: El nombre de la base de datos con la que nos vamos a conectar, el servidor, el nombre de usuario y su contraseña. Pero también debemos indicar a qué tipo de base de datos nos conectaremos, en mi caso mysql, aunque también podría ser otra como por ejemplo oracle. Probemos conectarnos de esta forma:

```
<?php
 $pdo = new PDO('mysql:host=localhost;dbname=batman', 'root', '');
?>
```

Por empezar, la clase PDO no necesita ser incluida, la misma está disponible desde la versión 5 de PHP y existe en memoria. Al crear el objeto necesitará tres parámetros.

El primer parámetro será una cadena donde tendré que ingresar el tipo de base de datos, mysql por ejemplo; el host o servidor, acá será localhost o 127.0.0.1 y el nombre de la base de datos, en este caso 'batman'. También puede agregarse el puerto, separado por punto y coma.

El segundo parámetro es el nombre del usuario y el tercero la contraseña. En mi caso el nombre de usuario es 'root' y la contraseña es una cadena vacía.

Existe un cuarto parámetro no obligatorio que es un array con opciones para la conexión, pero en este posteo lo omitiré.

Bueno, ahora siguiendo con el código, voy a aclarar algo que está faltando. Como dije en el posteo anterior, Php orientado a objetos, parte 9: Excepciones, hay ocasiones en que nuestro código puede lanzar errores aunque el código esté correcto. Por ejemplo, nosotros subimos nuestro sitio a un hosting y de pronto la base de datos deja de funcionar, algún dato cambia de nombre, etcétera. Para solucionar este problema usaremos la sentencia **try catch**, y de esta manera tendremos un control por si nuestro script pudiese tener problemas al conectarse con la base de datos:

```
<?php
 try{
    $pdo = new PDO('mysql:host=localhost;dbname=batman', 'root', '');
 }catch(PDOException $e){
    echo 'Error al conectarse con la base de datos: ' . $e->getMessage();
    exit;
 }
?>
```

Para probar el error pueden por ejemplo cambiar algún dato, como en el nombre de la base de datos poniendo un nombre que no existe, o apagar la base de datos y comprobarlo.

Ahora que ya sabemos cómo usar la clase PDO para conectarse a una base podríamos crear una clase propia que herede todos los métodos de la misma, pero que incluya dentro los datos necesarios para conectarse. De esta manera no tendríamos que ingresar todos los datos cada vez que en nuestro proyecto quisiéramos conectarnos a la base de datos. Creemos una clase llamada **Conexion**:

```
<?php
 class Conexion extends PDO {
   private $tipo_de_base = 'mysql';
   private $host = 'localhost';
   private $nombre_de_base = 'batman';
   private $usuario = 'root';
   private $contrasena = '';
   public function __construct() {
      //Sobreescribo el método constructor de la clase PDO.
      try{
                parent::__construct("{$this->tipo_de_base}:dbname={$this-
>nombre_de_base};host={$this->host};charset=utf8", $this->usuario, $this-
>contrasena);
      }catch(PDOException $e){
          echo 'Ha surgido un error y no se puede conectar a la base de
datos. Detalle: ' . $e->getMessage();
         exit;
      }
   }
 }
?>
```

Cómo ven la clase es muy corta, hereda de PDO, y sobrescribe el constructor padre ingresando los datos para conectarse que están guardados en propiedades.

Entonces nosotros podemos conectarnos con:

```
<?php
 require_once 'clases/Conexion.php';
 $conexion = new Conexion();
?>
```

Ok, hasta acá todo. El próximo post publicaré cómo realizar consultas a una base de datos mediante métodos de la clase **PDO**.

Chau Chau Chau!

PDO, Guardar registros en una base de datos

Bueno, siguiendo con el tema de mi anterior posteo, Php orientado a objetos, parte 10: PDO, Conectarse a una base de datos, hoy les mostraré cómo guardar registros en una base de datos.

