Programación orientada a objetos en PHP

Cómo trabajar con clases y objetos en PHP

Desde su aparición en lenguajes como C++ y Java, la programación orientada a objetos es una técnica que ha ido ganando popularidad por las ventajas que ofrece. Otros lenguajes como Perl o PHP, que inicialmente no contemplaban esta funcionalidad, la han adoptado posteriormente. PHP, a partir de PHP 5, dispone de una implementación completa del modelo de objetos. En este artículo se da una introducción paso a paso al los conceptos existentes en la programación orientada a objetos, ofreciendo ejemplos de su aplicación a un script escrito en PHP .

Clases

Una clase es la definición de un tipo de objeto, en forma de una colección de *variables* (llamadas propiedades) y *funciones* (llamadas métodos). En el siguiente ejemplo podemos ver cómo se define en PHP una clase “Coche”, con una variable llamada “velocidad”, y funciones “**poner_en_marcha()**” y “**leer_velocidad()**”:

```php
<?php

class Coche
{
	public $marca;
	public $modelo;
	public $velocidad = 0;
	public function poner_en_marcha($nueva_velocidad)
	{
		$this->velocidad = $nueva_velocidad;
	}
	public function leer_velocidad()
	{
		return $this->velocidad;
	}
}
?>
```

En este sencillo ejemplo vemos una serie de elementos y sintaxis que son propias de la programación de objetos:

- La palabra clave “class” se utiliza para dar un nombre a la clase.
- La palabra clave “public” se utiliza para definir variables (propiedades) y funciones (métodos) de la clase. Más adelante veremos que también podemos utilizar la palabras clave “private”, “static”,etc. al definir las propiedades de la clase.
- La variable “$this” se utiliza para referirse a variables (propiedades) o funciones (métodos) definidas en la propia clase, utilizando la sintaxis “$this->variable” y $this->metodo()”.

Propiedades y métodos

Como hemos dicho, una clase es simplemente una colección de definiciones de variables y funciones. Pero en programación orientada a objetos, las variables definidas en una clase se denominan “propiedades”, y las funciones definidas en la clase se denominan “métodos”. En el resto de este artículo se utilizarán preferentemente las denominaciones “propiedades” y “métodos”.

Objetos

Para utilizar una clase, creamos instancias de la clase, a las que llamamos objetos. Por ejemplo:

```php
<?php
$coche1 = new Coche;
$coche1->poner_en_marcha(80);
$coche2 = new Coche;
$coche2->poner_en_marcha(120);
?>
```

En este ejemplo hemos creado, con la palabra reservada “new”, dos objetos de la clase Coche, $coche1 y $coche2. Como vemos, podemos asignar distintos valores a las propiedades de cada uno de ellos.

Métodos mágicos

PHP nos ofrece una serie de métodos mágicos que son llamados cuando realizamos ciertas acciones con el objeto, como por ejemplo cuando un objeto es creado. Esto nos ayuda a automatizar trabajo en ciertas ocasiones.

Constructores y destructores

Cuando una clase se crea o se destruye, se llama a unos métodos mágicos como ya hemos adelantado. Si queremos realizar alguna acción en cualquiera de los dos eventos, tendremos que definir esos métodos e implementar su funcionalidad

Inicialización de un objeto (Constructores)

Un objeto puede requerir algún tipo de inicialización cuando es creado. Para ello, añadimos a la definición de la clase un método con el nombre "__construct()". Por ejemplo, podemos crear dicho método para inicializar la marca y modelo de un objeto de la clase Coche:

```
function __construct($la_marca, $el_modelo)
{
    $this->marca = $la_marca;
    $this->modelo = $el_modelo;
}
```

Como podemos ver en el ejemplo, aunque las propiedades se definen con el signo "$" de la forma "$propiedad", cuando se utilizan se hace referencia a ellas con la sintaxis $this->propiedad, sin el signo "$".

Hay que tener en cuenta que en PHP **NO existe el polimorfismo** es decir, que varios métodos, se llamen igual y que solo difieran en el numero de parámetros, por tanto **solo puedo tener un constructor**. Si no se pone ninguno se da por sentado que tiene el constructor vacío (sin parámetros) → **function** __construct()

Destrucción de un objeto

Por otro lado, el destructor se llama cuando la clase es destruida. Esto ocurre automáticamente cuando PHP termina de ejecutarse, libera todos los recursos que tenga asociados y va llamando al método __destruct() de todas las clases en caso de que lo tengan. También podemos forzarlo usando la función unset sobre el objeto.
En el destructor se suelen implementar cierres de conexiones a bases de datos, liberación de memoria, etc.

Por ejemplo:

```
class Usuario {
    public $usuario;
    public $password;

    public function __construct($usuario) {
        $this->password = '1234';
        $this->usuario = $usuario;
    }

    public function __destruct() {
        echo 'Muero...';
    }
}

$usuario = new Usuario('juan');
unset($usuario); // 'Muero...';
```

Propiedades estáticas

Las clases y métodos estáticos pueden ser de cualquiera de los tipos anteriores, **public**, **protected** o **private** pero tienen la peculiaridad de que no requieren que la clase sea instanciada para poder acceder a ellos.

Además, las propiedades estáticas guardan el valor durante la ejecución del script. Veamos un breve ejemplo:

```
class Usuario {
    public $usuario;
    public $password;

    public static $contador = 0;

    public function __construct() {
        self::$contador++;
    }
}

$usuario1 = new Usuario;
echo Usuario::$contador; // 1
$usuario2 = new Usuario;
echo Usuario::$contador; // 2
```

Ten en cuenta, que cuando nos referimos a una propiedad o método estático, no podremos hacer uso de $this, pero tendremos acceso a **self**, que viene a ser lo mismo **solo que únicamente podremos usar métodos y propiedades estáticas**.

