PHP - Objetos

1. Clases en PHP
2. Constructores
3. Destructores
4. Herencia
5. Modificadores de Acceso
6. Public
7. Private
8. Sobrecarga
9. Call()
10. Atributos como Objetos

Clases en PHP

Vemos que es una clase, y como podemos definirlas e instanciarlas.

Las clases en Programación orientada a objetos (POO) son definiciones de los elementos que forman un sistema, en este caso, definiciones de los objetos que van a intervenir en nuestros programas.

Un objeto se define indicando qué propiedades y funcionalidades tiene. Justamente esas declaraciones son lo que es una clase. Cuando se hace una clase simplemente se especifica qué propiedades y funcionalidades tiene. Por ejemplo, un persona podría tener como propiedades el nombre o la edad y como funcionalidades, comer, moverse o estudiar.

En la clase persona declararíamos dos atributos: la edad o el nombre, que serían como dos variables. También deberíamos crear tres métodos, con los procedimientos a seguir para que el persona pueda comer, moverse o estudiar. Estos métodos se definen declarando funciones dentro de la clase.

El código para definir una clase se puede ver a continuación:

```
class persona{
var $nombre;
var $edad;

function comer($comida){
//aquí el código del método
}

function moverse($destino){
//aquí el código del método
}

function estudiar($asignatura){
//aquí el código del método
}
}
```

Podrá comprobarse que este código no difiere en nada del de las versiones anteriores de PHP, que ya soportaban ciertas características de la POO. Esta situación cambiará a poco que exploremos las características más avanzadas de PHP , que implicarán mejoras que no estaban presentes en las versiones anteriores

Instanciar objetos a partir de clases

Hemos visto que una clase es tan sólo una definición. Si queremos trabajar con las clases debemos instanciar objetos, proceso que consiste en generar un ejemplar de una clase.

Por ejemplo, tenemos la clase persona anterior. Con la clase en si no podemos hacer nada, pero podemos crear objetos persona a partir de esa clase. Cada objeto persona tendrá unas características propias, como la edad o el nombre. Además podrá desempeñar unas funciones como comer o moverse aunque cada uno comerá o se moverá por su cuenta cuando le sea solicitado, sin interferir en principio con lo que pueda estar haciendo otra persona.

Vamos a ver cómo se generarían dos de personas, es decir, cómo se instanciarían un par de objetos de la clase persona. Para ello utilizamos el operador new.

```
$pepe = new persona();
$juan = new persona();
```

Es importante darse cuenta de la diferencia entre un objeto y una clase. La clase es una definición de unas características y funcionalidades, algo abstracto que se concreta con la instancia de un objeto de dicha clase.

Un objeto ya tiene propiedades, con sus valores concretos, y se le pueden pasar mensajes (llamar a los métodos) para que hagan cosas.

Constructores en PHP 5

Los constructores se encargan de resumir las acciones de inicialización de los objetos. Cuando se instancia un objeto, se tienen que realizar varios pasos en su inicialización, por ejemplo, dar valores a sus atributos y eso es de lo que se encarga el constructor. Los constructores pueden recibir unos datos para inicializar los objetos como se desee en cada caso.

La sintaxis para la creación de constructor varía con respecto a las versiones de PHP, ya que debe llamarse con un nombre fijo: __construct(). (Son dos guiones bajos antes de la palabra "construct")

Por ejemplo: definiendo una clase cliente, que utilizaremos luego en nuestro proyecto de programa.

```
class cliente{
	var $nombre;
	var $numero;
	var $maquinas_alquiladas;

	function __construct($nombre,$numero){
	$this->nombre=$nombre;
	$this->numero=$numero;
	$this->maquinas_alquiladas=array();
	}

	function dame_numero(){
	return $this->numero;
	}
}
```

El constructor en esta clase recibe el nombre y número a asignar al cliente, que introduce luego en sus correspondientes propiedades. Además inicializa el atributo maquinas_alquiladas como un array, en este caso vacío porque todavía no tiene ninguna maquina en su poder.

