Programación Orientada a Objetos en PHP

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.

Para crear un objeto en PHP se necesita tener una clase. Una clase es como un molde, un modelo. Algo que defina la estructura del objeto. Por ejemplo si tuviéramos 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, los aprendices deberán crear 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'. 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
<?php
 class Persona {
    //Acá dentro va el código
 }
?>
```

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

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

Las propiedades en PHP se escriben igual que una variable, con el signo $ adelante y el nombre.

Ahora ya que tenemos nuestra clase creada, iremos a nuestro archivo **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 />';
?>
```

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**.

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.

A continuación se muestra 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 ';
    }
 }
?>
```

También creamos un método **saludar()** que tiene la misma estructura que una función, sólo que con la palabra reservada **public**. 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
<?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();
?>
```

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

Y ahora probemos su funcionamiento de la siguiente manera:

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

Ejecutar el archivo **test.php** y comprobar su funcionamiento. 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();
?>
```

Ahora que sabemos de qué 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.

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 útil 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 tenemos la necesidad de recuperar el valor de una propiedad o de cambiar sus valores existen los métodos **getters** y **setters**, teniendo en cuenta que por la propiedad de encapsulación en POO, los atributos de las clases deben ser *privados.*

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**. La siguiente clase llamada **Producto** tendrá cuatro propiedades: un id, un nombre, una descripción y el precio:

```php
<?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;
    }
 }
?>
```

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.

```php
<?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;
    }
 }
?>
```

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.

Para probar el funcionamiento la clase creada de Producto es 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 />';
?>
```

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;
    }
 }
?>
```

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.

Puntos a tener en cuenta:

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);
 }
```

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, en la clase madre no usamos **private** para las propiedades sino que usamos otra palabra reservada, **protected**.

Hay que tener en cuenta que **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
<?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
<?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';
 }
?>
```

Sesiones en PHP

Las sesiones permiten mantener y manejar información de los usuarios en el servidor mediante el array **`$_SESSION`**

`$_SESSION` es un array especial utilizado para guardar información a través de los requests que un usuario hace durante su visita a un sitio web o aplicación. La información guardada en una sesión puede llamarse en cualquier momento mientras la sesión esté abierta.

A cada usuario que accede a la aplicación e inicia sesión se le asigna un session ID único, y es lo que le permite identificar la sesión y que esté disponible para ese usuario en concreto. La forma más segura de manejar sesiones es almacenando en el cliente sólo esta session ID, y cualquier información de la sesión guardarla en el lado del servidor.

Para identificar al usuario que generó las variables de sesión, el servidor genera una clave única que es enviada al navegador y almacenada en una cookie. Luego, cada vez que el navegador solicita otra página al mismo sitio, envía esta cookie (clave única) con la cual el servidor identifica de qué navegador proviene la petición y puede rescatar de un archivo de texto las variables de sesión que se han creado. Cuando han pasado 20 minutos sin peticiones por parte de un cliente (navegador) las variables de sesión son eliminadas automáticamente (se puede configurar el entorno de PHP para variar este tiempo)

Iniciar una sesión

Antes de guardar cualquier información en una sesión es necesario iniciar el manejo de la sesión (session handling). Esto se hace al principio del código PHP, y debe hacerse antes de que cualquier texto, HTML o JavaScript se envíe al navegador. Para comenzar la sesión tan sólo hay que utilizar la función **`session_start()`**

Para eliminar un valor de sesión simplemente se utiliza **unset():**

```
unset($_SESSION["usuario"]);
```

- Para desvincular todos los valores de sesión de una vez se puede emplear **session_unset()**

```
session_unset();
```

Ambas acciones anteriores sólo afectan a los valores guardados en la sesión, no a la propia sesión. Todavía se pueden guardar nuevos valores en $_SESSION. Si se quiere dejar de utilizar por completo la sesión, por ejemplo cuando el usuario hace log out, se utiliza **session_destroy()**:

Es muy recomendable que cuando ya no se necesite la sesión se destruya con session_destroy(), en vez de desvincular el valor de sus valores con session_unset(). Si sólo se desvinculan los valores la sesión permanecerá activa, lo que deja la puerta abierta a intrusos.

A continuación se muestra un ejemplo donde cargaremos en un formulario el nombre de usuario y clave de un cliente, en la segunda página crearemos dos variables de sesión y en una tercera página recuperaremos los valores almacenados en las variables de sesión. La primer página es un formulario HTML puro:

```
<html>
<head>
<title>Problema</title>
</head>
<body>
<form action="pagina2.php" method="post">
Ingrese nombre de usuario:
<input type="text" name="campousuario"><br>
Ingrese clave:
<input type="password" name="campoclave"><br>
<input type="submit" value="confirmar">
</form>
</body>
</html>
```

La segunda página es donde creamos e inicializamos las dos variables de sesión:

```
<?php
session_start();
$_SESSION['usuario']=$_REQUEST['campousuario'];
$_SESSION['clave']=$_REQUEST['campoclave'];
?>
<html>
<head>
<title>Problema</title>
</head>
<body>
Se almacenaron dos variables de sesión.<br><br>
<a href="pagina3.php">Ir a la tercer página donde se recuperarán
las variables de sesión</a>
</body>
</html>
```

Cuando creamos o accedemos al contenido de variables de sesión debemos llamar a la función session_start() antes de cualquier salida de etiquetas HTML.

Para almacenar los valores en las variables de sesión lo hacemos:

```
$_SESSION['usuario']=$_REQUEST['campousuario'];
$_SESSION['clave']=$_REQUEST['campoclave'];
```

Es decir, tenemos el vector asociativo $_SESSION que almacena las variables de sesión.

Por último, esta página tiene un hipervínculo a la tercera página. La última página de este ejemplo tiene por objetivo acceder a las variables de sesión:

```
<?php
session_start();
?>
<html>
<head>
<title>Problema</title>
</head>
<body>
<?php
echo "Nombre de usuario recuperado de la variable de
sesión:".$_SESSION['usuario'];
```

```
echo "<br><br>";
echo "La clave recuperada de la variable de sesión:".$_SESSION['clave'];
?>
</body>
</html>
```

De nuevo vemos que la primera línea de esta página es la llamada a la función session_start() que, entre otras cosas, rescata de un archivo de texto las variables de sesión creadas para ese usuario (recordemos que desde el navegador todas las veces retorna una cookie con la clave que generó PHP la primera vez que llamamos a una página del sitio).

Para mostrar las variables de sesión, las accedemos por medio del vector asociativo $_SESSION:

```
echo "Nombre de usuario recuperado de la variable de
sesión:".$_SESSION['usuario'];
echo "<br><br>";
echo "La clave recuperada de la variable de sesión:".$_SESSION['clave'];
```

Tengamos en cuenta que en cualquier otra página del sitio tenemos acceso a las variables de sesión sólo con llamar inicialmente a la función session_start().

Actividad 01

Los aprendices deberán realizar los siguientes ejercicios en PHP con base en las instrucciones dadas por el instructor

1. Crear los siguientes archivos:
   Empleado.class.php: clase empleado que definirá como atributos el nombre, sueldo, email, fecha de nacimiento, peso y altura. Definir un método inicializar que lleguen como datos todos los atributos de esta clase. Plantear un segundo método que imprima el nombre y un mensaje si debe o no pagar impuestos (si el sueldo supera los $2.000.000 paga impuestos). Este segundo método deberá llamar a un tercer método que será privado, recibirá como parámetro el sueldo y devolverá TRUE o FALSE para indicar si debe pagar o no impuestos.
   Tener otro método que permita calcular la edad con base a la fecha de nacimiento y que muestre un mensaje si la persona puede o no votar
   index.php: archivo que contiene el formulario html con todos los campos necesarios del empleado. La acción del formulario deberá mandar los datos por POST al archivo procesardatos.php
   procesardatos.php: archivo que recibirá los datos del empleado, creará un objeto de la clase empleado y llamará los métodos inicializar e imprimir para mostrar el resultado
   Se debe tener en cuenta la función de PHP **include** para incluir el archivo Empleado.class.php y poder utilizar dicha clase en el archivo procesardatos.php

2. Crear una clase llamada cuenta con los atributos (número de cuenta, nombre titular, saldo, tipo de cuenta). Crear métodos para ingresar dinero a la cuenta, retirar dinero de la cuenta pagar una factura y transferir saldo a otro usuario. Se debe crear la interfaz para realizar cada acción y mostrar el resultado
3. Modificar el ejercicio anterior para que los objetos de la clase cuenta sean guardados en una variable de sesión y de esta manera podamos tener guardada la información que se registre de cada cuenta, entre ella sus saldos para que con las diferentes transacciones que tengamos, podamos saber el saldo real de cada cuenta. Se deberá crear la funcionalidad de crear cuentas y guardar estos objetos en un vector dentro de una variable de sesión