Una **propiedad, ó método** estático puede ser accedida desde fuera de la clase utilizando la sintaxis **CLASE::PROPIEDAD**, sin necesidad de crear una instancia, en el ejemplo anterior :

```
echo Usuario::$contador;
```

En la definición de una propiedad podemos añadir la palabra reservada “static”. por ejemplo:

```
class Coche
{
    public static $velocidad_maxima = 160;
    public $velocidad;
    ...
}
```

Una propiedad ó método estático puede ser accedida por el script utilizando la sintaxis CLASE::PROPIEDAD, sin necesidad de crear una instancia. Por ejemplo:

```
print "La velocidad máxima de un coche es" . Coche::$velocidad_máxima . "n";
```

Por otra parte, el valor de una variable estática puede ser cambiado, pero es único y compartido por todos los objetos de la clase. Si ejecutamos el siguiente código:

```
class Coche
 {
     public static $velocidad_maxima = 80;
     ...
 }

echo "La velocidad máxima de un coche es: " . Coche::$velocidad_maxima .
"n";

$coche1 = new Coche();
echo "La velocidad máxima de coche1 es: " . $coche1::$velocidad_maxima .
"n";
echo "nCambiamos el valor de la propiedad estátican";
Coche::$velocidad_maxima = 160;
echo "Ahora, la velocidad máxima de un coche es: " .
Coche::$velocidad_maxima . "n";
$coche2 = new Coche();
echo "La velocidad máxima de coche1 es: " . $coche1::$velocidad_maxima .
"n";
echo "La velocidad máxima de coche2 es: " . $coche2::$velocidad_maxima .
"n";
```

Al ejecutarlo, obtenemos como resultado:

```
a velocidad máxima de un coche es: 80
La velocidad máxima de coche1 es: 80

Cambiamos el valor de la propiedad estática
Ahora, la velocidad máxima de un coche es: 160
La velocidad máxima de coche1 es: 160
La velocidad máxima de coche2 es: 160
```

Como vemos, incluso en el caso del objeto $coche1, que fue creado antes de cambiar el valor de $velocidad_maxima, cuando se accede a dicha propiedad estática se obtiene el valor modificado.

Ejemplo de acceso a un **método estático**.

```
class Cadena {
  public static function largo($cad)
  {
    return strlen($cad);
  }
  public static function mayusculas($cad)
  {
    return strtoupper($cad);
  }
  public static function minusculas($cad)
  {
    return strtolower($cad);
  }
}
```

```
$c='Hola';
echo 'Cadena original:'.$c;
echo '<br>';
echo 'Largo:'.Cadena::largo($c);
echo '<br>';
echo 'Toda en mayúsculas:'.Cadena::mayusculas($c);
echo '<br>';
echo 'Toda en minúsculas:'.Cadena::minusculas($c);
```

Propiedades constantes

Una propiedad constante contiene un valor asignado en su definición, que no puede ser modificado posteriormente.

Estas propiedades se definen utilizando la palabra reservada “const”. Por otra parte, no utilizan el signo “$” al comienzo del nombre.

```
class Coche
 {
    const descripcion = "Esta es la clase Coche";
    ...
}
```

En el código de la clase (**dentro de la clase**), se hace referencia a estas constantes con la sintaxis:

self::$descripcion.

Para acceder al valor de la constante desde fuera de la clase debemos usar la variable que hace de objeto, o el nombre de la clase seguido del símbolo “*dos puntos*” dos veces (::), así:

```
<?php

$clase = new MiClase();

echo $clase::MI_CONSTANTE; //Imprime 'Una Constante'
echo MiClase::MI_CONSTANTE; //Imprime 'Una Constante'

?>
```

Clases derivadas. Herencia de clases

Una clase puede ser definida a partir de otra clase que se toma como punto de partida, mediante la palabra reservada “extends”.

La nueva clase puede definir propiedades y métodos adicionales, y puede redefinir los métodos de la clase padre. Los métodos que no se redefinen están disponibles en la nueva clase con la misma funcionalidad que en la clase padre (es decir, se “heredan”).

Ejemplo:

```
class Coche
   {
       ...
   }

   class Deportivo extends Coche
   {
       public $potencia;

       function __construct($marca, $modelo,$la_potencia)
       {
       parent::__construct($marca, $modelo);
       $this->potencia = $la_potencia;
       }

       public function cambiar_potencia($pot)
       {
           $this->potencia=$pot;
       }
   }

$coche3 = new Deportivo("Ferrari","Testarossa");
$coche3->poner_en_marcha(190);
```

Como vemos, en la nueva clase podemos utilizar la propiedad “velocidad” y el método “poner_en_marcha()”, cuya definición se encuentra en la clase “Coche” de la que se deriva. Por otra parte, se ha redefinido el método “leer_velocidad()” de modo que presente la velocidad en negrita si supera el valor de 160.

```
?php

class Figura
{
        protected $base;
        protected $altura;

        function __construct($base, $altura)
        {
                $this->base = $base;
                $this->altura = $altura;
        }
}

class Rectangulo extends Figura
{
    function area(){
                return $this->base * $this->altura;
        }
}
```

```
class Triangulo extends Figura
{
    function area(){
                return $this->base * $this->altura /2;
        }
}

$rectangulo = new Rectangulo(2,2);
$triangulo = new Triangulo(2,2);

echo "<div>Para base = 2 y altura = 2:
                <ul>
                        <li>Área para el rectángulo: "
 . $rectangulo->area() . ">/li>
                        <li>Área para el triángulo: "
 . $triangulo->area() . "</li>
                </ul>
        </div>"
?>
```

Herencia múltiple

PHP al igual que Java u Objectice-C no soporta herencia múltiple. Ello se debe a que si se heredasen dos clases que contengan métodos con el mismo nombre no sabría que clase debería de tener preferencia.

