

## PROGRAMACION ORIENTADA A OBJETOS

### PROGRAMACION ORIENTADA A OBJETOS (POO)

Concepto igual en todos los lenguaje de programación.

La POO trata de trasladar el concepto y comportamiento de los objetos de la vida real al código de programación

El objetivo de la POO es:

- reutilizar (algo parecido a la función, pero mas complejo) el código y
- hacer la vida más fácil de la programación ,

Hay que tener en cuenta que tenemos dos tipos de lenguajes de programación

- Lenguaje orientado a objetos (JAVA, JSCRIPT , PHP)
- Lenguaje orientado a programación (antiguos)

Conclusión : La Programación Orientada a Objetos, permite facilitar la programación y reutilizar el código

Hay tres términos que se deben aprender

#### **Objeto**

Ej un coche tiene unas propiedades y unos métodos

#### **Clase**

Modelo donde se indican las características comunes de un grupo de objetos

#### **Instancia**

Dentro de que tienen las características comunes de la clase del objeto coche, tendrán características propias

Para dar nombre a las clases se sigue las mismas pautas que cuando se da nombre a las variables:

NO SIMBOLOS EXTRAÑOS

NO NOMBRES DE PROGRAMACION

NO ESPACIOS EN BLANCO

SIN EMBARGO SE PONE EL PREFIJO **CLASS** Y LUEGO EL NOMBRE DE LA CLASE (SIEMPRE PRIMERA EN MAYÚSCULA)

Se crea una instancia con **NEW**

```

7
8 <body>
9
10 <?php
11
12 class Coche{
13
14
15
16
17
18 }
19
20
21 new Coche();
22
23
24 ?>

```

Como las instancias son diferentes tendrán su propio nombre, por ejemplo en el objeto coche tendremos la instancia RENAULT, MAZDA, etc. y eso hay que indicarlo en el código

Podemos crear todas las instancias que quisiéramos

```

9
10 <?php
11
12 class Coche{
13
14
15
16
17
18 }
19
20
21 $renault=new Coche();
22
23 $mazda=new Coche();
24
25 $seat=new Coche();
26
27
28 ?>

```

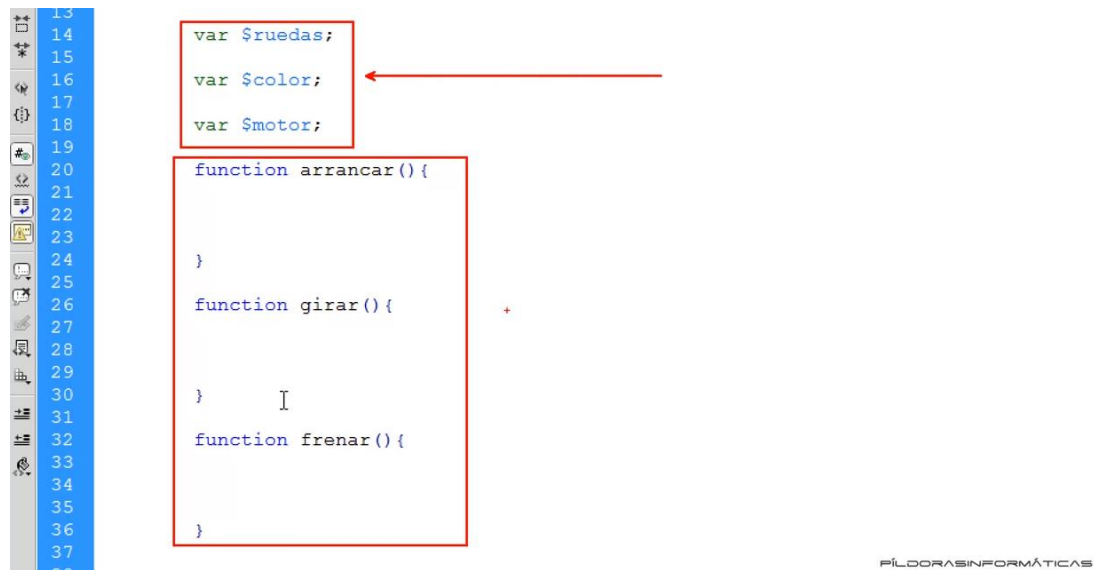
Vamos a ver ahora lo siguiente:

- Propiedades y métodos de objetos
- Cambio de propiedades
- Llamadas a métodos

Cómo dar propiedades a esos objetos creados, simplemente con variables: ruedas, color,

Método y función son sinónimos siempre y cuando la función esté dentro de una clase

Por tanto las propiedades del objeto las indicamos mediante variables y los métodos mediante funciones dentro de la clase de ese objeto



Todo objeto tiene unas características iniciales. Ej el objeto coche tiene unas características iniciales;

Km0

Sin arrancar

Un color

Un motor

ESTAS CARACTERÍSTICAS INICIALES SE CREAN MEDIANTE EL CONSTRUCTOR : se crea una función con el mismo nombre de la clase



Una vez creada la clase, al introducir el this y el -> salen todas las propiedades y funciones creadas en el objeto

Le damos las funciones

```

function arrancar(){
    echo "Estoy arrancando";
}

function girar(){
    echo "Estoy girando";
}

function frenar(){
    echo "Estoy frenando";
}

```

Recapitulando nuestro objeto coche tienen un estado inicial marcado por nuestro constructor, con 4 ruedas, sin color y motor 1600.... Y son capaces de arrancar, girar, frenar....

```

2   var $color;
3   var $motor;
4   function Coche(){
5       $this->ruedas=4;
6       $this->color= "";
7       $this->motor=1600;
8   }
9
10  function arrancar(){
11      echo "Estoy arrancando <br>";

```



Vamos a hacer ahora llamadas a un método para que una instancia haga una función

```

7   $mazda=new Coche();
8
9   $seat=new Coche();
10
11  $mazda->girar();
12
13  ?>
14  </body>
15  </html>

```

Si ejecutamos el programa ahora, indicará en el navegador "Estoy girando"

A continuación si quisiéramos acceder a una propiedad de nuestro objeto imprimimos con echo

```
60  
61  
62 $mazda->girar();  
63  
64 echo $mazda->ruedas;  
65  
66  
67 ?>
```

Por qué en la propiedad pongo echo y en el método no? Porque el método o función ya lleva el echo dentro, y la propiedad no

Y como diferenciar propiedades o métodos? La propiedad va sin paréntesis

Cómo podemos cambiar las propiedades ahora en cada instancia (por ejemplo el color en el constructor está sin definir)

1. Creamos un método o una función que reciba un parámetro(\$color\_coche), que va a ser el color y después que asigne ese parámetro a la variable color que nos lo imprima en pantalla  
esta función va asignar a la propiedad color , el parámetro que le pasemos a esta función

```
function frenar(){  
    echo "Estoy frenando<br>";  
  
function establece_color($color_coche){  
    $this->color=$color_coche;  
    echo "El color de este coche es: " . $this->color . "<br>";  
}
```

2. Como cambiar ahora el color a cada una de las instancias:

```
66 $mazda=new Coche();  
67  
68 $seat=new Coche();  
69  
70 $renault->establece_color("Rojo");  
71  
72 //$mazda->girar();  
73  
74 //echo $mazda->ruedas;  
75
```

El navegador nos devuelve ahora: "El color del coche es rojo"

Si quisiéramos indicar también el nombre del coche, pasaríamos por parámetros el segundo argumento , así

---

```
echo "Estoy frenando<br>";

}

function establece_color($color_coche,$nombre_coche){

    $this->color=$color_coche;

    echo "El color de " . $nombre_coche . " es: " . $this->color . "<br>";

}

}

$renault=new Coche(); //Estado inicial al objeto o instancia
$mazda=new Coche();
$seat=new Coche();

$renault->establece_color("Rojo","Renault");
$seat->establece_color("Azul","Seat");

//$mazda->girar();

//echo $mazda->ruedas;
```

Y nos devolvería

---

```
El color de Renault es: Rojo
El color de Seat es: Azul
```

Cómo podemos reutilizar este código para crear coches en un futuro:

1. primero borramos las instancias
2. cortamos la clase entera y la pegamos en un documento a parte, creando un archivo php , llamado vehículos php
3. En este mismo documento php copio otra vez la misma clase cambiando ahora el nombre de coche por el de camión y cambiando el constructor

```
class Camion{  
  
    var $ruedas;  
  
    var $color;  
  
    var $motor;  
  
    function Camion(){ //Método constructor;  
  
        $this->ruedas=8;  
  
        $this->color="";  
        $this->motor=2600;  
  
    }  
  
    function arrancar(){  
  
        echo "Estoy arrancando<br>";  
    }  
}
```

4. Ahora desde otro documento, queremos reutilizar el código
  - a. Primero llamamos al documento con el include
  - b. A la instancia le aplicamos el constructor que queramos, en este caso el del objeto coche
  - c. Por otro lado el método constructor puede recibir parámetros o no. (igual que la función. Cuando un método constructor no recibe parámetros no será necesario poner paréntesis.
  - d. Y también puede darse el caso de que una clase no tenga método constructor, es decir cómo si no tuviera método inicial, en ese caso es como si tuviera un constructor vacío

En el ejemplo se ha creado una instancia de cada uno de los constructores (coche y camión) y creamos un echo para que nos muestre la cantidad de ruedas que tiene cada

uno de ellos

```
<?php

include("vehiculos.php");

$mazda=new Coche();

$pegaso=new Camion();

echo "El Mazda tiene " . $mazda->ruedas . " ruedas <br>";

echo "El Pegaso tiene " . $pegaso->ruedas . " ruedas <br>";
```

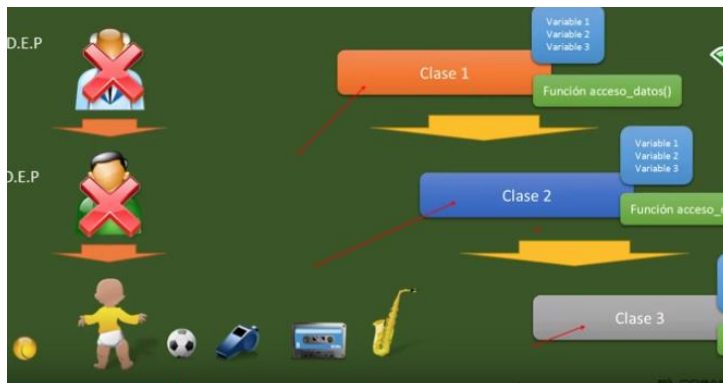
## HERENCIA EN POO

Muy útil para la reutilización de código

Teniendo en cuenta el concepto de herencia tradicional, en el que vamos heredando los objetos de nuestros antecesores



Sintaxis para conseguir en PHP el concepto de herencia. Hay que tener en cuenta que cada clase puede heredar de la anterior:

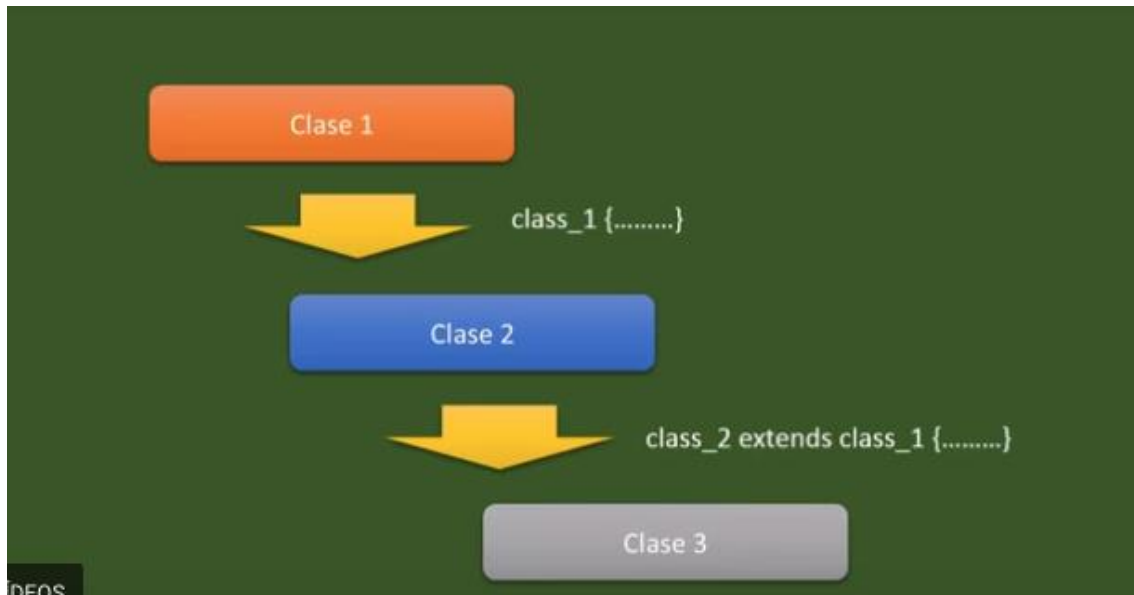


Con el prefijo extends para que herede de la anterior

Ojo: PHP NO SOPORTA LA HERENCIA MÚLTIPLE (que un elemento herede de diferentes antecesores)

Así que el class dos hereda del class uno....





En el ejemplo, si el constructor camión quiere heredar del constructor coche, pondríamos esta sintaxis

```
class Camion extends Coche{  
  
    function Camion() { //Método constructor;  
        $this->ruedas=8;  
        $this->color="gris";  
        $this->motor=2600;  
    }  
}
```

Simplemente con esta sintaxis ya tenemos en clase Camión todas las variables y los métodos de la clase Coche

Ahora al crear la instancia pegaso, saldrán todas las propiedades y los métodos de coche y camión, porque camión las ha heredado de coche:

\$pegaso->

arrancar()	Coche	vehiculos.php
color	Coche	vehiculos.php
establece_color(\$color coche, \$nombre coche)	Coche	vehiculos.php
frenar()	Coche	vehiculos.php
girar()	Coche	vehiculos.php
motor	Coche	vehiculos.php
ruedas	Coche	vehiculos.php

si elegimos la función frenar,

```
echo "El Pegaso tiene " . $pegaso->ruedas . " ruedas <br>";
$pegaso->frenar();
```

me devuelve

“Estoy frenando”

puede haber un problema en el método de “establece color”, para esto se utiliza una sobrescritura de método

```
class Camion extends Coche{

    function Camion(){ //Método constructor;

        $this->ruedas=8;

        $this->color="gris";

        $this->motor=2600;

    }

    function establece_color($color_camion,$nombre_camion){

        $this->color=$color_camion;

        echo "El color de " . $nombre_camion . " es: " . $this->color . "<br>";

    }

}
```

vamos a imaginarnos ahora que para el método arrancar, resulta que queremos que nos devuelvan cosas diferentes cuando es coche y cuando es camión. EN este caso en lugar de sobrescribir totalmente, podemos utilizar la inscripción parent, que lo que hace es llamar al método de la clase padre

```

function establece_color($color_camion,$nombre_camoin){

    $this->color=$color_camion;

    echo "El color de " . $nombre_camion . " es: " . $this->color . "<br>";

}

function arrancar(){

    parent::arrancar();

    echo "Camión arrancado";

}

```

EN PROGRAMAS COMPLEJOS DEBEMOS CREAR MODULARIZACIÓN. ES decir dividir el código en módulos o en partes(es decir clases)

Podemos crear un programa complejo en PHP en una única clase o podemos dividir el código en pequeñas partes o clases que estén relacionadas y funcionen como una unidad