En primer lugar, seguiré con la misma base de datos de la vez pasada, si no viste el anterior posteo había creado una base de datos con el nombre **batman**, de esta forma:

```
CREATE DATABASE batman;
```

Ahora crearé una tabla para guardar los personajes de esta serie:

```
CREATE TABLE personaje(
 id tinyint(3) unsigned not null auto_increment primary key,
 nombre varchar(255) not null,
 descripcion text not null
)
```

Bueno, hasta acá nada nuevo, creé una base de datos con una sola tabla que guardará los nombres y una descripción de los personajes, y por supuesto tendrá un id o clave primaria.

Ahora, para probar mis consultas crearé dos clases. Una será una clase para conectarse a la base de datos, la misma que usé en el posteo anterior:

```
<?php
 class Conexion extends PDO {
   private $tipo_de_base = 'mysql';
   private $host = 'localhost';
   private $nombre_de_base = 'batman';
```

```
   private $usuario = 'root';
   private $contrasena = '';
   public function __construct() {
      //Sobreescribo el método constructor de la clase PDO.
      try{
                parent::__construct("{$this->tipo_de_base}:dbname={$this-
>nombre_de_base};host={$this->host};charset=utf8", $this->usuario, $this-
>contrasena);
      }catch(PDOException $e){
          echo 'Ha surgido un error y no se puede conectar a la base de
datos. Detalle: ' . $e->getMessage();
         exit;
      }
   }
 }
?>
```

Y luego crearé una segunda clase que se encargará de todo lo referido con consultas con la tabla de personajes que se llame **Personaje**, en la misma carpeta de la clase **Conexion**. Pero antes de mostrarles el código paremos aquí.

Primero pensemos en qué propiedades va a necesitar nuestra clase **Personaje**. Por cada registro necesitará guardar el id, el nombre y la descripción del personaje, pero también para realizar consultas necesitará a qué tabla apuntará. Así que crearemos tres propiedades para guardar los datos de cada personaje y una constante donde guardará el nombre de la tabla. Probemos de esta forma:

```
<?php
 require_once 'Conexion.php';
 class Personaje {
    private $id;
    private $nombre;
    private $descripcion;
    const TABLA = 'personaje';
    public function __construct($nombre, $descipcion, $id=null) {
       $this->nombre = $nombre;
       $this->descripcion = $descipcion;
       $this->id = $id;
    }
 }
?>
```

Ok, echemos un vistazo a esta clase. En primer lugar, importé la clase **Conexion**, ya que más adelante tendré que hacer consultas a la base y necesitaré mucho de la misma. Luego creé tres propiedades para guardar los datos de cada personaje y una constante con el nombre de la tabla. Y por último creé un constructor donde tendré que setear el valor de las propiedades. Si se han fijado bien, el último parámetro, **$id**, es un parámetro que por defecto es **null**, esto significa que en el momento de crear el constructor si yo no ingreso ese último parámetro tomará ese valor, nulo. Esto se debe a que no siempre el id tendrá un valor definido, por ejemplo, si yo estoy creando un nuevo usuario no ingresaré el valor del id ya que eso será auto incrementable y el valor lo recuperaré una vez insertado el registro.

Continuemos, necesitaremos un método que nos permita insertar y modificar datos en la tabla **personaje**, para ello crearemos un método llamado **guardar()**, que nos servirá para insertar y modificar registros. Para ello crearemos un objeto de la clase **Conexion** que por medio de los métodos que heredó de la clase **PDO** podrá realizar esto. La clase **Personaje** ahora quedaría así:

```
<?php
 require_once 'Conexion.php';
 class Personaje {
   private $id;
   private $nombre;
   private $descripcion;
   const TABLA = 'personaje';
   public function getId() {
      return $this->id;
   }
   public function getNombre() {
      return $this->nombre;
   }
   public function getDescripcion() {
      return $this->descripcion;
   }
   public function setNombre($nombre) {
      $this->nombre = $nombre;
   }
   public function setDescripcion($descripcion) {
      $this->descripcion = $descripcion;
   }
   public function __construct($nombre, $descipcion, $id=null) {
      $this->nombre = $nombre;
      $this->descripcion = $descipcion;
      $this->id = $id;
   }
   public function guardar(){
      $conexion = new Conexion();
      if($this->id) /*Modifica*/ {
          $consulta = $conexion->prepare('UPDATE ' . self::TABLA .' SET
nombre = :nombre, descripcion = :descripcion WHERE id = :id');
         $consulta->bindParam(':nombre', $this->nombre);
         $consulta->bindParam(':descripcion', $this->descripcion);
         $consulta->bindParam(':id', $this->id);
         $consulta->execute();
      }else /*Inserta*/ {
          $consulta = $conexion->prepare('INSERT INTO ' . self::TABLA .'
(nombre, descripcion) VALUES(:nombre, :descripcion)');
         $consulta->bindParam(':nombre', $this->nombre);
         $consulta->bindParam(':descripcion', $this->descripcion);
         $consulta->execute();
         $this->id = $conexion->lastInsertId();
      }
      $conexion = null;
   }
 }
?>
```