Pero si estamos ante una situación en la que necesitamos herencia múltiple no está todo perdido. Ya que existen dos métodos para conseguir simular este tipo de herencia. Aunque el segundo de ellos solo está disponible a partir de PHP 5.4.

1. **Uso de interfaces para simular la herencia múltiple**
2. **Uso de 'traits'**

Ejemplo completo Herencia

Por ejemplo, vamos a definir la clase persona de la siguiente manera:

```
class Persona {
    $nombre;
    $apellido;
    $dni;
    $calle;



    function __construct($nombre, $apellido, $dni, $calle) {
        print "Creando objecto de la clase Persona con nombre completo: " .
               $nombre . " " . $apellido . " y DNI: " . $dni . "\n";
        $this->nombre = $nombre;
        $this->apellido = $apellido;
        $this->dni = $dni;
        $this->calle = $calle;
    }
   function mostrarDatos(){
        print "Nombre: " .  $this->nombre .  "\n";
        print "Apellido: " .  $this->apellido .  "\n";
        print "DNI: " .  $this->dni . "\n";
        print "Calle: " .  $this->calle . "\n\n";
    }
}
```

```
class Vendedor extends Persona {
    $identificadorVendedor;

    function __construct($nombre, $apellido, $dni, $calle,
$identificadorVendedor) {
        parent::__construct($nombre, $apellido, $dni, $calle);
        $this->identificadorVendedor = $identificadorVendedor;
        print  $nombre . " es un VENDEDOR \n";
    }

    function vender() {
        print "Vendedor " .  $this->nombre . " con numero: " . $this-
>identificadorVendedor . ", vende un producto \n";
    }
}


class Comprador extends Persona {

    function comprar() {
        print "Comprador " .  $this->nombre . " compra un producto\n";
    }
}
$pedro = new Vendedor ("Pedro", "Sanchez", "40123903K", "Calle Falsa 123",
"4214");
$juan = new Comprador("Juan", "Gabilondo", "10392129E", "Calle Loca n: 2");
$pedro->vender();
$juan->comprar();
```

Visibilidad de métodos y propiedades

Hasta ahora tanto las propiedades como los métodos que hemos vistos, eran todos **public**. Ahora vamos a ver qué significa eso y el resto de tipos que tenemos.

Public →A los métodos y propiedades public, se puede acceder desde cualquier sitio. Tanto dentro de la clase como fuera.

Protected →Cuando declaramos una propiedad o método como protected, solo podremos accederlos desde la propia clase o sus descendientes (las clases hijas).

Private → Los métodos y propiedades private solo pueden ser leídos dentro de la propia clase **que lo define**. Este breve ejemplo nos muestra lo que pasaría:

Ejemplo:

```
class ClasePadre
{
    protected $protegida = 'Protegida';
    private $privada = 'Privada';
   function imprime_propiedades() {
        echo $this->protegida . ", " . $this->privada;
        echo "n";
    }
}
```

```
$obj = new ClasePadre();

// Las siguientes sentencias devuelven un error de la forma:
// PHP Fatal error:  Cannot access protected property ClasePadre::$protegida
echo $obj->protegida; // Error Fatal
echo $obj->privada;   // Error Fatal

$obj->imprime_propiedades(); // Imprime "Protegida, Privada"
```

La diferencia entre ambos tipos está en su comportamiento cuando se crea una clase derivada de otra: Una propiedad “protected” es heredada y puede ser redefinida en la clase derivada, pero si se intenta acceder a una propiedad “private” en la clase hija se obtiene el valor “undefined”.

```
class ClaseHijo extends ClasePadre
{
    // Podemos redeclarar la variable protected
    protected $protegida = 'Protegida Hija';


    function imprime_propiedades() {
        echo $this->protegida . ", " . $this->privada;
        echo "n";
    }
}

$obj2 = new ClaseHijo();
echo $obj2->protegida . "n"; // Error Fatal

// Las siguiente sentencias devuelven un error de la forma:
// PHP Notice:  Undefined property: ClaseHijo::$privada
echo $obj2->privada . "n";
$obj2->imprime_propiedades(); // Imprime "Protegida Hija,"
```

Clases abstractas en PHP

Ya sabemos que las clases normales permiten ser heredadas y, al menos, cuentan con algún método público que permite acceder a la información para leerla o manipularla una vez creada una instancia. Una estructura tipo podría contar con un método constructor, otro con funciones getter y otro con funciones setter.

```
class clase_normal
{
  protected $propiedad1;
  public function __construct ($recoge) {
     $this->propiedad1 = $recoge;
  }
  public function metodoSetter () {
     $this->propiedad1 = $this->propiedad1 + 1;
     return $this->propiedad1;
  }
```

```
  public function metodoGetter () {
    return $this->propiedad1;
  }
}
$nueva_instancia = new clase_normal (5);
echo $nueva_instancia ->metodoSetter();
```