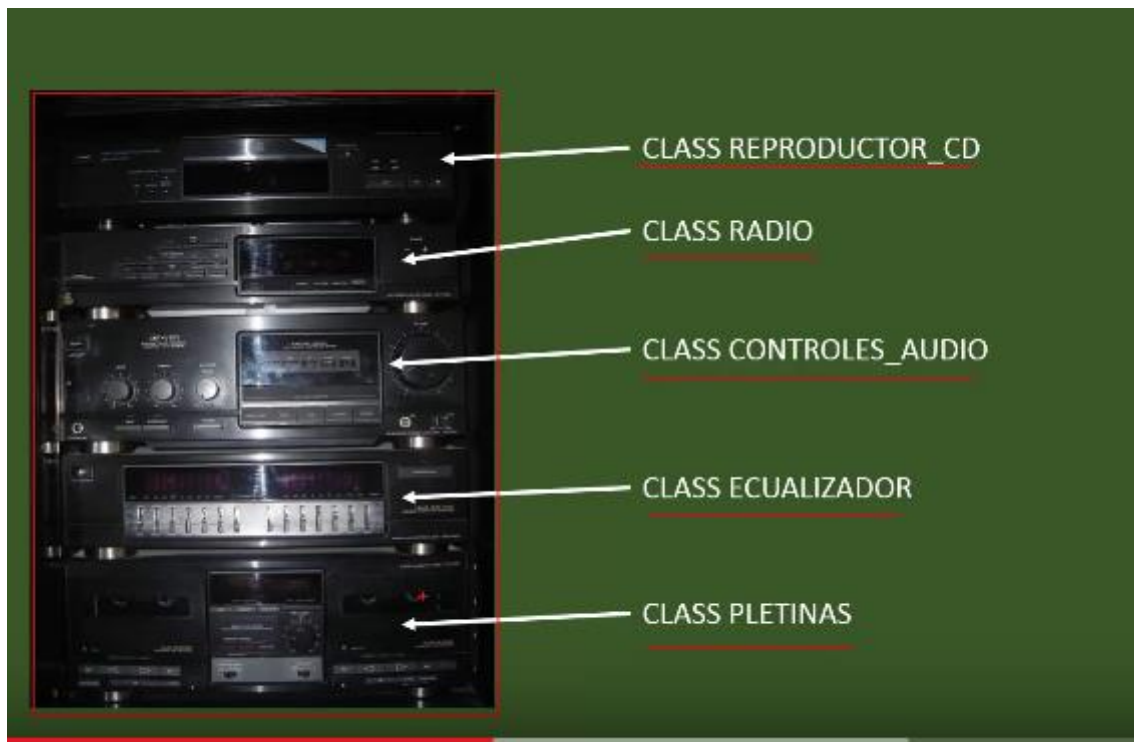
Cuando conseguimos que estas partes funcionen entre si, tenemos un programa en PHP

Una ventaja de la modularización es que al funcionar de forma independiente, si uno no funciona, el resto no el afecta : lo programas de forma independiente, y además la depuración de errores

## ENCAPSULACION

Partiendo de la modularización, consiste en que cada una de sus partes deben de tener zonas comunes, pero otras que no deben ser accesibles.

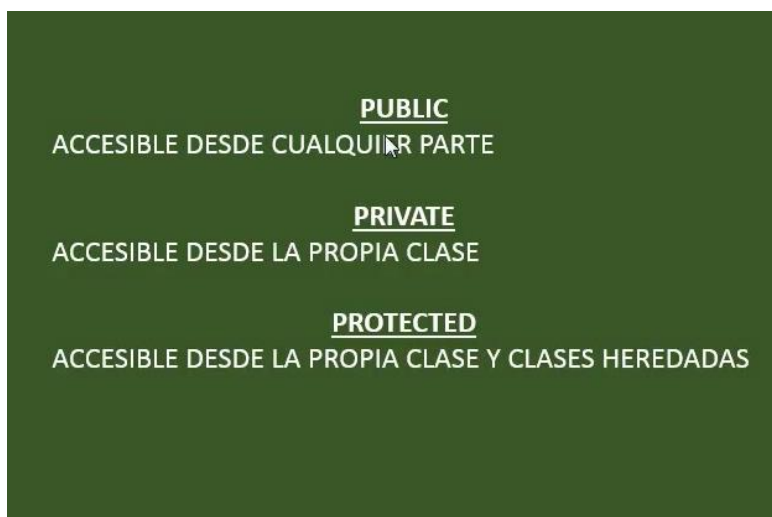
Ejemplo si tuviéramos que programar un equipo de música antiguo lo dividiríamos en las siguientes clases



No tendría sentido que yo avanzara o retrocediera un cd desde la clase pletina.