Ok, analicemos los cambios. En primer lugar agregué los métodos getters y setters que me estaban faltando, luego creé un método **guardar()** que primero crea un objeto **Conexion**, para así luego por medio de ese objeto poder conectarse a la base de datos:

```
$conexion = new Conexion();
```

Luego pregunté por el valor del id mediante un **if**, si el valor es cualquier cosa que no sea 0, **false** o **null** entonces entrará en el lado verdadero del condicional. Esto sirve para indicarle a nuestro método si tendrá que hacer un **update** o un **insert**, si el valor del id es **algo** significa existe un id, por ende se está modificar un usuario con ese valor, de lo contrario se está intentando insertar un usuario nuevo.

Analicemos el lado verdadero del condicional, osea el **update**:

Primero escribimos la consulta que modificará los datos del usuario:

```
$consulta = $conexion->prepare('UPDATE ' . self::TABLA .' SET nombre =
:nombre, descripcion = :descripcion WHERE id = :id');
```

Como ven creamos un nuevo objeto **$consulta**, que nos devuelve el método **prepare()**. Aquí es donde se define la consulta que se está haciendo en la base.

Luego por medio del método **bindParam()** agregamos los valores de la consulta, por ejemplo **:nombre** será el valor de la propiedad **$nombre**:

```
$consulta->bindParam(':nombre', $this->nombre);
$consulta->bindParam(':descripcion', $this->descripcion);
$consulta->bindParam(':id', $this->id);
```

Y finalmente ejecutamos la consulta:

```
$consulta->execute();
```

El lado falso del **if**, osea el que sirve para insertar un nuevo registro es muy parecido al del update, sólo cambia la consulta (obviamente) y que no debe pasarle el valor del id, porque como dijimos antes, en el **INSERT** no hay id, sino que se auto incrementa en la consulta.

Por eso al finalizar de ejecutar la consulta se recupera el id que se acaba de insertar:

```
$this->id = $conexion->lastInsertId();
```

El método **lastInsertId()** me permite recuperar la última clave primaria o id insertada en la base de datos, es un método de la clase **PDO**.

Finalmente igualo el objeto **$conexion** a **null** para cerrar la conexión:

```
$conexion = null;
```

Ahora para comprobar que todo ha salido bien probaremos crear un objeto **Personaje** para insertar un registro en la base de datos:

```
<?php
 require_once 'clases/Personaje.php';
 $persona = new Personaje('Batman', 'Encapuchado, disfrazado de murciélago que sale por las noches a combatir el mal.', 1);
 $persona->guardar();
 echo $persona->getNombre() . ' se ha guardado correctamente con el id: ' . $persona->getId();
?>
```

Al no ingresarle en el constructor el tercer parámetro no obligatorio **$id** el método **guardar()** interpretará que se está intentado insertar un nuevo registro. Prueben ejecutar este script y verán que se ha insertado un nuevo registro en la tabla **personaje**.

En la próxima ocasión veremos cómo hacer **select** en la base de datos.

Saludos!

PDO, Buscar registros en una base de datos

Bueno, continuando con el posteo anterior que publiqué hace poco, Php orientado a objetos, parte 11: PDO, Guardar registros en una base de datos, hoy vamos a aprender como hacer un select mediante **PDO**.