Las clases abstractas funcionan de forma distinta. Para declararlas se antepone la palabra reservada **abstract,** tanto en la definición de la clase como de los métodos.

```
abstract class SoyUnaClaseAbstracta

{

abstract protected function soyUnMetodoAbstracto();

}
```

De las clases abstractas **no se puede sacar ninguna instancia**, es decir, **NO se puede definir ningún objeto directamente a partir de ellas**. Dicho en código, esto no funciona.

```
abstract class SoyUnaClaseAbstracta

{

    abstract protected function soyUnMetodoAbstracto();

    public function soyUnMetodoPublico() {

        echo "No funciona";

}

}

// Esto no se vale

$mi_claseAbstracta = new SoyUnaClaseAbstracta();

$mi_claseAbstracta->soyUnMetodoPublico();
```

Esto es lógico, ya que carece de sentido instanciar una abstracción que no está relacionada con nada en concreto.

En cambio, claro está, sí que podemos instanciar todas las clases derivadas, herederas, de una clase abstracta.

Métodos abstractos

Una clase abstracta puede incluir métodos privados, protegidos y públicos, pero además estos métodos pueden declararse como abstractos. De hecho, toda clase que contenga al menos un método abstracto debe definirse abstracta, aunque una clase abstracta no necesita contener obligatoriamente un método abstracto.

Los **métodos abstractos** no se implementan, es decir, no se desarrollan en la clase madre, sino en las clases hijas. Y esto es obligatorio: todo método abstracto de la clase madre debe recogerse en la clase hija.

Expresado en código, si intentamos hacer esto:

```
abstract class SoyUnaClaseAbstracta
{
abstract protected function soyUnMetodoAbstracto();
}
class hija extends SoyUnaClaseAbstracta
{
// No implementamos el método...
}
```

Nos dirá que «***Fatal error**: Class hija contains 1 abstract method and must therefore be declared abstract or implement the remaining methods (SoyUnaClaseAbstracta::soyUnMetodoAbstracto)*».

¿Y para qué diantres sirven entonces? muy sencillo, para que la jerarquía de clases sea más sólida, pues fuerzan a que las clases hijas se comporten de determinada manera. Por ejemplo, imaginemos que tenemos una clase abstracta que dibuja figuras. En la clase no están definidas ni qué figuras son ni en qué soporte ni con qué deben pintarse, pero sí que deben dibujarse a partir de los parámetros recibidos; pues ahí podría funcionar un método abstracto dibujar(), que ya en la clase hija rectángulo se definirá como trazar 4 lados, en la de triángulo, 3 lados, etcétera.

Importante: se puede modificar el tipo de acceso de un método abstracto cuando se implementa en la clase hija, pero no de forma más restrictiva. Es decir, se puede redefinir un método private como protected o public y uno protected como public, pero no al revés.

```
abstract class SoyUnaClaseAbstracta
{
      abstract protected function soyUnMetodoAbstracto();
}
class hija extends SoyUnaClaseAbstracta
{
      // Esto sí funciona
      public function soyUnMetodoAbstracto() {
        echo "Jajajaja";
      }
}
```

Clases finales

En cierta manera, lo contrario de las clases abstractas son las clases finales, que no pueden ser **heredadas**. También se puede declarar final un método, en cuyo caso **no puede sobrescribirse**. Tanto las clases como los métodos se definen finales anteponiendo la palabra final.

```
final class SoyUnaClaseFinal
{
  final protected function soyUnMetodoFinal() {
      return "Jajajaja, nadie me puede redefinir y dominaré el mundo";
  }
}
// Esto no funciona
class hija extends SoyUnaClaseFinal
{
// Y esto también daría error
    function soyUnMetodoFinal() {
    }

}
```

Instanceof

Instanceof es USADO para determinar si una variable de PHP es una Instancia de un objeto de CIERTA Clase :

```
<?php
clase  MiClase
{
} clase

NoMiClase
{
}
$a  = new  MiClase;
$b = new  NoMiClase;

var_dump ($a  instanceof  MiClase);
var_dump ($a  instanceof  NoMiClase);
?>
El Resultado del EJEMPLO seria:
bool (true)
bool (false)
```

Arrays de Objetos

Supongo que tengo una clase que se llama **Usuario** con este constructor:

```
function __construct($id, $nombre, $apellidos, $codigoPostal){
        $this->id = $id;
        $this->nombre = $nombre;
        $this->apellidos = $apellidos;
        $this->codigoPostal = $codigoPostal;
}
```

Para crear un Array de Objetos lo haré de la siguiente forma:

```
include("clases.php");

    $vector_usuarios = array();

    $tmp = new Usuario(1, "José", "Gómez Martínez", 14004);
    array_push($vector_usuarios, $tmp);
    $tmp = new Usuario(2, "Javier", "Pérez García", 28080);
    array_push($vector_usuarios, $tmp);
    $tmp = new Usuario(3, "Jorge", "Reina Ramírez", 18001);
    array_push($vector_usuarios, $tmp);

    echo "<h2>Ejemplo array objetos con PHP </h2>";

    foreach($vector_usuarios as $elemento_vector)
    {
        echo "<div>
                <h3>Usuarios sistema</h3>
                <ul>
                    <li>Id. - " . $elemento_vector->getId() . "</li>
                    <li>Nombre - " . $elemento_vector->getNombre() .
"</li>
                    <li>Apellidos - " . $elemento_vector->getApellidos() .
"</li>
                    <li>Código postal - " .
$elemento_vector->getCodigoPostal() . "</li>
                </ul>
            </div>";
    }
```

El patrón de diseño Singleton (implementado en PHP)

La idea del patrón **Singleton**, también llamado **de instancia única**, es **restringir la creación de objetos pertenecientes a una clase a sólo uno**. Garantiza que sólo hay una instancia y **proporciona un único punto de acceso**.

