Desarrollo Web en entorno servidor
Unidad 3:
Aplicaciones Web en PHP
Programación orientada a objetos.

Tabla de contenido

1. Programación orientada a objetos

Esta primera parte del tema reproduce el capítulo dedicado a la programación orientada a objetos del libro APRENDER A DESARROLLAR UN SITIO WEB CON PHP Y MYSQL (2ª ED.) de Olivier Rollet, publicado en 2015por el Editorial ENI.

La Programación Orientada a Objetos, en adelante POO, es un paradigma de programación según el cual los sistemas a desarrollar se modelan creando clases. Las clases son las definiciones de un conjunto de datos (atributos) y funciones (métodos) y nos permiten crear objetos.

Los objetos son ejemplares de una clase determinada y como tal, disponen de los datos (atributos) y funciones (métodos) definidos.

Pongamos un ejemplo, pensemos en un animal cualquiera. De todos los animales podemos distinguir ciertos atributos como su color o su peso. Por otra parte, los animales pueden hacer ciertas funciones como moverse o comer.

Por tanto, podemos definir la clase “Animal”, la cual tendrá por atributos el color y el peso, y como métodos moverse y comer.

Definida una clase, podemos pensar en los objetos de dicha clase.

Dos objetos de la clase “Animal” son, por ejemplo, el objeto “perro” y el objeto “gato”.

Cuando se crean ejemplares de animales en la clase Animal, decimos que se crea una instancia de esta clase. Crear una instancia de una clase significa que se crea un objeto (“perro” o “gato”) de un tipo determinado (“Animal”) con ciertos atributos (“color”, “peso”) y ciertos métodos (“comer”, “moverse”)

CLASE	ANIMAL
ATRIBUTOS	Color Peso
MÉTODOS	Comer() Moverse()

Tabla 1 - Clase animal

1.1. Creación de una clase en PHP

El código en PHP para crear una clase es el siguiente.

```
<?php
class Animal // palabra clave class seguida del nombre de la clase.
{
// Declaración de atributos y métodos.
}
```

Es recomendable crear una clase por cada archivo PHP que tenga el mismo nombre que la clase.

1.2. La encapsulación

Todos los atributos en POO deben estar ocultos para otras personas que utilizan las clases que definimos. Si estoy trabajando en equipo y creo la clase Animal, los otros programadores no deben poder cambiar directamente los atributos de mi clase.

De esta forma, los atributos color y peso se ocultan en otras clases; se declaran privadas. La clase Animal tiene métodos para leer o escribir en estos atributos. Este es el principio de encapsulación, el cual permite proteger el código cuando se trabaja en equipo.

La clase Animal, que tiene como propiedades el color y el peso, a de disponer de un método para modificar su color, un método para leer el color, un método para modificar su peso, un método para leer su peso, así como otros métodos tales como comer o moverse.

1.3. Visibilidad de los atributos y de los métodos

Hay tres tipos de palabra clave para definir la visibilidad de un atributo o de un método:

- private: solo el código de su clase puede ver y acceder a este atributo o método.
- public: todas las demás clases pueden acceder a este atributo o método.
- protected: solo el código de su clase y de sus subclases pueden acceder a este atributo o método.

Vamos a crear la clase en PHP con sus atributos:

```php
<?php
class Animal // palabra clave seguida del nombre de la clase.
{
// Declaración de atributos.
private $color;
private $peso;
}
```

Podemos definir los valores por defecto de sus atributos:

```php
<?php
class Animal // palabra clave seguida del nombre de la clase.
{
// Declaración de atributos.
private $color = "gris";
private $peso = 10;
}
```

Añadimos los métodos a la clase, las normas de visibilidad son las mismas que en los atributos:

```php
<?php
class Animal // palabra clave seguida del nombre de la clase.
{
    // Declaración de atributos y métodos.
    private $color = "gris";
    private $peso = 10;
    public function comer()
    {
```

```
        //Método que puede acceder a las propiedades color y peso
    }
    public function moverse()
    {
        //Método que puede acceder a las propiedades color y peso
    }
}
```

1.4. Añadir un método en la clase

Añadir el código de un método a la clase significa aplicar la clase. Para acceder a los atributos de la clase, debemos utilizar la pseudovariable $this, que representa el objeto sobre el que va a escribir.

Para acceder al atributo o al método del objeto, utilizamos el operador ->.

Por ejemplo, vamos a añadir el código del método añadir_un_kilo() en la clase Animal:

```
<?php
class Animal // palabra clave seguida del nombre de la clase.
{
    // Declaración de atributos y métodos.
    private $color = "gris";
    private $peso = 10;
    public function comer()
    {
        //Método que puede acceder a las propiedades color y peso
    }
    public function moverse()
    {
        //Método que puede acceder a las propiedades color y peso
    }
    public function añadir_un_kilo()
    {
        $this->peso = $this->peso + 1;
    }
}
```

Cuando llamamos al método añadir_un_kilo(), añadiremos 1 al peso actual y por lo tanto el peso final será 11.

Las propiedades se declaran con el símbolo `$`, pero se llaman con `$this` sin este símbolo.

1.5. Utilización de la clase

Como primer paso, tenemos que crear un archivo que contenga el código PHP de nuestra clase Animal.class.php.

Para utilizar la clase Animal, debemos incluirla en la página donde la queramos llamar.

Vamos a crear una página uso.php y en ella escribimos el siguiente código:

```
<?php
//carga de la clase
```

```php
include('Animal.class.php');
//instanciar la clase Animal
$perro = new Animal();
?>
```

Vemos como hemos cargado la clase y la hemos instanciado, es decir, hemos creado un objeto que tiene como modelo la clase Animal:

La variable $perro es una instancia de la clase Animal, con los atributos propios de color, peso, y como métodos comer, moverse, añadir_un_kilo.

1.6. Actualizar y leer los atributos de la instancia

El principio de encapsulación requiere que todos los atributos sean privados. Por lo tanto, debemos crear métodos públicos que permitan leer o escribir en sus atributos desde otra página PHP.

Estos métodos se denominan **accesos** y, generalmente, sus nombres van precedidos del prefijo **get** para leer el valor del atributo y **set** para escribir el valor del atributo.