Repasando un poco lo que vimos, primero creamos una base de datos:

```
CREATE DATABASE batman;
```

Luego hicimos una tabla para guardar los personajes de la saga Batman:

```
CREATE TABLE personaje(
 id tinyint(3) unsigned not null auto_increment primary key,
 nombre varchar(255) not null,
 descripcion text not null
)
```

Para trabajar con la misma utilizamos dos clases: La clase **Conexion** que hereda de **PDO** y permite conectarse a la base de datos y la clase **Personaje**, con todo lo relacionado a la tabla **personaje**, y que además usa la clase **Conexion** para hacer consultas con la base de datos.

Conexion:

```
<?php
 class Conexion extends PDO {
   private $tipo_de_base = 'mysql';
```

```
   private $host = 'localhost';
   private $nombre_de_base = 'batman';
   private $usuario = 'root';
   private $contrasena = '';
   public function __construct() {
      //Sobreescribo el método constructor de la clase PDO.
      try{
                parent::__construct("{$this->tipo_de_base}:dbname={$this-
>nombre_de_base};host={$this->host};charset=utf8", $this->usuario, $this-
>contrasena);
      }catch(PDOException $e){
          echo 'Ha surgido un error y no se puede conectar a la base de
datos. Detalle: ' . $e->getMessage();
         exit;
      }
   }
 }
?>
```

Personaje:

```
<?php
 require_once 'Conexion.php';
 class Personaje {
   private $id;
   private $nombre;
   private $descripcion;
   const TABLA = 'personaje';
   public function getId() {
      return $this->id;
   }
   public function getNombre() {
      return $this->nombre;
   }
   public function getDescripcion() {
      return $this->descripcion;
   }
   public function setNombre($nombre) {
      $this->nombre = $nombre;
   }
   public function setDescripcion($descripcion) {
      $this->descripcion = $descripcion;
   }
   public function __construct($nombre, $descipcion, $id=null) {
      $this->nombre = $nombre;
      $this->descripcion = $descipcion;
      $this->id = $id;
   }
   public function guardar(){
      $conexion = new Conexion();
      if($this->id) /*Modifica*/ {
          $consulta = $conexion->prepare('UPDATE ' . self::TABLA .' SET
nombre = :nombre, descripcion = :descripcion WHERE id = :id');
         $consulta->bindParam(':nombre', $this->nombre);
         $consulta->bindParam(':descripcion', $this->descripcion);
         $consulta->bindParam(':id', $this->id);
```

```
            $consulta->execute();
        }else /*Inserta*/ {
            $consulta = $conexion->prepare('INSERT INTO ' . self::TABLA .'
(nombre, descripcion) VALUES(:nombre, :descripcion)');
            $consulta->bindParam(':nombre', $this->nombre);
            $consulta->bindParam(':descripcion', $this->descripcion);
            $consulta->execute();
            $this->id = $conexion->lastInsertId();
        }
        $conexion = null;
    }
 }
?>
```

Esta última clase nos permitía crear un objeto con los datos del personaje y luego por medio de un método **guardar()** podíamos insertar o modificar un registro dependiendo de si le pasábamos un id o no:

```
<?php
 require_once 'clases/Personaje.php';
 $persona = new Personaje('Batman', 'Encapuchado, disfrazado de
murciélago que sale por las noches a combatir el mal.');
 $persona->guardar();
 echo $persona->getNombre() . ' se ha guardado correctamente con el id: '
. $persona->getId();
?>
```

Con este código el método **guardar()** insertaba un nuevo registro ya que nosotros no le pasamos ningún id, si no existe id significa que el registro es nuevo, pero si la propiedad **$id** no está vacía intentará modificar el registro con dicho valor.