La idea es que la clase tenga un **constructor privado** para que no pueda ser accedido desde fuera de la clase y que aunque haya clases hijas estas no puedan **instanciar objetos**. También

debe **tener una propiedad estática privada donde almacenaremos nuestra instancia** y un método estático que sea **el punto de acceso global** a la misma, devolviéndola cuando se lo llama.

Vamos con el ejemplo típico y tópico del objeto de base de datos, que suele ser el uso más común de este tipo de clases. Porque crear varias instancias de acceso a una base de datos en la mayoría de los casos es un malgasto de recursos, lo mejor suele ser tener una sola instancia para realizar las conexiones.

```
class Database {
//Propiedad estática, inicializada a nulo, donde guardaremos
//la instancia de la propia clase
    static private $instance = null;
//Definimos el método constructor como privado
    private function __contruct() {}

//Método estatico que sirve como punto de acceso global
    public static function getInstance() {
        if (self::$instance == null) {
//Si no hay instancia creada, lo hace. Si la hay tira p'alante
            self::$instance = new Database();
        }
//Al final devuelve la instancia
        return self::$instance;
    }

 //a continuación ya meterías todas las funciones necesarias
// para un objeto de base de datos. CRUD, etc...
}
```

Con el código así todavía es posible que alguien logre hacer más de una instancia. Y te dirás ¿cómo? Pues porque **todavía es posible usar la clonación y la serialización para lograrlo**. Cosa mala, pero con los métodos mágicos de PHP5 podemos arreglarnos.

Para evitar la cuestión de la clonación definimos el método mágico **__clone()** en la clase, para que en lugar de permitir el clonado del objeto nos **lance un error y detenga el script**:

```
public function __clone()
{
  trigger_error("No puede clonar una instancia de ".
get_class($this) ." class.", E_USER_ERROR );
}
```

El método mágico **__clone** se dispara en el momento de clonar el objeto mediante la instrucción **clone.**

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

```
    public function __toString(){
        return $this->_nombre;
    }

    public function setNombre( $nombre ){
        $this->_nombre = $nombre;
    }

    public function __clone(){
        $this->_nombre = 'Objeto Clonado';
    }
}
$persona = new Persona( 'Juan' );
$persona2 = clone $persona;

var_dump($persona); //Imprime: object(Persona)#1 (1) {
["_nombre":"Persona":private]=> string(4) "Juan" }
var_dump($persona2); //Imprime: object(Persona)#2 (1) {
["_nombre":"Persona":private]=> string(14) "Objeto Clonado" }
?>
```

No es necesario pasarlo ningún parámetro a __clone, y como se aprecia en el ejemplo su uso es muy sencillo, simplemente definimos la rutina a ejecutar y al clonar el objeto nos ha cambiado el nombre de '*Juan*' a '*Objeto clonado*'.

Y vamos con el tema de la **serialización y deserialización**. Con la función **serialize()** se puede almacenar una representación apta de una variable (y una instancia de un objeto es, al fina, una variable) mientras que con **unserialize()** podemos acceder a ella como se si tratara de un clon. De nuevo hay que redefinir un método mágico, en concreto esta vez **__wakeup()**, que es el **método que se invoca al deserializar** un objeto. De esa forma se evita que pueda ser accedido. También podría hacerse con**__sleep()**, que es el que se invoca al serializar, pero veo más práctico evitar la deserialización.

```
    public function __wakeup()
    {
      trigger_error("No puede deserializar una instancia de ".
    get_class($this) ." class.", E_USER_ERROR );
    }
```

Serializar Objetos en PHP

En determinadas ocasiones es posible que necesitemos serializar un Objeto o un array para así por ejemplo guardarlo en un archivo de texto o en una tabla de una base de datos, o bien propagarlo en una sesión (necesario dependiendo de la configuración de PHP).
Para serializar un Objeto usaremos la función de PHP **serialize()**, que devuelve una cadena de texto con la representación del mismo, y para volverlo a convertir en un Objeto usaremos **unserialize()**.

En el siguiente ejemplo serializamos un Objeto en PHP y lo pasamos en la sesión.

Supongamos que tienes el siguiente array:

```
$arr = array(
  "fruta" => "manzana",
  "objeto" => "bicicleta",
  "animales" => array(
              "perro",
              "gato",
              "caballo")
);
```

Si lo deseas guardar en una tabla de una base de datos, lo puedes serializar de esta forma:

```
$string = serialize($arr);
```

La función serialize devolverá un valor fácilmente almacenable o transmitible:

```
a:3:{s:5:"fruta";s:7:"manzana";s:6:"objeto";s:9:"bicicleta";
s:8:"animales";a:3:{i:0;s:5:"perro";i:1;s:4:"gato";
i:2;s:7:"caballo";}}
```

Para recuperar el valor original en PHP a partir de la cadena seriada:

```
$arr = unserialize($string);
```

Un ejemplo del uso de la serialización la tenemos en la forma en que **WordPress** almacena información variada en algunas de sus tablas.

Excepciones

Una excepción en PHP es similar a las excepciones en otros lenguajes.

Una excepción puede ser lanzada y atrapada. Dentro de un bloque ***try*** definiremos nuestro código para facilitar la captura de posibles excepciones. Cada bloque try debe poseer al menos un bloque ***catch***, y este actuará en caso de que una excepción sea lanzada.