La clase Animal con los accesos es:

```php
<?php
class Animal // palabra clave seguida del nombre de la clase.
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    //accesos
    public function getColor()
    {
        return $this->color; //devuelve el color
    }
    public function setColor($color)
    {
        $this->color = $color; //escrito en el atributo color
    }
    public function getPeso()
    {
        return $this->peso; //devuelve el peso
    }
    public function setPeso($peso)
    {
        $this->peso = $peso; //escrito en el atributo peso
    }
    //métodos
    public function comer()
    {
        //Método que puede acceder a las propiedades color y peso
    }
    public function moverse()
    {
```

```
        //Método que puede acceder a las propiedades color y peso
    }
    public function añadir_un_kilo()
    {
        $this->peso = $this->peso + 1;
    }
}
```

Los accesos son públicos y por lo tanto permiten leer o escribir en los atributos desde cualquier otra clase o página PHP.

Ejemplo con la página uso.php:

```
<?php
//carga de la clase
include('Animal.class.php');
//instanciar la clase Animal
$perro = new Animal();
//leer el peso
echo "El peso del perro es:".$perro->getPeso().parent
//añadir un kilo al perro
$perro->añadir_un_kilo();
//leer el peso
echo "El peso del perro es:".$perro->getPeso().parent
//actualizar el peso del perro
$perro->setPeso(15);
//leer el peso
echo "El peso del perro es:".$perro->getPeso().parent
?>
```

Da como resultado:

El peso del perro es:10 kg
El peso del perro es:11 kg
El peso del perro es:15 kg

En efecto, el peso del perro se inicializa a 10. A continuación el método añadir_un_kilo() añade 1; por lo tanto, su peso queda como sigue 11. Para terminar, el método setPeso(15) ajusta el peso a 15.

Podemos crear tantas instancias de una clase como queramos.

Por ejemplo, para crear un gato blanco de 5 kg y un perro negro de 18 kg:

```
<?php
//carga de la clase
include('Animal.class.php');
//instanciar la clase Animal
$perro = new Animal();
//actualizar el peso del perro
$perro->setPeso(18);
//leer el peso
```

```
echo "El peso del perro es:".$perro->getPeso().parent
//actualizar el color del perro
$perro->setColor("negro");
//leer el color
echo "El color del perro es:".$perro->getColor()."<br>";
//instanciar la clase Animal
$gato = new Animal();
//actualizar el peso del gato
$gato->setPeso(5);
//leer el peso
echo "El peso del gato es:".$gato->getPeso().parent
//actualizar el color del gato
$gato->setColor("blanco");
//leer el color
echo "El color del gato es:".$gato->getColor()."<br>";
?>
```

Da como resultado:

El peso del perro es:18 kg
El color del perro es:negro
El peso del gato es:5 kg
El color del gato es:blanco

1.7. Paso como argumento de tipo objeto

Los métodos son como las funciones, pueden tomar argumentos de tipos diferentes (Integer, String, …) e incluso de tipo Objeto.

`$gato` y `$perro` son objetos de tipo Animal. Pueden pasar como argumento un método, siempre y cuando acepte este tipo de objeto.

Para probar este ejemplo, cambiamos el método `comer()` de la clase Animal por `comer_animal(Animal $animal_comido)`. El método toma ahora como argumento un objeto de tipo Animal.

La página Animal.class.php queda como sigue:

```
<?php
class Animal
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    //accesos
    public function getColor()
    {
        return $this->color; //devuelve el color
    }
    public function setColor($color)
    {
        $this->color = $color; //escrito en el atributo color
```

```
    }
    public function getPeso()
    {
        return $this->peso; //devuelve el peso
    }
    public function setPeso($peso)
    {
        $this->peso = $peso; //escrito en el atributo peso
    }
    //Métodos
    public function comer_animal(Animal $animal_comido)
    {
        //el animal que come aumenta su peso tanto como el del animal comido
        $this->peso = $this->peso + $animal_comido->peso;
        //el peso del animal comido y su color se restablecen a 0
        $animal_comido->peso = 0;
        $animal_comido->color = "";
    }
    public function moverse()
    {
        //método que pueda acceder a las propiedades color y peso
    }
    public function añadir_un_kilo()
    {
        $this->peso = $this->peso + 1;
    }
}
```

Para probar este método, la página uso.php queda como sigue:

```
<?php
//carga de la clase
include('Animal.class.php');
//instanciar la clase Animal
$gato = new Animal();
//actualizar el peso del gato
$gato->setPeso(8);
//leer el peso
echo "El peso del gato es:".$gato->getPeso().parent
//actualizar el color del gato
$gato->setColor("negro");
//leer el color
echo "El color del gato es:".$gato->getColor()."<br>";
//instanciar la clase Animal
$pez = new Animal();
//actualizar el peso del pez
$pez->setPeso(1);
//leer el peso
echo "El peso del pez es:".$pez->getPeso().parent
//actualizar el color del pez
```

```
$pez->setColor("blanco");
//leer el color
echo "El color del pez es:".$pez->getColor()."<br><br>";
//el gato come al pez
$gato->comer_animal($pez);
//leer el peso
echo "El nuevo peso del gato es:".$gato->getPeso().parent
//leer el peso
echo "El peso del pez es:".$pez->getPeso().parent
//leer el color
echo "El color del pez es:".$pez->getColor()."<br><br>";
?>
```

Da como resultado:
El peso del gato es:8 kg
El color del gato es:negro
El peso del pez es:1 kg
El color del pez es:blanco
El nuevo peso del gato es:9 kg
El peso del pez es:0 kg
El color del pez es:

El objeto `$gato` llama al método `comer_animal ($pez)` y pasa como argumento el objeto de tipo Animal `$pez`. Es decir, el objeto `$pez` con sus atributos y sus métodos se pasan como argumento. Esto permite pasar como argumento varios valores con un único parámetro. El método `comer_animal(Animal $animal_comido)` solo acepta un argumento de tipo Animal.

Por lo tanto, no puede llamar al método de la siguiente manera:

```
$gato->comer_animal("Rana");
```

O de esta manera:

```
$gato->comer_animal(4);
```

Ya que los tipos "Rana" (String) y 4 (Integer) no son de tipo Animal.

1.8. El constructor

El constructor, como su nombre indica, sirve para construir un objeto de la clase que necesitemos. En los ejemplos anteriores, cuando escribíamos new Animal(), por defecto llamábamos al constructor de la clase Animal.

Podemos crear nuestros propios constructores y así pasar como argumento el valor de los atributos que deseamos asignar a su objeto.

El constructor se designa como __construct y no tiene return. Tanto __construct como otros métodos “mágicos” llevan un doble guion bajo delante.