Ahora crearemos un nuevo método que buscará un registro en la base de datos con un id específico con un nuevo método, **buscarPorId()**:

```
<?php
 require_once 'Conexion.php';
 class Personaje {
    private $id;
    private $nombre;
    private $descripcion;
    const TABLA = 'personaje';
    public function getId() {
        return $this->id;
    }
    public function getNombre() {
        return $this->nombre;
    }
    public function getDescripcion() {
        return $this->descripcion;
    }
    public function setNombre($nombre) {
        $this->nombre = $nombre;
    }
    public function setDescripcion($descripcion) {
```

```
        $this->descripcion = $descripcion;
    }
    public function __construct($nombre, $descipcion, $id=null) {
        $this->nombre = $nombre;
        $this->descripcion = $descipcion;
        $this->id = $id;
    }
    public function guardar(){
        $conexion = new Conexion();
        if($this->id) /*Modifica*/ {
            $consulta = $conexion->prepare('UPDATE ' . self::TABLA .' SET
nombre = :nombre, descripcion = :descripcion WHERE id = :id');
            $consulta->bindParam(':nombre', $this->nombre);
            $consulta->bindParam(':descripcion', $this->descripcion);
            $consulta->bindParam(':id', $this->id);
            $consulta->execute();
        }else /*Inserta*/ {
            $consulta = $conexion->prepare('INSERT INTO ' . self::TABLA .'
(nombre, descripcion) VALUES(:nombre, :descripcion)');
            $consulta->bindParam(':nombre', $this->nombre);
            $consulta->bindParam(':descripcion', $this->descripcion);
            $consulta->execute();
            $this->id = $conexion->lastInsertId();
        }
        $conexion = null;
    }
    public static function buscarPorId($id){
        $conexion = new Conexion();
        $consulta = $conexion->prepare('SELECT nombre, descripcion FROM '
. self::TABLA . ' WHERE id = :id');
        $consulta->bindParam(':id', $id);
        $consulta->execute();
        $registro = $consulta->fetch();
        if($registro){
            return new self($registro['nombre'], $registro['descripcion'],
$id);
        }else{
            return false;
        }
    }
 }
?>
```

El método buscará un personaje por su id, si lo encuentra devolverá un objeto con sus datos guardados en las propiedades correspondientes, pero sino devolverá **false**.

Ahora analicemos el código.

Creamos una nueva conexión para realizar consultas, nada nuevo:

```
$conexion = new Conexion();
```

Escribimos la consulta que buscará un personaje por su id:

```
$consulta = $conexion->prepare('SELECT nombre, descripcion FROM ' . self::TABLA . ' WHERE id = :id');
```

Agregamos los valores a la consulta:

```
$consulta->bindParam(':id', $id);
```

La ejecutamos:

```
$consulta->execute();
```

En la siguiente línea lo que hacemos es guardar la respuesta de la consulta mediante un método llamado **fetch()**:

```
$registro = $consulta->fetch();
```

Lo que hace el método **fetch()** es devolvernos un array asociativo con los datos que le pedimos, en caso de encontrar un resultado (**fetch()** se usa para consultas que deberían devolver un solo registro) Pero en caso de no encontrar nada nos devolverá **false**.

Preguntamos si nos devolvió algún resultado. Si así fue retornamos un objeto **Personaje** con las propiedades **$id**, **$nombre** y **$descripcion** cargadas con sus valores correspondientes. Pero si la consulta no encontró nada con ese id nos devolverá **false**, osea nada.

```
if($registro){
   return new self($registro['nombre'], $registro['descripcion'], $id);
 }else{
   return false;
 }
```

Intenten probarlo con un id que exista en la base de datos. En mi caso el id 1 es Batman:

```
<?php
 require_once 'clases/Personaje.php';
 $personaje = Personaje::buscarPorId(1);
 if($personaje){
    echo $personaje->getNombre();
    echo '<br />';
    echo $personaje->getDescripcion();
 }else{
    echo 'El personaje no ha podido ser encontrado';
 }
?>
```

Y ahora que tenemos un objeto creado y recuperado por su id, podríamos cambiarle por ejemplo la descripción mediante el método **setDescripcion()** y probar si lo modifica correctamente con el método **guardar()**:

```
<?php
 require_once 'clases/Personaje.php';
 $personaje = Personaje::buscarPorId(1);
```

```
 if($personaje){
    $personaje->setDescripcion('En realidad es el millonario Bruno
Diaz');
    $personaje->guardar();
    echo 'El personaje ha sido modificado';
 }else{
    echo 'El personaje no ha podido ser encontrado';
 }
?>
```

Bueno, ahora, si quisiéramos por ejemplo recuperar una lista, en este caso de varios personajes, deberíamos usar el método **fetchAll()**. Este método a diferencia de **fetch()** no nos devolverá un array asociativo sino que nos devolverá un array de arrays asociativos.