La excepción la lanzaremos mediante ***throw***

Cuando una excepción es lanzada todo el código que sigue a ese lanzamiento dentro del bloque try no será ejecutado, entonces el flujo del programa buscará el “catch” correspondiente a ese bloque try. De no haber un catch para ese bloque PHP omitirá un fatal error con un mensaje de “Uncaught Exception”, a menos que esté definido un gestor con **set_exception_handler().**

```php
<?php

function exception_error_handler($severidad, $mensaje, $fichero, $línea) {
   if (!(error_reporting() & $severidad)) {
      // Este código de error no está incluido en error_reporting
      return;
   }
   throw new ErrorException($mensaje, 0, $severidad, $fichero, $línea);
}
//set_error_handler — Establecer una función de gestión de errores definida por el usuario
set_error_handler("exception_error_handler");

/* Desencadenar la excepción */

if (!isset ($_POST['calcular']))
{
ECHO 'DIVISION ENTERA<br> <br>';
echo'<form action = "index.php" method = "POST">
   Dividendo: <input type="text" name="dividendo" value="" size="5" />
   <br><br>Divisor: <input type="text" name="divisor" value="" size="5" />
   <br><br><input type="submit" value="Calcular" name="calcular" />
</form>';
}
else
{
   try{
            $dividendo=$_POST['dividendo'];
           $divisor=$_POST['divisor'];

            $resultado=$dividendo/$divisor;

           echo "<br><br>El resultado es :".$resultado;

   }
   catch(Exception $ex)
   {
      echo "ERROR.....<BR> Introduce bien los datos";
   }
}
```

Es muy común en una clase de **base de datos** que se lance una excepción cuando no se puede conectar a la base.

Es muy común en una clase de **base de datos** que se lance una excepción cuando no se puede conectar a la base.

EJERCICIOS

Ejercicio 1: Objeto animal, introducción a objetos

Dada la siguiente declaración de una clase de nombre animal:

```
class Animal{

    private $edad;
    private $nombre;

    public function __construct()
    {
    }
    public function getNombre(){
        System.out.println(nombre);
    }
    public function getEdad(){
        System.out.println(edad);
    }

}
```

a) Hacer un programa de nombre **Animalito** en el que se declaren 2 objetos que pertenezcan a la clase Animal. Invocar a los métodos **getEdad** () y **getNombre** desde el segundo objeto.

b) Añadirle a la clase los métodos **setNombre** y **setEdad** que permitan asignarle valores a las propiedades de la clase.

c) Probar estos métodos desde los objetos creados en el apartado

d) Crea un método que se llame **cumpleanios** que incremente la edad del animal.

Ejercicio 2: Clase rectángulo con constructores y pruebas

Programar una clase de nombre **Rectangulo** que describa y de soporte a esta figura geométrica. La clase tendrá dos propiedades privadas de nombres **largo** y **alto**, y los **métodos**:

- set y get que permitan asignar y ver el contenido de las propiedades de la clase.
- superficie, que devolverá la superficie del rectángulo.
- perímetro, que devolverá el perímetro del rectángulo
- visualiza, que visualizará información del rectángulo con el formato:

  largo: xxxx

  alto: xxxxx

superficie: xxxx

perímetro: xxxx

Ejercicio 3: Fechas

Programar una clase de nombre Fecha que guarde una fecha como tres enteros privados de nombres **dia**, **mes** y **anyo**. Serían las propiedades de la clase.

Métodos de la clase:

- Constructor que reciba 3 datos de tipo entero y se los asigne a los propiedades **dia mes** y **anyo**.
- Métodos **set** y **get** para las propiedades de la clase.
- **incrementate($d)**: incrementa en d días la fecha.
- **imprimete():** escribe la fecha en el formato: 5-Febrero-1010 (para los valores 5-2-2010)
- **método privado mesLetra** que devuelva el mes en letras asociado al mes numérico guardado en una determinada instancia (objeto) de la clase; este método será invocado desde imprímete.

Hacer un programa en el que se declaren objetos de la clase Fecha, y comprobar si el comportamiento de los métodos es el correcto.

Ejercicio 4: Clase llamada (con clase hora)

Programar una clase de nombre **Llamada** que guardará la información relativa a una llamada telefónica. Tendrá las siguientes propiedades y métodos:

Atributos: (todas ellas privadas)

$numeroTelefono;

$comienzoLlamada;(tipo Hora)

$finLlamada;(tipo hora)

Métodos de la clase (públicos):

- Método **constructor** que reciba tres variables de tipo (Telefono, Hora_Comienzo, Hora_fin) y se los asigne a las propiedades de la clase.
- Método de nombre **escribeHoraComienzo** que visualice el valor del atributo **comienzoLLamada** con el formato: 13:25:13.
- Método de nombre **escribeHoraFin** que visualice el valor del atributo **finLLamada** con el mismo formato que el método anterior.

- Método de nombre **duraciónLlamada** que devuelva en segundos la duración de la llamada. Si la hora de finalización es inferior a la hora de comienzo de la llamada el método devolverá -1.
- Métodos **set** y **get** para todas los atributos.