Para añadir un constructor que toma como argumentos el peso y el color, la página Animal.class.php queda como.

```
<?php
class Animal
```

```php
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    public function __construct($color, $peso)
    //Constructor que solicita 2 argumentos.
    {
        echo 'Llamar al constructor.<br>';
        $this->color = $color; // Inicialización del
        // color.
        $this->peso = $peso; // Inicialización del peso.
    }
    //accesos
    public function getColor()
    {
        return $this->color; //devuelve el color
    }
    public function setColor($color)
    {
        $this->color = $color; //escrito en el atributo color
    }
    public function getPeso()
    {
        return $this->peso; //devuelve el peso
    }
    public function setPeso($peso)
    {
        $this->peso = $peso; //escrito en el atributo peso
    }
    //Métodos
    public function comer_animal(Animal $animal_comido)
    {
        //el animal que come aumenta su peso tanto
        //como el del animal comido
        $this->peso = $this->peso + $animal_comido->peso;
        //el peso del animal comido y su color se restablecen a 0
        $animal_comido->peso = 0;
        $animal_comido->color = "";
    }
    public function moverse()
    {
        //método que pueda acceder a las propiedades
        //color y peso
    }
    public function añadir_un_kilo()
    {
        $this->peso = $this->peso + 1;
    }
}
```

Llamamos al constructor en la página uso.php:

```php
<?php
//carga de la clase
include('Animal.class.php');
//instanciar la clase Animal con su constructor
$perro = new Animal("beige",7);
//leer el peso
echo "El peso del perro es:".$perro->getPeso().parent
//leer el color
echo "El color del perro es:".$perro->getColor()."<br>";
//actualizar el color del perro
$perro->setColor("negro");
//leer el color
echo "El color del perro es:".$perro->getColor()."<br>";
?>
```

Da como resultado:

Llamar al constructor.
El peso del perro es:7 kg
El color del perro es:beige
El color del perro es:negro

Se muestra en primer lugar "Llamar al constructor", ya que la instrucción echo que se ha escrito en el constructor __construct de su clase Animal se llama cada vez que ejecuta new Animal().

El constructor toma como argumento los valores de sus atributos. Esto evita llamar los métodos setColor() y setPeso().

En PHP no se puede declarar dos constructores en la misma clase con la palabra mágica __contruct, sin embargo, podemos crear métodos estáticos que nos permitan instanciar objetos de la clase como veremos en el apartado 11.c.

1.9. El destructor

El destructor sirve para destruir el objeto con el fin de liberarlo de la memoria. Se llama automáticamente al final del script PHP o cuando se destruye el objeto.

Para destruir un objeto, utilizamos la función unset(), pasándole el objeto a destruir en la misma página donde hemos creado el objeto.

Podemos modificarlo si añadimos la función __destruct() en la clase.

Ejercicio 1

Crea dos peces en la página uso.php:
pez1, gris, 10 kg
pez2, rojo, 7 kg

Muestra su peso y a continuación haz que el pez1 se coma al pez2.

Vuelve a mostrar su peso.

1.10. Las constantes de clase

Una constante de clase es similar a una constante normal, es decir, un valor asignado a un nombre y que no cambia nunca. Recordemos que las constantes normales se crean con la función define()

```
define('PI',3.1415926535);
```

Una constante de clase representa una constante pero que está unida a dicha clase.

Por ejemplo, si creamos un animal con el constructor `__construct($color, $peso)`, no podemos saber inmediatamente que el $peso se corresponde con el peso del animal, ya que la instancia de la clase se realiza asignando un número, por ejemplo el 10.

```
$perro = new Animal("gris",10);
```

Para solventar esto, podemos usar constantes que representen, cada una, un peso distinto:

```
const PESO_LIGERO = 5;
const PESO_MEDIO = 10;
const PESO_PESADO = 15;
```

Las constantes siempre están en mayúsculas, sin el símbolo $ y precedidas de la palabra clave const.

La clase Animal.class.php queda como sigue:

```
<?php
class Animal
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    //constantes de clase
    const PESO_LIGERO = 5;
    const PESO_MEDIO = 10;
    const PESO_PESADO = 15;

  …

?>
```

Para llamar a estas constantes desde la página uso.php, la sintaxis es algo peculiar, ya que debemos escribir **::** entre la clase y su constante:

```
<?php
//carga de la clase
include('Animal.class.php');
//instanciar la clase Animal con su constructor
$pez1 = new Animal("gris",Animal::PESO_MEDIO);
$pez2 = new Animal("rojo",Animal::PESO_LIGERO);
//leer el peso
echo "El peso del pez1 es:".$pez1->getPeso().parent
//leer el peso
```

```
echo "El peso del pez2 es:".$pez2->getPeso().parent
//el pez1 se come al pez2
$pez1->comer_animal($pez2);
//leer el peso
echo "El nuevo peso del pez1 es:".$pez1->getPeso().parent
//leer el nuevo peso
echo "El nuevo peso del pez2 es:".$pez2->getPeso().parent
?>
```

Da como resultado:

Llamada al constructor.
Llamada al constructor.
El peso del pez1 es:10 kg
El peso del pez2 es:5 kg
El nuevo peso del pez1 es:15 kg
El nuevo peso del pez2 es:0 kg

`Animal::PESO_MEDIO` siempre es 10, sea cual sea la instancia. Por lo tanto, la constante no está unida a la instancia, sino a la clase. Por este motivo la sintaxis es peculiar.

1.11. Los atributos y métodos estáticos

a. Método estático

Un método estático está unido a la clase, pero no al objeto.

Para convertir un método a estático, debemos añadir la palabra clave static delante de function.

Modificamos el método moverse() para convertirlo en estático y mostrar "El animal se mueve.".