Nota: En programación orientada a objetos $this hace referencia al objeto sobre el que se está ejecutando el método. En este caso, como se trata de un constructor, $this hace referencia al objeto que se está construyendo. Con $this->nombre=$nombre; estamos asignando al atributo nombre del objeto que se está construyendo el valor que contiene la variable $nombre, que se ha recibido por parámetro.
Luego hemos creado un método muy sencillo para poder utilizar el objeto. Vamos a ver unas acciones simples para ilustrar el proceso de instanciar y utilizar los objetos.

```
//instanciamos un par de objetos cliente
$cliente1 = new cliente("Pepe", 1);
$cliente2 = new cliente("Roberto", 564);

//mostramos el numero de cada cliente creado
echo "El identificador del cliente 1 es: " . $cliente1->dame_numero();
echo "El identificador del cliente 2 es: " . $cliente2->dame_numero();
```

Este ejemplo obtendría esta salida como resultado de su ejecución:

```
El identificador del cliente 1 es: 1
El identificador del cliente 2 es: 564
```

Destructores en PHP

Los destructores son funciones que se encargan de realizar las tareas que se necesita ejecutar cuando un objeto deja de existir. Cuando un objeto ya no está referenciado por ninguna variable, deja de tener sentido que esté almacenado en la memoria, por tanto, el objeto se debe destruir para liberar su espacio. En el momento de su destrucción se llama a la función destructor, que puede realizar las tareas que el programador estime oportuno realizar.

La creación del destructor es opcional. Sólo debemos crearlo si deseamos hacer alguna cosa cuando un objeto se elimine de la memoria.

El destructor es como cualquier otro método de la clase, aunque debe declararse con un nombre fijo: __destruct().

En el código siguiente vamos a ver un destructor en funcionamiento. Aunque la acción que realiza al destruirse el objeto no es muy útil, nos puede servir bien para ver cómo trabaja.

```
class cliente{
	var $nombre;
	var $numero;
	var $maquinas_alquiladas;

	function __construct($nombre,$numero){
	$this->nombre=$nombre;
	$this->numero=$numero;
	$this->maquinas_alquiladas=array();
	}

	function __destruct(){
	echo "<br>destruido: " . $this->nombre;
	}

	function dame_numero(){
	return $this->numero;
	}
}

//instanciamos un par de objetos cliente
$cliente1 = new cliente("Pepe", 1);
$cliente2 = new cliente("Roberto", 564);

//mostramos el numero de cada cliente creado
echo "El identificador del cliente 1 es: " . $cliente1->dame_numero();
echo "<br>El identificador del cliente 2 es: " . $cliente2→dame_numero();
```

Este código es igual que el anterior. Sólo se ha añadido el destructor, que imprime un mensaje en pantalla con el nombre del cliente que se ha destruido. Tras su ejecución obtendríamos la siguiente salida.

```
El identificador del cliente 1 es: 1
El identificador del cliente 2 es: 564
destruido: Pepe
```

destruido: Roberto

Como vemos, antes de acabar el script, se libera el espacio en memoria de los objetos, con lo que se ejecuta el destructor y aparece el correspondiente mensaje en la página.

Un objeto puede quedar sin referencias y por lo tanto ser destruido, por muchas razones. Por ejemplo, el objeto puede ser una variable local de una función y al finalizarse la ejecución de esa función la variable local dejaría de tener validez, con lo que debe destruirse.

El código siguiente ilustra cómo una variable local a cualquier ámbito (por ejemplo, local a una función), se destruye cuando ese ámbito ha finalizado.

```
function crea_cliente_local(){
        $cliente_local = new cliente("soy local", 5);
}
crea_cliente_local()
```

La función simplemente crea una variable local que contiene la instancia de un cliente. Cuando la función se acaba, la variable local deja de existir y por lo tanto se llama al destructor definido para ese objeto.

Nota: También podemos deshacernos de un objeto sin necesidad que acabe el ámbito donde fue creado. Para ello tenemos la función unset() que recibe una variable y la elimina de la memoria. Cuando se pierde una variable que contiene un objeto y ese objeto deja de tener referencias, se elimina al objeto y se llama al destructor.

Herencia

La herencia es uno de los mecanismos fundamentales de la programación orientada a objetos. Por medio de la herencia, se pueden definir clases a partir de la declaración de otras clases. Las clases que heredan incluyen tanto los métodos como las propiedades de la clase a partir de la que están definidos.