Ejercicio 5: Clase cuenta corriente, paso de enteros a cadenas y empezando con excepciones

Programar una clase de nombre **Cuenta** que permita almacenar e interactuar con la cuenta bancaria de un cliente. Esta clase tendrá las siguientes propiedades, métodos y constructores:

Atributos privados:

- **nombre**, **apellidos**, **dirección**, **teléfono**, **NIF**.
- **Saldo**, **tipo_interes**

Constructor:

- Constructor al que se le pasen como argumentos todas las propiedades que tiene la clase

Métodos públicos:

- **sacarDinero($cantidad)**: le resta al saldo una cantidad de dinero pasada como argumento.
- **ingresarDinero($cantidad)**: le añade al saldo una cantidad de dinero.
- Métodos **set** y **get** para todas los atributos.
- **consultarCuenta**: visualizará todos los datos de la cuenta.
- **saldoNegativo**: devolverá un valor lógico indicando si la cuenta está o no en números rojos.

EJERCICIO HERENCIA y ABSTRACCION

Vamos a partir de la clase del ejercicio anterior.

Además deberá tener varios métodos abstractos:

- **intereses():** calcula los intereses producidos por la cuenta, este método tiene la particularidad **de no tener cuerpo**, porque la idea es proporcionar métodos que puedan ser redefinidos en la subclase y adaptarlos a las necesidades de estas.
- **saldo:**

Define las subclases de Cuenta que se describen a continuación:

- **CuentaCorriente**: La clase CuentaCorriente tiene 2 nuevos atributos, además de todos los atributos heredados de la clase Cuenta:

- Private **$numTransacciones**; //número de transacciones efectuadas sobre la cuenta
- Private **$importetransacciones**; //almacena el importe que la entidad bancaria cobra por cada transacción.

- **CuentaAhorro:** Esta cuenta tiene como atributos un **saldo mínimo** necesario. Al retirar dinero hay que tener en cuenta que no se sobrepase el saldo mínimo. Además tiene una **cuota de mantenimiento** que cobra el banco por el manejo de la cuenta, y que calcule el **interés mensual** (**Saldo()*TipoInteres())/100;)**producido por la cuenta de ahorros, el **saldo** después de cobrar el valor de la comisión mensual que descuenta el banco por el mantenimiento de la cuenta de ahorros (**Saldo()-CuotaMantenimiento()**) y el **saldo neto** de la cuenta de ahorros (**Saldo() - CuotaMantenimiento () + intereses()**).

- Incluir en ambas clases:
  - constructor con parámetros.
  - método **Visualizar_datos()** el cual me mostrará por pantalla los datos de un objeto

Crea un programa que cree varias cuentas y pruebe sus características.

Ejercicio 6: Clase Trabajador

Crear las clases correspondientes teniendo en cuenta lo siguiente.

- Los **impuestos** de un empleado son el 30% del sueldo.
- El **sueldo** de un empleado se calcula:(sueldo-impuestos)/14
- El **sueldo** de un consultor se calcula: horas*tarifa.

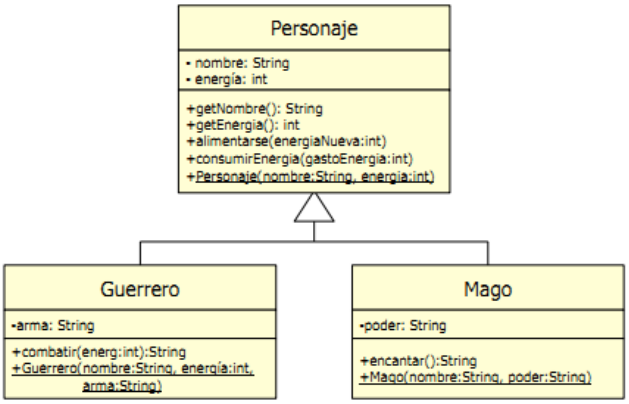
Nota: Método **toString** es para visualizar los datos de un objeto.

Utiliza clases y métodos **abstractos**

Ejercicio 7: Videojuego

Un videojuego tiene Personajes. Cada personaje tiene un **nombre** (String) y un **nivel** propio de energía (int). Además implementan el método alimentarse, que recibe por parámetro una cantidad de energía (int) con el que incrementa el nivel propio de energía y el método consumirEnergia. Los personajes pueden ser:

- **Guerreros**: tienen además un **arma** (String). Al momento de la instanciación reciben su nombre, arma y nivel propio de energía inicial. Los guerreros tienen un método combatir que recibe por parámetro la cantidad de energía a gastar en el ataque, la cual es descontada de su nivel propio de energía. El método combatir retorna el arma y la cantidad de energía del ataque concatenados.
- **Magos**: tienen además un **poder** (String). Al momento de la instanciación reciben su nombre y poder. Los magos son siempre creados con un nivel propio de energía igual a 100. Proveen un método encantar, que disminuye en 2 unidades el nivel propio de energía y que retorna el poder del mago.



OJO→ en PHP tener en cuenta que solo puede haber **un** constructor (es el método que se llama igual que las clases), por tanto lo mejor es en el constructor o bien no pasarle ningún parámetro o pasarle todos.

Ejercicio 8- Parques Naturales. Clases Abstractas y Arrays

Objetivo

Saber identificar e implementar clases abstractas
Identificar en un enunciado clases y relacionarlas entre sí, a través de la herencia.

Enunciado

Se pide desarrollar un programa PHP que permita gestionar todos los parques nacionales que existen en Kenia. Cada parque posee una extensión (**km cuadrados**), un **número de especies** de animales y poseen un **nombre**. Todos ellos sólo pueden ser de dos tipos, **las Reservas de**

Caza o **Áreas Protegidas**. Los primeros, tienen un **coste de licencia** y un sólo **tipo de arma** a utilizar en él, mientras que los segundos, poseen una **subvención del gobierno** (en Chelines) y colabora en ellos una **ONG**. Dentro de las Áreas Protegidas, se diferencian entre las **acuáticas** (**número de lagos**), las **terrestres** (**tipo de terreno**) y el resto, que no se clasifican bajo ningún tipo específico.