La clase Animal.class.php queda como sigue:

```
<?php
class Animal
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    //constantes de clase
    const PESO_LIGERO = 5;
    const PESO_MEDIO = 10;
    const PESO_PESADO = 15;
    public function __construct($color, $peso)
    // Constructor que solicita 2 argumentos.
    {
        echo 'Llamada al constructor.<br>';
        $this->color = $color; // Inicialización del color.
        $this->peso = $peso; // Inicialización del peso.
    }
    //accesos
```

```
    public function getColor()
    {
        return $this->color; //devuelve el color
    }
    public function setColor($color)
    {
        $this->color = $color; //escrito en el atributo color
    }
    public function getPeso()
    {
        return $this->peso; //devuelve el peso
    }
    public function setPeso($peso)
    {
        $this->peso = $peso; //escrito en el atributo peso
    }
    //métodos
    public function comer_animal(Animal $animal_comido)
    {
        //el animal que come aumenta su peso tanto como
        //el del animal comido
        $this->peso = $this->peso + $animal_comido->peso;
        //el peso del animal comido y su color se restablecen a 0
        $animal_comido->peso = 0;
        $animal_comido->color = "";
    }
    public static function moverse()
    {
        echo "El animal se mueve.";
    }
    public function añadir_un_kilo()
    {
        $this->peso = $this->peso + 1;
    }
}
```

Es imposible escribir en un método estático la palabra clave `$this`, ya que esta palabra representa al objeto, y el método estático está unido a la clase.

Por tanto, para llamar a este método desde la página uso.php, debemos utilizar la misma sintaxis que en las constantes (también unidas a la clase), es decir, introducir **::** entre la clase y su método estático:

```
<?php
//carga de la clase
include('Animal.class.php');
//llamada al método estático
Animal::moverse()
?>
```

Da como resultado:

El animal se mueve.

Podemos llamar al método estático desde un objeto, pero el método estático no puede cambiar nada de este objeto:

```php
<?php
//carga de la clase
include('Animal.class.php');
//instanciar la clase Animal con su constructor
$perro1 = new Animal("gris",Animal::PESO_MEDIO);
//llamada al método estático
$perro1->moverse();
?>
```

Da como resultado:

Llamada al constructor
El animal se mueve.

b. Atributo estático

Un atributo estático es un atributo propio de la clase y no del objeto, al igual que en los métodos estáticos. **Es el mismo principio que en una constante, salvo que el atributo está en una variable y puede cambiar su valor**.

Un atributo estático se escribe añadiendo la palabra clave static delante de su nombre.

Por ejemplo, para añadir un atributo estático que represente un contador que indica el número de veces que se instancia la clase, la clase Animal.class.php queda como sigue:

```php
<?php
class Animal
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    //constantes de clase
    const PESO_LIGERO = 5;
    const PESO_MEDIO = 10;
    const PESO_PESADO = 15;
    // Declaración de la variable estática $contador
    private static $contador = 0;

  …

?>
```

Para cambiar el valor de este contador no podemos utilizar `$this`. Ya que `$this` representa un objeto (perro, gato), y no la clase Animal. El contador es de tipo estático y por lo tanto está unido a la clase. Para llamar a este atributo en la clase, debemos utilizar la palabra clave **self**, que representa la clase.

Para añadir 1 al contador cada vez que vayamos a instanciar la clase Animal, debemos modificar el constructor. A continuación debemos añadir un método que permita leer este atributo privado con ayuda de un método de tipo public static y getContador().

La clase Animal.class.php queda como sigue:

```php
<?php
class Animal
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    //constantes de clase
    const PESO_LIGERO = 5;
    const PESO_MEDIO = 10;
    const PESO_PESADO = 15;
    // Declaración de la variable estática $contador
    private static $contador = 0;
    public function __construct($color, $peso) // Constructor
    //que solicita 2 argumentos.
    {
        echo 'Llamada al constructor.<br>';
        $this->color = $color; // Inicialización del color.
        $this->peso = $peso; // Inicialización del peso.
        self::$contador = self::$contador + 1;
    }
    // método estático que devuelve el valor del contador
    public static function getContador()
    {
        return self::$contador;
    }

  …

?>
```

La página uso.php:

```php
<?php
//carga de la clase
include('Animal.class.php');
//instanciar la clase Animal
$perro1 = new Animal("rojo",10);
//instanciar la clase Animal
$perro2 = new Animal("gris",5);
//instanciar la clase Animal
$perro3 = new Animal("negro",15);
//instanciar la clase Animal
$perro4 = new Animal("blanco",8);
//llamada al método estático
echo "Número de animales que se han instanciado:".Animal::getContador();
?>
```

Da como resultado:

Llamada al constructor.
Llamada al constructor.
Llamada al constructor.
Llamada al constructor.
Número de animales que se han instanciado:4

c. Declarando un método estático para crear objetos

Como comentábamos en la sección 11.8, no podemos hacer uso de __construct para declarar dos o más constructores. Sin embargo, sí podemos aprovecharnos de los métodos estáticos para instanciar la clase.

El siguiente método es un ejemplo de ello.

```
static function crea($color, $peso) {
    $instancia = new Animal();
    $instancia->peso = $peso;
    $instancia->color=$color;
    return $instancia;
}
```

En la segunda línea vemos como instanciamos la clase en la variable local $instancia. La palabra reservada new buscará el método __construct de la clase. En este ejemplo es necesario que __construct no reciba ningún argumento.

Una vez creado el objeto de la clase Animal en $instancia, fijamos sus atributos de la manera usual. Por último, devolvemos en la sentencia return el objeto "$instancia" ya creado.

Este objeto deberemos de recoger en la página de php en la que estemos trabajando de la siguiente manera:

```
$perro=Animal::crea("Marrón",14);
```

Observamos de nuevo el uso de los **::** para acceder al método estática "crea". El cual nos devuelve un objeto de la clase Animal, el cual guardamos en la variable $perro.

2. La Herencia en PHP

La herencia es un concepto muy importante en POO. Permite reutilizar el código de una clase sin necesidad de volver a escribirlo.

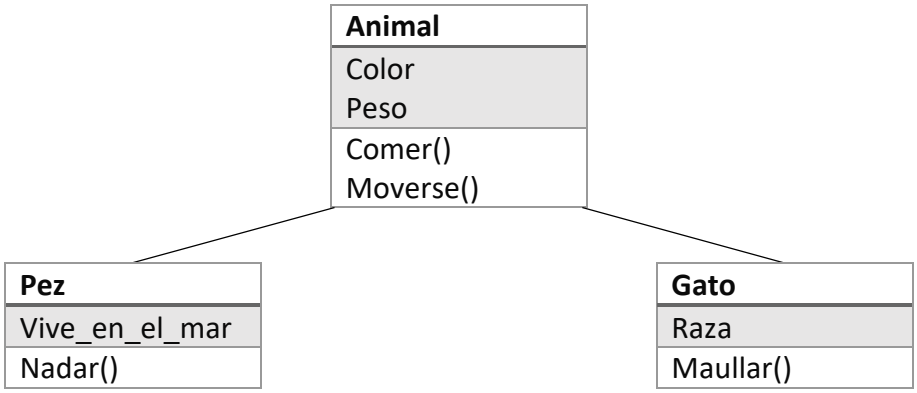
Una clase hija hereda de una clase madre, es decir, accede a todos los atributos y los métodos públicos de la clase madre.