Por ejemplo, pensemos en la clase "vehículo". Esta clase general puede incluir las características generales de todos los vehículos (atributos de la clase), como la matrícula, año de fabricación y potencia. Además, incluirá algunas funcionalidades (métodos de la clase) como podrían ser, arrancar() o moverse().

Ahora bien, en la práctica existen varios tipos de vehículos, como los coches, los micros y los camiones. Todos ellos tienen unas características comunes, que han sido definidas en la clase vehículo. Además, tendrán una serie de características propias del tipo de vehículo, que, en principio, no tienen otros tipos de vehículos. Por ejemplo, los camiones pueden tener una carga máxima permitida o los micros un número de plazas disponibles. Del mismo modo, las clases más específicas pueden tener unas funcionalidades propias, como los camiones cargar() y descargar(), o los micros aceptar_pasajeros() o vender_pasaje().

Lo normal en sistemas de herencia es que las clases que heredan de otras incluyan nuevas características y funcionalidades, aparte de los atributos y métodos heredados. Pero esto no es imprescindible, de modo que se pueden crear objetos que hereden de otros y no incluyan nada nuevo.

Sintaxis de herencia en PHP

La programación orientada a objetos nos ofrece una serie de mecanismos para definir este tipo de estructuras, de modo que se puedan crear jerarquías de objetos que heredan unos de otros. Veremos ahora cómo definir estas estructuras de herencia en PHP. Para ello, continuando con nuestro ejemplo, vamos a crear los distintos tipos de elementos que se ofrecen en alquiler.

Como todo el mundo conoce, los comercios de venta de juegos digitales ofrecen distintos tipos de soportes para vender, como pueden ser los cartuchos y los cd´s. Cada elemento tiene unas características propias y algunas comunes. Hemos llamado "soporte" a la clase general, que incluye las características comunes para todos los tipos de elementos en venta. Luego hemos creado tres tipos de soportes distintos, que heredan de la clase soporte, pero que incluyen algunas características y funcionalidades nuevas. Estos tipos de soporte serán "cd", "dvd" y "cartucho".

El esquema de herencia que vamos a realizar en este ejemplo se puede ver en la siguiente imagen.

Empezamos por la clase soporte. Su código será el siguiente:

```
class soporte{
   public $titulo;
   protected $numero;
   private $precio;

   function __construct($tit,$num,$precio){
      $this->titulo = $tit;
      $this->numero = $num;
```

```
    $this->precio = $precio;
  }

  public function dame_precio_sin_iva(){
    return $this->precio;
  }

  public function dame_precio_con_iva(){
    return $this->precio * 1.21;
  }

  public function dame_numero_identificacion(){
    return $this->numero;
  }

  public function imprime_caracteristicas(){
     echo $this->titulo;
     echo "<br>" . $this->precio . " (IVA no incluido)";
  }
}
```

Los atributos que hemos definido son, título, numero (un identificador del soporte) y precio. Hemos aplicado a cada uno un modificador de acceso distinto, para poder practicar los distintos tipos de acceso.

Hemos definido un constructor, que recibe los valores para la inicialización del objeto. Un método dame_precio_sin_iva(), que devuelve el precio del soporte, sin aplicarle el IVA. Otro método dame_precio_con_iva(), que devuelve el precio una vez aplicado el 21% de IVA. El método dame_numero_identificacion(), que devuelve el número de identificador y imprime_caracteristicas(), que muestra en la página las características de este soporte.

Nota: *Como se ha definido como private el atributo precio, este atributo sólo se podrá acceder dentro del código de la clase, es decir, en la propia definición del objeto. Si queremos acceder al precio desde fuera de la clase (algo muy normal si tenemos en cuenta que vamos a necesitar el precio de un soporte desde otras partes de la aplicación) será necesario crear un método que nos devuelva el valor del precio. Este método debería definirse como public, para que se pueda acceder desde cualquier sitio que se necesite.*

En todos los métodos se hace uso de la variable $this. Esta variable no es más que una referencia al objeto sobre el que se está ejecutando el método. En programación orientada a objetos, para ejecutar cualquiera de estos métodos, primero tenemos que haber creado un objeto a partir de una clase.

Luego podremos llamar los métodos de un objeto.