Además todos los parques deben tener un método en común que se llame visitable, el cual tendrá devolverá Verdadero o Falso en función de si el Parque puede recibir visitas o no. En principio solo las Reservas de Caza pueden recibir visitas.

Se pide:

- Crear 2 Reservas de Caza y 3 Áreas Protegidas (una de cada), insertándolas en un array.
- Mostrar toda la información de los parques por consola recorriendo dicho array.
- Mostrar sólo el nombre de los parques almacenados.

EJERCICIOS BASICOS DE POO

EJERCICIO 1

Programar una clase de nombre **CuentaCorriente** que permita almacenar e interactuar con la cuenta bancaria de un cliente. Esta clase tendrá las siguientes propiedades, métodos y constructores:

Propiedades (atributos) privadas:

- nombre, apellidos, dirección, teléfono, NIF Y saldo.

Constructor:

- Constructor al que se le pasen como argumentos todas las propiedades que tiene la clase

Métodos públicos:

- **sacarDinero**: le resta al saldo una cantidad de dinero pasada como argumento.
- **ingresarDinero**: le añade al saldo una cantidad de dinero.
- **consultarSaldo**, **consultarNombre**……. devolverá cada uno de ellos el valor de su propiedad.
- **establecerSaldo**, **establecerNombre**….. asignará cada uno de ellos el valor de su propiedad.
- **consultarCuenta**: visualizará todos los datos de la cuenta.
- **saldoNegativo**: devolverá un valor lógico indicando si la cuenta está o no en números rojos.

EJERCICIO 2

En esta práctica de laboratorio vamos a continuar afianzando los conceptos de clases y objetos, repitiendo conceptos aprendidos la semana pasada. La definición de clases y objetos es una de las partes más importantes a la hora de desarrollar programas en PHP.

Ejercicio 1. Clases y Creación de Objetos

Clase `Album`

La clase `Album` va a representar una producción musical de un determinado cantante o grupo, que se caracteriza por los siguientes atributos:

- **titulo**: el título del álbum
- **autor**: el nombre del cantante o grupo
- **discografica**: el nombre de la discográfica
- **año**: año de edición
- **soporte**: un carácter que indica el soporte: 'C' (CD), 'D' (disco), 'S' (casete), 'V' (DVD), 'M' (minidisc)
- **elementos**: el número de CDs, DVDs, etc que incluye el álbum
- **precio**: el precio recomendado de venta al público, sin incluir IVA
- **genero**: una letra que indica el tipo de música: 'D' (dance), 'P' (pop), 'R' (rock)
- **español**: true/false si el álbum es de un cantante o grupo español

Razona de qué tipo sería cada uno de los atributos (¡y los modificadores de acceso!). Luego, escribe la clase `Album` en PHP.

Clase `PruebaAlbum`

Ahora programa una fichero llamado `PruebaAlbum.php`, que sirve para probar cómo funciona la clase `Album`. La clase de prueba debe realizar lo siguiente:

1. Crear dos objetos de tipo `Album`, llamados, por ejemplo, `album1` y `album2`
2. Dar valor a los atributos de ambos objetos con los datos de dos álbumes que te gusten
3. Imprimir los datos de cada álbum correctamente tabulados, como se presentan a continuación:

```
Datos del álbum
Título:          Crazy Hits
Autor:          Crazy Frog
Discográfica:    Blanco y Negro
Año:            2005
Soporte (num):  C (1)
Precio:         14.99
Género:         D
Español:        false
```

El código que resulta es bastante repetitivo. ¿Sabrías hacerlo más simple escribiendo un método `imprimirAlbum` que imprima lo anterior y que se le pueda llamar para cada objeto de tipo `Album`? ¿En qué clase iría ese método?

Ejercicio 2. Métodos de acceso

Añade al programa métodos para rellenar y leer todos los campos de la clase

Modifica el `index` anterior para que utilice los métodos definidos para rellenar los campos de los objetos. Comprueba que imprime el resultado correcto.

Ejercicio 3. Operadores

Modifica el programa anterior para que, además de lo anterior, el programa calcule el precio total de los dos álbumes, imprimiendo un mensaje en pantalla:

```
      Precio total (sin IVA):     29,98 euros
      Precio total (con 16% IVA): 34,7768 euros
```

(Nota: lógicamente, el precio total depende del precio que hayas puesto a tus objetos).

Ejercicio 4. Constructores

Hasta ahora no se ha escrito ningún constructor para la clase `Album` de los ejercicios anteriores. Así, después de crear un objeto de la clase `Album`, hemos llamado a los métodos `ponTitulo`, `ponAutor`, etc. para asignar valor a los atributos del objeto creado. Esto, como sabes de la práctica anterior, tiene el inconveniente de que uno puede olvidarse en el programa de rellenar algún atributo lo que dejaría un objeto `Album` con valores inconsistentes.
Debes escribir un constructor para la clase Album que rellene todos los atributos del objeto. Para ello, debéis pasarle al constructor una lista de parámetros para rellenar todos los atributos del objeto. Esto nos permitirá llamar al constructor de la siguiente forma:

```
      new Album("Crazy Hits","Crazy Frog","Blanco y
Negro",2005,'C',1,14.99,'D',false)
```

... para crear un objeto de la clase Album.