Ahora insertaremos varios registros a mano para poder probarlo rápidamente esto:

```
INSERT INTO personaje(nombre, descripcion)
VALUES
('Robin', 'El compañero de Batman'),
('El Guasón', 'El que se ríe'),
('Dos Caras', 'El que tiene dos caras'),
('El Acertijo', 'El de verde con un signo de preguntas'),
('Alfred', 'El mayordomo'),
('Bane', 'El que usa máscara');
```

El nuevo método para recuperar todos los personajes que se llamará **recuperarTodos()**. La clase ahora quedará así:

```
<?php
 require_once 'Conexion.php';
 class Personaje {
    private $id;
    private $nombre;
    private $descripcion;
    const TABLA = 'personaje';
    public function getId() {
       return $this->id;
    }
    public function getNombre() {
       return $this->nombre;
    }
    public function getDescripcion() {
       return $this->descripcion;
    }
    public function setNombre($nombre) {
       $this->nombre = $nombre;
    }
    public function setDescripcion($descripcion) {
       $this->descripcion = $descripcion;
    }
    public function __construct($nombre, $descipcion, $id=null) {
       $this->nombre = $nombre;
       $this->descripcion = $descipcion;
       $this->id = $id;
    }
```

```
    public function guardar(){
       $conexion = new Conexion();
       if($this->id) /*Modifica*/ {
          $consulta = $conexion->prepare('UPDATE ' . self::TABLA .' SET
nombre = :nombre, descripcion = :descripcion WHERE id = :id');
          $consulta->bindParam(':nombre', $this->nombre);
          $consulta->bindParam(':descripcion', $this->descripcion);
          $consulta->bindParam(':id', $this->id);
          $consulta->execute();
       }else /*Inserta*/ {
          $consulta = $conexion->prepare('INSERT INTO ' . self::TABLA .'
(nombre, descripcion) VALUES(:nombre, :descripcion)');
          $consulta->bindParam(':nombre', $this->nombre);
          $consulta->bindParam(':descripcion', $this->descripcion);
          $consulta->execute();
          $this->id = $conexion->lastInsertId();
       }
       $conexion = null;
    }
    public static function buscarPorId($id){
       $conexion = new Conexion();
       $consulta = $conexion->prepare('SELECT nombre, descripcion FROM '
. self::TABLA . ' WHERE id = :id');
       $consulta->bindParam(':id', $id);
       $consulta->execute();
       $registro = $consulta->fetch();
       if($registro){
          return new self($registro['nombre'], $registro['descripcion'],
$id);
       }else{
          return false;
       }
    }
    public static function recuperarTodos(){
       $conexion = new Conexion();
       $consulta = $conexion->prepare('SELECT id, nombre, descripcion
FROM ' . self::TABLA . ' ORDER BY nombre');
       $consulta->execute();
       $registros = $consulta->fetchAll();
       return $registros;
    }
 }
?>
```

Analicemos el código del método **recuperarTodos()**:

Conectamos a la base de datos:

```
$conexion = new Conexion();
```

Creamos la consulta:

```
$consulta = $conexion->prepare('SELECT id, nombre, descripcion FROM ' .
self::TABLA . ' ORDER BY nombre');
```

La ejecutamos:

```
$consulta->execute();
```

Recuperamos la respuesta de la consulta y la retornamos:

```
$registros = $consulta->fetchAll();
return $registros;
```

Ahora vemos cómo funciona:

```
<?php
 require_once 'clases/Personaje.php';
 $personajes = Personaje::recuperarTodos();
?>
<html>
   <head></head>
   <body>
      <ul>
      <?php foreach($personajes as $item): ?>
      <li> <?php echo $item['nombre'] . ' - ' . $item['descripcion']; ?>
</li>
      <?php endforeach; ?>
      </ul>
   </body>
</html>
```

Simplemente guardamos en la variable **$personajes** el resultado de la consulta y la mostramos en pantalla con código html.

Bueno con esto termino, más adelante espero publicar más información sobre clases en PHP.

Les dejo un ejemplo de listar, crear, modificar y eliminar registros con PDO:

Descargar ejemplo

Les recomiendo esta página: https://www.phpclasses.org/ hay una comunidad muy grande que sube clases para realizar distintas cosas por ejemplo crear archivos PDF y cosas así.