Esto se hace con $mi_objeto->metodo_a_llamar(). Dentro de método, cuando se utiliza la variable $this, se está haciendo referencia al objeto sobre el que se ha llamado al método, en este caso, el objeto $mi_objeto. Con $this->titulo estamos haciendo referencia al atributo "titulo" que tiene el objeto $mi_objeto.

Si queremos probar la clase soporte, para confirmar que se ejecuta correctamente y que ofrece resultados legibles, podemos utilizar un código como el siguiente.

```
$soporte1 = new soporte("Futbol 2020",22,3);
```

```
echo "<b>" . $soporte1->titulo . "</b>";
echo "<br>Precio: " . $soporte1->dame_precio_sin_iva() . " pesos";
echo "<br>Precio IVA incluido: " . $soporte1->dame_precio_con_iva() . " pesos";
```

En este caso hemos creado una instancia de la clase soporte, en un objeto que hemos llamado $soporte1. Luego imprimimos su atributo titulo (como el título ha sido definido como public, podemos acceder a él desde fuera del código de la clase.

Luego se llaman a los métodos dame_precio_sin_iva() y dame_precio_con_iva() para el objeto creado.

Nos daría como resultado esto:

Futbol 2020
Precio: 3 pesos
Precio IVA incluido: 3.63 pesos

Herencia II

Continuando con nuestro ejemplo de venta de juegos, vamos a construir una clase para los soportes de tipo cd´s. Los cd´s tienen un atributo nuevo que es su capacidad. No tienen ninguna clase nueva, aunque debemos aprender a sobrescribir métodos creados para el soporte, dado que ahora tienen que hacer tareas más específicas.

Sobrescribir métodos

Antes de mostrar el código de la clase cds, vamos a hablar sobre la sobrescritura o sustitución de métodos, que es un mecanismo por el cual una clase que hereda puede redefinir los métodos que está heredando.

Pensemos en una cafetera. Sabemos que existen muchos tipos de cafeteras y todas hacen café, pero el mecanismo para hacer el café es distinto dependiendo del tipo de cafetera. Existen cafeteras express, cafeteras por goteo y cafeteras con filtro de tela. Nuestra cafetera "padre" (de la que va a heredar todas las cafeteras) puede tener definido un método hacer_cafe(), pero no necesariamente todas las cafeteras que puedan heredar de esta hacen el café siguiendo el mismo proceso.

Entonces podemos definir un método para hacer café estándar, que tendría la clase cafetera. Pero al definir las clases cafetera_express y cafetera_goteo, deberíamos sobrescribir el método hacer_cafe() para que se ajuste al procedimiento propio de estas.

La sobrescritura de métodos es algo bastante común en mecanismos de herencia, puesto que los métodos que fueron creados para una clase "padre" no tienen por qué ser los mismos que los definidos en las clases que heredan.

Veremos cómo sobrescribir o sustituir métodos en un ejemplo de herencia, siguiendo nuestro ejemplo.

Sintaxis para heredar

Habíamos comentado que la venta de juegos dispone de distintos soportes. Habíamos creado una clase soporte, que vamos a heredar en cada uno de los elementos disponibles para vender. Vamos a empezar por la clase cds, cuyo código será el siguiente.

```
class cds extends soporte{
  private $capacidad;

  function __construct($tit,$num,$precio,$capacidad){
    parent::__construct($tit,$num,$precio);
    $this-> capacidad = $capacidad;
  }

  public function imprime_caracteristicas(){
    echo "Juego de cd:<br>";
    parent::imprime_caracteristicas();
    echo "<br>Capacidad: " . $this->capacidad;
  }
}
```

Con la primera línea class cds extends soporte estamos indicando que se está definiendo la clase cds y que va a heredar de la clase soporte.

Como se está heredando de una clase, PHP tiene que conocer el código de la clase "padre", en este caso la clase soporte. De modo que el código de la clase soporte debe estar incluido dentro del archivo de la clase cds. Podemos colocar los dos códigos en el mismo fichero, o si están en ficheros independientes, debemos incluir el código de la clase soporte con la instrucción include o require de PHP.
En la clase cds hemos definido un nuevo atributo llamado $capacidad, que almacena tamaño del juego en el cd.

Aunque la clase sobre la que heredamos (la clase soporte) tenía definido un constructor, el cd debe inicializar la nueva propiedad $capacidad, que es específica de los cds. Por ello, vamos a sobrescribir o sustituir el método constructor, lo que se hace simplemente volviendo a escribir el método.