Siguiendo con nuestro ejemplo de la clase Animal, la clase Mamífero hereda de la clase Animal.

Si la clase A es una subcategoría de la clase B, entonces podemos hacer que la clase A (Mamífero o Coche) herede de la clase B (Animal o Vehículo).

Podemos ver el uso profesional que se le da a la programación orientada a objetos a través de la documentación sobre Windows accesible en la red. En el siguiente enlace podemos acceder a la clase ListBox: ListBox Class (System.Windows.Controls) | Microsoft Docs.

Volvemos a nuestro ejemplo de los animales. En el siguiente esquema podemos observar cómo las clases Pez y Gato son ejemplos de herencia de la clase Animal.



Para crear la clase Pez que hereda de la clase Animal, debemos utilizar la palabra clave extends entre el nombre de la clase hija y el nombre de la clase madre.

Cree un archivo Pez.class.php y escribimos el siguiente código:

```
<?php
class Pez extends Animal
{
}
?>
```

Ahora añadimos un atributo privado que corresponda a la variable vive_en_el_mar y los accesos get y set. Para terminar, el método público nadar() devolverá simplemente la palabra “Nado”.

```
<?php
class Pez extends Animal
{
    private $vive_en_el_mar; //Verdadero si vive en el mar, Falso si no
    //accesos
    public function getTipo()
    {
```

```
        if ($this->vive_en_el_mar) {
            return "vive_en_el_mar";
        } else if ($this->vive_en_el_mar === false) {
            return "no_vive_en_el_mar";
        } else {
            return "";
        }
    }
    public function setTipo($vive_en_el_mar)
    {
        $this->vive_en_el_mar = $vive_en_el_mar;
        //escrito en el atributo vive_en_el_mar
    }
    //método
    public function nadar()
    {
        echo "Nado";
    }
}
```

De la misma manera, creamos un archivo para la clase Gato, Gato.class.php:

```
<?php
class Gato extends Animal
{
    private $raza; //raza del gato
    //accesos
    public function getRaza()
    {
        return $this->raza; //devuelve la raza
    }
    public function setRaza($raza)
    {
        $this->raza = $raza; //escrito en el atributo raza
    }
    //método
    public function maullar()
    {
        echo "Miau";
    }
}
```

Las clases Gato y Pez, que heredan de la clase Animal, tienen acceso a los **atributos públicos** de la clase Animal.

La página uso.php:

```
<?php
//carga de clases
include('Animal.class.php');
include('Pez.class.php');
include('Gato.class.php');
```

```php
//instanciar la clase Pez que llama al constructor de
//la clase Animal
$pez = new Pez("gris", 8);
//instanciar la clase Gato que llama al constructor de la
//clase Animal
$gato = new Gato("blanco", 4);
//leer el peso con el acceso de la clase madre
echo "El peso del pez es:" . $pez->getPeso() . " kg<br>";
//leer el peso con el acceso de la clase madre
echo "El peso del gato es:" . $gato->getPeso() . " kg<br>";
$pez->setTipo(true);
//leer el tipo con el acceso de su propia clase
echo "El tipo de pez es:" . $pez->getTipo() . "<br>";
//llamada al método de la clase Pez
$pez->nadar();
$gato->setRaza("Angora");
//leer la raza con el acceso de su propia clase
echo "La raza del gato es:" . $gato->getRaza() . "<br>";
//llamada al método de la clase gato
$gato->maullar();
//llamada al método estático
echo "Número de animales que se han instanciado:" . Animal::getContador();
```

Da como resultado:

Llamada al constructor.
Llamada al constructor.
El peso del pez es:8 kg
El peso del gato es:4 kg
El tipo de pez es:vive en el mar
Nado
La raza del gato es:Angora
Miau
Número de animales que se han instanciado:2

La clase Pez no tiene acceso a los atributos de la clase Gato y viceversa, ya que una hereda de la otra. Las clases Pez y Gato no tienen directamente acceso a los atributos privados color y peso de la clase Animal. Deben pasar por sus accesos públicos.

Ejercicio 2
La clase pez presenta un problema en la función setTipo. ¿Cuál es?

2.1. Protected

Este tipo de visibilidad equivale a private, salvo que las clases hijas puedan ver los atributos protected de la clase madre.

Por ejemplo, vamos a añadir el atributo $edad de visibilidad protegida (protected) en la clase madre Animal:

```php
<?php
class Animal
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    protected $edad = 0;

  …

?>
```

Este atributo no tiene acceso público; por lo tanto, ninguna de las otras clases hijas que heredan de Animal y de la clase Animal pueden modificarlo o leerlo.

Añadimos el método mostrarAtributos() en la clase Pez:

```php
    public function mostrarAtributos()
    {
        echo "Tipo:" . $this->vive_en_el_mar; // correcto ya que es
        // privada de esta clase
        echo "<br>";
        echo "Edad:" . $this->edad; // correcto, ya que el atributo
        // está protegido en la clase madre.
        echo  "<br>";
        echo "Peso:" . $this->peso; // error, ya que el atributo es
        //privado en la clase madre, el acceso está prohibido. Debe pasar
        //por sus accesos públicos para modificar o leer su valor
        echo  "<br>";
    }
```

Para probar este método, la página uso.php queda como sigue:

```php
<?php
//carga de clases
include('Animal.class.php');
include('Pez.class.php');
//instanciar la clase Pez que llama al constructor de
//la clase Animal
$pez = new Pez("gris",8);
//leer el peso con el acceso de la clase madre
echo "El peso del pez es:".$pez->getPeso()." kg<br>";
//actualizar el tipo de pez
$pez->setTipo(true);
//llamada al método mostrando los atributos de la clase Pez
//y Animal
$pez->mostrarAtributos();
?>
```

Da como resultado:

Llamada al constructor.
El peso del pez es:8 kg
Tipo:1

Edad:0
Warning: Undefined property: Pez::$peso
in **C:\xampp\htdocs\DWES_Prueba\Pez.class.php** on line **34**
Peso:

En conclusión, se recomienda poner los atributos en visibilidad protected si la clase va a tener subclases, ya que la propia clase, las clases hijas y las que hereden a su vez de estas tendran acceso a este atributo.