Saludos!

Anterior: Php orientado a objetos, parte 11: PDO, Guardar registros en una base de datos

Siguiente: Php orientado a objetos, parte 13: Singleton

Singleton

Ok, hace ya casi tres años o más, inicié esta serie de publicaciones de PHP, y lo he terminado hace bastante tiempo también, sin embargo retomo con una publicación más, con una duda que me compartió un chico hace unos meses.

Todo partía desde el inicio de una conexión a la base de datos, ¿qué pasa cuando debemos hacer varias consultas, y necesitamos un objeto conexión, pero uno solo que podamos reutilizar, sin necesidad de crear nuevas instancias?

Para lograr esta solución debemos recurrir a un patrón muy común en los lenguajes que manejan objetos: singleton.

¿Pero qué es esto? Bueno, básicamente un singleton, es una clase que sólo tendrá una única instancia. Para ello, entonces lo primero es crear privado el constructor:

```
class MiSingleton {
   private function __construct(){
      //
   }
}
```

Aquí si intentamos crear una instancia de dicha clase nos devolverá un Fatal error.

Sin embargo, esto no tiene mucha utilidad, ya que no tendremos más de una instancia, pero tampoco tendremos una sola, para eso debemos crear un método que va a verificar si ya hay una instancia creada, de haberlo nos devolverá ese único objeto y de lo contrario, creará una.

La clase quedaría:

```
class MiSingleton {
   private static $instancia = null;
   private function __construct(){
      //
   }
   public static function getInstancia(){
      if(!self::$instancia){
         self::$instancia = new self();
      }
      return self::$instancia;
   }
 }
```

Y para crear una instancia de este objeto:

```
$instancia = MiSingleton::getInstancia();
```

Podemos probar si esto es así, que se crea una sólo instancia y realizamos la siguiente prueba:

```
class MiSingleton {
   private static $instancia = null;
```

```
   private function __construct(){
      //
   }
   public static function getInstancia(){
      if(!self::$instancia){
         echo 'No existe una instancia, así que la creamos. <br />';
         self::$instancia = new self();
      }else{
         echo 'Ya hay una instancia creada. <br />';
      }
      return self::$instancia;
   }
 }
```

Y a continuación:

```
$instancia = MiSingleton::getInstancia();
$instancia = MiSingleton::getInstancia();
$instancia = MiSingleton::getInstancia();
```

La primera vez que llamamos al método **getInstancia()**, como no existe un objeto creado, entonces se creará uno nuevo, y la segunda y tercera nos devolverá el objeto creado en la primera.

Bien, ahora que ya sabemos cómo funciona una clase singleton, podemos retomar el tema con el que iniciamos. Cuando vamos a usar una base de datos, podemos tener una sólo instancia, no es necesario, como hicimos en las publicaciones pasadas, crear una nueva cada vez que generamos una nueva instancia. Por tanto podemos modificar la clase Conexión de la siguiente manera:

```
<?php
if (!defined('CONTROLADOR'))
   exit;
class Conexion {
   private $tipo_de_base = 'mysql';
   private $host = 'localhost';
   private $nombre_de_base = 'batman';
   private $usuario = 'root';
   private $contrasena = '';
   private static $instancia = null;
   private function __construct() {
      try {
         self::$instancia = new PDO("{$this->tipo_de_base}:dbname={$this-
>nombre_de_base};host={$this->host};charset=utf8", $this->usuario, $this-
>contrasena);
      } catch (PDOException $e) {
         echo 'Ha surgido un error y no se puede conectar a la base de
datos. Detalle: ' . $e->getMessage();
         exit;
      }
   }
   public static function getInstancia(){
      if(!self::$instancia){
         new self();
```

```
        }
        return self::$instancia;
    }
    public static function cerrar(){
        self::$instancia = null;
    }
}
```

Además de agregar el método **getInstancia()**, también agregamos uno llamado **cerrar()**, que dejará en **null** esa instancia única, en criollo, va a cerrar la base de datos.

A continuación dejo el ejemplo completo:

Descargar ejemplo