Todas las clases deben encapsular propiedades y funcionalidades

Para modificar algunos elementos si y otros no, utilizamos los **modificadores de acceso**. Y en PHP contamos con tres



ejemplo de public: permite modificar el volumen desde cualquier lado

private encapsula

protected para que sea accesible desde la misma clase y desde objetos que hereden desde el objeto en cuestión

Vamos ahora a verlo en un ejemplo.

Si seguimos con el ejercicio anterior, nos damos cuenta de que en cualquier momento yo puedo cambiar el valor del constructor, poniendo por ejemplo que un coche tenga 7 ruedas, lo cual es totalmente un despropósito

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

$mazda=new Coche();

$pegaso=new Camion();

$mazda->ruedas=7;

echo "El Mazda tiene " . $mazda->ruedas . " ruedas <br>";

echo "El Pegaso tiene " . $pegaso->ruedas . " ruedas <br>";
```

Para evitar esto encapsulamos la variable ruedas, cambiando el prefijo VAR por PRIVATE

```
class Coche{

    private $ruedas;

    var $color;

    var $motor;

    function Coche(){ //Método constructor;

        $this->ruedas=4;

        $this->color="";

        $this->motor=1600;
```

al encapsular la propiedad ruedas, ya dará un error si intentas modificar el valor del constructor

Ni siquiera el asistente (si utilizamos Dreamweaver, me da la opción de escoger ahora la propiedad ruedas



Pero ahora hay otro problema. En el caso de que solo queramos acceder a la información de propiedad ruedas del constructor, con el encapsulador PRIVATE, tampoco vamos a poder, por lo tanto tenemos que utilizar los métodos getters y setters que nos van a permitir acceder a las propiedades de un objeto.

### Setter para modificar las propiedades de un objeto

### Getter para ver las propiedades de un objeto

Hemos visto en el ejercicio anterior, como las expresiones

```
echo "El Mazda tiene " . $mazda->ruedas . " ruedas <br>";  
echo "El Pegaso tiene " . $pegaso->ruedas . " ruedas <br>";
```

ya no sirven de nada. Ahora utilizaremos el método getter para ver cual es el valor de la propiedad ruedas

¿cómo?

En cualquier parte del código, aunque siempre después del constructor, se introduce una función , que por convención (aunque no por obligación), suele empezar por get, ejemplo get\_ruedas

```
$this->color="";  
$this->motor=1600;  
  
}  
  
function get_ruedas() {  
    return $this->ruedas;  
}  
  
function arrancar() {
```

Podíamos perfectamente poner dame\_ruedas y seguiría siendo un método getter que me permite acceder al valor o estado de la propiedad del constructor, que yo quiera.

Ahora desde el documento html que está vinculado, a el php, hago un echo y no sale la propiedad ruedas pero si el getter, get\_ruedas()

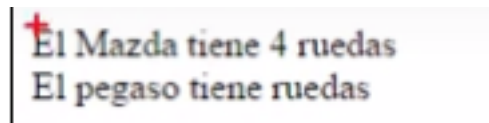
```
echo "El Mazda tiene " . $mazda->
```



la instrucción sería

```
echo "El Mazda tiene " . $mazda->get_ruedas() . " ruedas<br>";
```

si ahora esto mismo lo pruebo en pegaso, que pertenece al constructor camión, devolvería...



no reconoce la rueda, porque el objeto camión está heredando de Coche, y el getter solo funciona en el objeto en el que se encuentra. Para que pudiera heredar el getter, la propiedad ruedas debería estar precedida de PROTECTED, que le permitiría que sus propiedades sean accesibles también a los objetos heredados

En cuanto a los métodos setter, ya hemos hecho uno sin saberlo, puesto que hemos creado la función establece\_color que permite cambiar una propiedad del objeto por lo que es un setter.

Por convención se debería llamar **function set\_color**

Public es el modificador por defecto. VAR equivale a PUBLI