2.2. Sustitución

La Sustitución[1] sirve para modificar un método que ya existe en una clase madre, con el objetivo de cambiar su comportamiento. El método existe en dos clases diferentes y según el contexto se ejecuta el de la clase hija o el de la clase madre.

Por ejemplo, vamos a modificar el método `comer_animal(Animal $animal_comido)` de la clase Animal de forma que ahora los usemos en la clase Pez para inicializar el tipo de pez comido. Añadimos el siguiente código en la clase Pez.class.php:

```
//metodo sustituido
    public function comer_animal(Animal $animal_comido)
    {
        if (isset($animal_comido->raza)) {
            $animal_comido->raza = "";
        }
        if (isset($animal_comido->vive_en_el_mar)) {
            $animal_comido->vive_en_el_mar = "";
        }
    }
```

Este método inicializa modifica los atributos vive_ en_ el_mar y raza del pez comido. Sin embargo, ya no modifica el peso y el color del pez comido, los cuales deberían de cambiar al comer el animal.

La solución está en llamar al método `comer_animal(Animal $animal_comido)`de la clase Animal en el método `comer_animal(Animal $animal_comido)`de la clase Pez:

```
    public function comer_animal(Animal $animal_comido)
    {
        // llamaos al método comer_animal() de la clase parent,
        // es decir Animal
        parent::comer_animal($animal_comido);
        if (isset($animal_comido->raza)) {
            $animal_comido->raza = "";
        }
        if (isset($animal_comido->vive_en_el_mar)) {
            $animal_comido->vive_en_el_mar = "";
        }
    }
```

[1] No confundir con la sobrecarga de los métodos. Entendemos por sobrecarga de un método cuando definimos dos o tres métodos con el mismo nombre, pero con diferentes números o tipos de parámetro. Ver __call en la sección 3. Los métodos mágicos en PHP

`parent` es una palabra clave que designa la clase madre, es decir, la clase Animal.

Vamos a verlo en la página uso.php:

```php
<?php
//carga de clases
include('Animal.class.php');
include('Pez.class.php');
//instanciar la clase Pez que llama al constructor de la
//clase Animal
$pez = new Pez("gris", 8);
//actualizar el tipo de pez
$pez->setTipo(true);
//instanciar la clase Pez que llama al constructor de la
//clase Animal
$otro_pez = new Pez("negro", 5);
// el tipo de pez
$otro_pez->setTipo(false);
//llamada al método que muestra los atributos de la clase Pez
//y Animal, no podemos usar el método mostrarAtributos()
$pez->comer_animal($otro_pez);
//leer el tipo por el acceso de su propia clase
echo "El tipo de pez comido es:" . $otro_pez->getTipo() . "<br>";
//leer el peso por el acceso de la clase madre
echo "El peso del pez comido es:" . $otro_pez->getPeso() . " kg<br>";
```

Da como resultado:

Llamada al constructor.
Llamada al constructor.
El tipo de pez comido es:
El peso del pez comido es:0 kg

El peso se ha inicializado a 0 por el método `comer_animal(Animal $animal_comido)` de la clase Animal.

En la clase Pez, el método `comer_animal(Animal $animal_comido)` puede cambiar el atributo tipo en un objeto de tipo Animal, ya que Pez hereda de Animal y, en el archivo uso.php, es un Pez lo que pasa como argumento, y no un Animal. A esto lo denominamos polimorfismo de herencia.

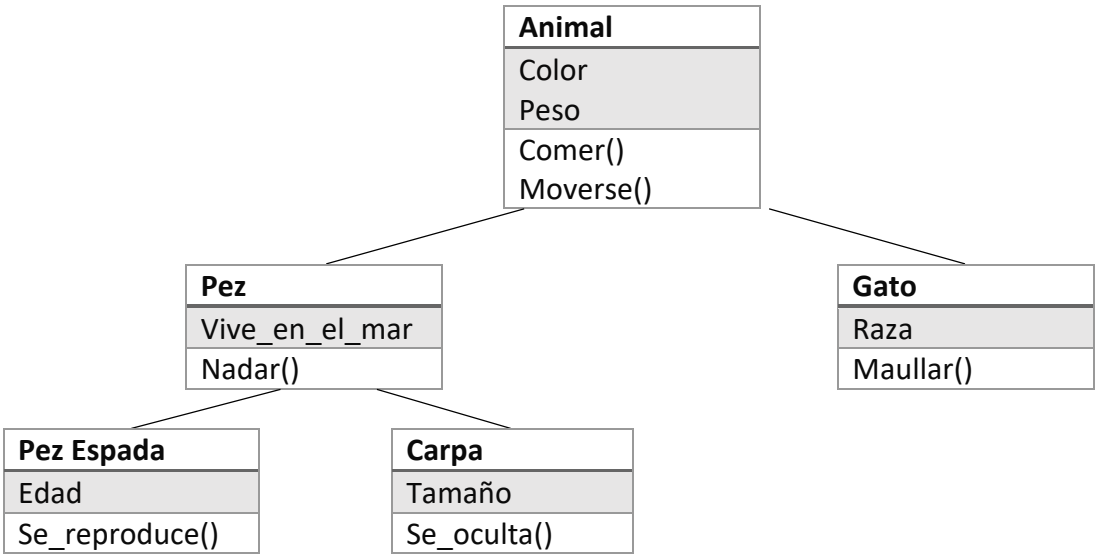
2.3. Herencia en cascada

La herencia múltiple no existe en PHP. Entendemos por herencia múltiple cuando una subclase tiene dos clases madre. En PHP una clase solo puede heredar de una única clase, que a su vez puede heredar de una clase, etc.

De esta forma, siempre tendremos una clase primigenia de la cual derivaran todas sus subclases.

Veamos un ejemplo en el que, a partir de la clase Pez, generamos dos nuevas subclases:

- Pez espada
- Carpa



La clase Pez Espada accede a:

- Todos los atributos y métodos privados, protegidos y públicos por sí misma.
- Todos los atributos y métodos protegidos y públicos de la clase Pez.
- Todos los atributos y métodos protegidos y públicos de la clase Animal.
- Todos los atributos y métodos públicos de la clase Gato.

2.4. Las clases abstractas

Las clases abstractas se escriben con la palabra clave `abstract` delante de la palabra `class`. Una clase abstracta no se puede instanciar, es decir, no permite crear un objeto perteneciente a la clase.

En las clases abstractas podemos escribir métodos abstractos, que son métodos donde solo escribe la firma precedida por la palabra clave abstract: `abstract visibilidad function nombre_método (atributo)`. Estas clases solo sirven para obligar, a las clases que heredan de la clase abstracta, a reemplazar los métodos abstractos declarados en la clase abstracta.

En el siguiente ejemplo, hacemos que la clase Animal sea abstracta, ya que no vamos a permitir crear animales, sino peces o gatos. Añadimos también un método abstracto `respira()` en la clase Animal:

```php
<?php
abstract class Animal
{
    // Declaración de atributos
    private $color = "gris";
    private $peso = 10;
    //constantes de clase
    const PESO_LIGERO = 5;
    const PESO_MEDIO = 10;
    const PESO_PESADO = 15;
    // Declaración de la variable estática $contador
    private static $contador = 0;
    public function __construct($color, $peso) // Constructor
```

```
    //que solicita 2 argumentos.
    {
        echo 'Llamada al constructor.<br>';
        $this->color = $color; // Inicialización del color.
        $this->peso = $peso; // Inicialización del peso.
        self::$contador = self::$contador + 1;
    }
    //accesos
    public function getColor()
    {
        return $this->color; //devuelve el color
    }
    public function setColor($color)
    {
        $this->color = $color; //escrito en el atributo color
    }
    public function getPeso()
    {
        return $this->peso; //devuelve el peso
    }
    public function setPeso($peso)
    {
        $this->peso = $peso; //escrito en el atributo peso
    }
    //métodos públicos
    public function comer_animal(Animal $animal_comido)
    {
        //el animal que come aumenta su peso tanto como
        //el del animal comido
        $this->peso = $this->peso + $animal_comido->peso;
        //el peso del animal comido y su color se restablecen a 0
        $animal_comido->peso = 0;
        $animal_comido->color = "";
    }
    public static function moverse()
    {
        echo "El animal se mueve.";
    }
    public function añadir_un_kilo()
    {
        $this->peso = $this->peso + 1;
    }
    // método estático que devuelve el valor del contador
    public static function getContador()
    {
        return self::$contador;
    }
    //código no aplicado por el método abstracto
    abstract public function respira();
}
```

Observe que el método abstracto `respira()` no tiene cuerpo, es decir, no hay llaves {} en la aplicación del método.

Si tenemos abiertas las páginas Pez.class.php y Gato.class.php veremos que nos arrojan un error, esto es debido a que es necesario Sustituir dentro de cada una de ellas el método `respira()` de la clase Animal.

Por tanto, las clases Pez y Gato al ser herederas de la clase Animal, están obligadas a definir de nuevo el método `respira()`.

Añadimos en la clase Pez lo siguiente:

```
public function respira()
{
    echo "El pez respira.<br>";
}
```

Y en la clase Gato:

```
public function respira()
{
    echo "El gato respira.<br>";
}
```

La página uso.php queda como sigue:

```
<?php
//carga de clases
include('Animal.class.php');
include('Pez.class.php');
include('Gato.class.php');
//instanciar la clase Pez, que llama al constructor de
//la clase Animal
$pez = new Pez("gris",8);
//instanciar la clase Gato, que llama al constructor de
//la clase Animal
$gato = new Gato("blanco",4);
//llamada al método respira()
$pez->respira();
$gato->respira();
?>
```

Da como resultado:

Llamada al constructor.
Llamada al constructor.
El pez respira.
El gato respira.

2.5. Las clases finales

Una clase puede ser declarada como final, a partir de ella no se pueden crear subclases. Para ello debemos añadir la palabra clave `final` delante de la palabra clave `class`.

Por ejemplo, si no creásemos ninguna clase que hereda de la clase Pez, podemos declararla como final:

```
final class Pez extends Animal
{
    private $vive_en_el_mar; //tipo de pez
    //accesos
    ...
    //método
    public function nadar()
    {
        echo "Nado <br>";
    }
    public function respira()
    {
        echo "El pez respira.<br>";
    }
}
```

También podemos declarar los métodos finales. Estos métodos no se podrán sustituir según los visto anteriormente. Antes hemos visto un ejemplo donde el método `comer_animal(Animal $animal_comido)`se ha sustituido en la clase Pez. Si este método fuese final en la clase Animal, entonces no habríamos podido sustituirlo en la clase Pez.

```
    final public function comer_animal(Animal $animal_comido)
    {
        //el animal que come aumenta su peso tanto como
        //el del animal comido
        $this->peso = $this->peso + $animal_comido->peso;
        //el peso del animal comido y su color se restablecen a 0
        $animal_comido->peso = 0;
        $animal_comido->color = "";
    }
```

Si tenemos la clase Pez abierta vemos cómo nos muestra un error.

Guardamos las clases

3. Los métodos mágicos en PHP

Un método mágico es un método al que se llama automáticamente cuando se produce un acontecimiento. Por ejemplo `__construct` es un método mágico. Se ejecuta automáticamente cuando se instancia una clase.

3.1. Métodos mágicos para atributos

Los métodos mágicos `__get` y `__set` permiten leer o modificar los atributos que no existen y en los que el acceso está prohibido.

Reescribimos la clase Animal. Esta vez, el atributo color es privado y el atributo peso es público:

```php
<?php
class Animal
{
    // Declaración de atributos
    private $color = "gris";
    public $peso = 10;
    public function __construct($color, $peso) // Constructor
    //que solicita 2 argumentos.
    {
        echo 'Llamada al constructor.<br>';
        $this->color = $color; // Inicialización del color.
        $this->peso = $peso; // Inicialización del peso.
    }
    //métodos públicos
    public function comer()
    {
    }
    public function moverse()
    {
    }
}
```

Cuando creamos una instancia de la clase Animal, podemos acceder al atributo `$peso` porque es público, pero no al atributo `$color` porque es privado. Veámoslo en la página uso.php.

```php
<?php
//carga de clases
include('Animal.class.php');
//instanciar la clase Animal
$perro = new Animal("gris",8);
$perro->color = "negro";
echo $perro->color."<br>";
?>
```

La salida muestra un error porque intenta acceder directamente al atributo color, que es privado:

Ahora añadimos los métodos mágicos `__get` y `__set` en la clase Animal:

```php
<?php
class Animal
{
    // Declaración de atributos
    private $color = "gris";
    public $peso = 10;
    private $tab_atributos = array();
    public function __construct($color, $peso) // Constructor
    //que solicita 2 argumentos.
    {
        echo 'Llamada al constructor.<br>';
        $this->color = $color; // Inicialización del color.
        $this->peso = $peso; // Inicialización del peso.
    }
```

```
    //métodos mágicos
    public function __get($nombre)
    {
        echo "__get <br>";
        if (isset($this->tab_atributos[$nombre]))
            return $this->tab_atributos[$nombre];
    }
    public function __set($nombre, $valor)
    {
        echo "__set <br>";
        $this->tab_atributos[$nombre] = $valor;
    }
    public function __isset($nombre)
    {
        return isset($this->tab_atributos[$nombre]);
    }
    //métodos públicos
    public function comer()
    {
    }
    public function moverse()
    {
    }
}
```

Estos métodos se activan automáticamente si el atributo es privado o no existe. Almacenan el valor en la tabla `$tab_atributos`, y tienen por índice el nombre del atributo. El método `__isset($nombre)` permite saber si existe o no el atributo.

Veámoslo en la página uso.php.

```
<?php
//carga de clases
include('Animal.class.php');
//instanciar la clase Animal
$perro = new Animal("gris", 8);
if (isset($perro->color)) {
    echo "El atributo color existe.<br>";
} else {
    echo "El atributo color no existe.<br>";
}
$perro->color = "negro";
echo $perro->color . "<br>";
if (isset($perro->peso)) {
    echo "El atributo peso existe.<br>";
} else {
    echo "El atributo peso no existe.<br>";
}
$perro->peso = 25;
echo $perro->peso . "<br>";
```

Da como resultado:

Llamada al constructor.
El atributo color no existe.
__set
__get
Negro
El atributo peso existe.
25

Vamos a ver porqué obtenemos esta salida:

`$perro = new Animal("gris", 8);` da como resultado: Llamada al constructor.

`isset($perro->color)` devuelve falso, ya que no se puede acceder al atributo color, que es privado; por lo tanto, muestra: *El atributo color no existe.*

`$perro->color = "negro";` da como resultado: `__set`. El atributo color es privado; por lo tanto, no se puede acceder a él y se llama automáticamente a `__set`, almacenando el valor en el array `$tab_atributos`.

`echo $perro->color . "<br>";` da como resultado: `__get` y negro. El atributo color siempre es privado; por lo tanto, llama automáticamente a `__get` para mostrar el color.

`isset($perro->peso)` devuelve verdadero porque el atributo peso es público, se puede acceder a él y por lo tanto muestra: el atributo peso existe.

`$perro->peso = 25;` no muestra nada porque no llama a la función `__set`. El atributo peso es público, puede acceder directamente a él.

`echo $perro->peso . "<br>";` da como resultado 25. El atributo peso es público y por lo tanto puede acceder directamente a él.

Para eliminar un atributo que el método mágico `__set` ha añadido, debemos ejecutar el método mágico `__unset($atributo)`, que eliminará el atributo de la tabla `$tab_atributos`.

Para ello, añadimos este método en la clase Animal:

```
public function __unset($nombre)
    {
        if (isset($this->tab_atributos[$nombre]))
            unset($this->tab_atributos[$nombre]);
    }
```

3.2. Métodos mágicos para métodos

Para terminar, los métodos mágicos `__call` y `__callStatic` permiten llamar a los métodos privados o que no existen. La función `method_exist()` comprueba si un método existe en un objeto. Toma como argumento el objeto y el nombre del método. Devuelve true si el método existe y false si no.

La clase Animal con un método público `comer()` y un método privado `moverse()` queda como sigue:

```
<?php
class Animal
{
```

```
    // Declaración de atributos
    private $color = "gris";
    public $peso = 10;
    private $tab_atributos = array();
    public function __construct($color, $peso) // Constructor
    //que solicita 2 argumentos.
    {
        echo 'Llamada al constructor.<br>';
        $this->color = $color; // Inicialización del color.
        $this->peso = $peso; // Inicialización del peso.
    }
    //métodos mágicos
    public function __get($nombre)
    {
        echo "__get <br>";
        if (isset($this->tab_atributos[$nombre]))
            return $this->tab_atributos[$nombre];
    }
    public function __set($nombre, $valor)
    {
        echo "__set <br>";
        $this->tab_atributos[$nombre] = $valor;
    }
    public function __isset($nombre)
    {
        return isset($this->tab_atributos[$nombre]);
    }
    public function __call($nombre, $argumentos)
    {
        echo "El método " . $nombre . " no es accesible. Sus argumentos
eran los siguientes :" . implode($argumentos, ‘, ‘) . "<br>";
        if (method_exists($this, $nombre)) {
            $this->$nombre(implode($argumentos, ‘, ‘));
        }
    }
    public static function __callStatic($nombre, $argumentos)
    {
        echo "El método estático " . $nombre . " no es accesible.
Sus argumentos eran los siguientes :" . implode($argumentos, ‘, ‘)
            . "<br>";
        if (method_exists(__CLASS__, $nombre)) {
            echo __CLASS__ . ’::’ . $nombre . ’ < br > ‘;
            self::$nombre(implode($argumentos, ‘, ‘));
        }
    }
    //método público
    public function comer()
    {
        echo "Método público comer() <br>";
    }
```

```
    //método privado
    private function moverse($lugar)
    {
        echo "Método privado moverse() <br>";
    }
}
```

La página uso.php:

```
<?php
//carga de clases
include('Animal.class.php');
//instanciar la clase Animal
$perro = new Animal("gris", 8);
$perro->comer();
$perro->moverse("París");
```

Da como resultado:

Llamada al constructor.
Método público comer()
El método moverse no es accesible. Sus argumentos eran los siguientes:París
Método privado moverse()

El código `$perro->comer();` llama al método público `comer().` Pero al método `moverse($lugar)`, que es privado, no se puede acceder directamente. Por tanto, se llama al método mágico `__call`, que comprueba que existe el método `moverse($lugar)` y llama al método `moverse($lugar)`.

Si el método `moverse($lugar)` fuese estático, entonces la sentencia `Animal::moverse("París");` escrita en uso.php llamaría al método mágico `__callStatic`.

Existen otros métodos mágicos que se pueden consultar en el manual de PHP, en particular a través del siguiente enlace: http://php.net/manual/es/language.oop5.magic.php