Aunque se había definido un constructor para la clase soporte, que inicializaba los atributos de esta clase. Ahora, para la clase cds, hay que inicializar los atributos definidos en la clase soporte, más el atributo $capacidad, que es propio de cds.

El código del constructor es el siguiente:

```
function __construct($tit,$num,$precio,$capacidad){
   parent::__construct($tit,$num,$precio);
   $this-> capacidad = $capacidad;
}
```

En la primera línea se llama al constructor creado para la clase "soporte". Para ello utilizamos parent:: y luego el nombre del método de la clase padre al que se quiere llamar, en este caso __construct(). Al constructor de la clase padre le enviamos las variables que se deben inicializar con esa clase.

En la segunda línea del constructor se inicializa el atributo capacidad, con lo que hayamos recibido por parámetro.

Nos pasa lo mismo con el método imprime_caracteristicas(), que ahora debe mostrar también el nuevo atributo, propio de la clase cds. Como se puede observar en el código de la función, se hace uso también de parent::imprime_caracteristicas() para utilizar el método definido en la clase padre.

Si queremos probar la clase cds, podriamos utilizar un código como este:

```
$mijuego = new cds("Mario 2020", 22, 4.5, "500mb");
echo "<b>" . $mijuego->titulo . "</b>";
echo "<br>Precio: " . $mijuego->dame_precio_sin_iva() . " pesos";
echo "<br>Capacidad: " . $mijuego->imprime_caracteristicas() ;
```

Lo que nos devolvería lo siguiente:

```
Juego de cd:Mario 2020
4.5 (IVA no incluido)
Capacidad: 500mb
```

Modificadores de acceso a métodos y propiedades

Uno de los principios de la programación orientada a objetos es la encapsulación, que es un proceso por el que se ocultan las características internas de un objeto a aquellos elementos que no tienen porque conocerla. Los modificadores de acceso sirven para indicar los permisos que tendrán otros objetos para acceder a sus métodos y propiedades.

Modificador public
Es el nivel de acceso más permisivo. Sirve para indicar que el método o atributo de la clase es público. En este caso se puede acceder a ese atributo, para visualizarlo o editarlo, por cualquier otro elemento de nuestro programa. Es el modificador que se aplica si no se indica otra cosa.

Veamos un ejemplo de clase donde hemos declarado como public sus elementos, un método y una propiedad. Se trata de la clase "dado", que tiene un atributo con su puntuación y un método para tirar el dado y obtener una nueva puntuación aleatoria.

```
class dado{
	public $puntos;

	function __construct(){
	srand((double)microtime()*1000000);
	}

	public function tirar(){
	$this->puntos=$randval = rand(1,6);
	}
}

$mi_dado = new dado();

for ($i=0;$i<30;$i++){
	$mi_dado->tirar();
	echo "<br>Han salido " . $mi_dado->puntos . " puntos";
}
```

Vemos la declaración de la clase dado y luego unas líneas de código para ilustrar su funcionamiento. En el ejemplo se realiza un bucle 30 veces, en las cuales se tira el dado y se muestra la puntuación que se ha obtenido.

Como el atributo $puntos y el método tirar() son públicos, se puede acceder a ellos desde fuera del objeto, o lo que es lo mismo, desde fuera del código de la clase.

Modificador private
Es el nivel de acceso más restrictivo. Sirve para indicar que esa variable sólo se va a poder acceder desde el propio objeto, nunca desde fuera. Si intentamos acceder a un método o atributo declarado private desde fuera del propio objeto, obtendremos un mensaje de error indicando que no es posible acceder a ese elemento.

Si en el ejemplo anterior hubiéramos declarado private el método y la propiedad de la clase dado, hubiéramos recibido un mensaje de error.

Aquí tenemos otra posible implementación de la clase dado, declarando como private el atributo puntos y el método tirar().

```
class dado{
	private $puntos;

	function __construct(){
	srand((double)microtime()*1000000);
	}

	private function tirar(){
	$this->puntos=$randval = rand(1,6);
	}

	public function dame_nueva_puntuacion(){
	$this->tirar();
	return $this->puntos;
	}
}

$mi_dado = new dado();

for ($i=0;$i<30;$i++){
	echo "<br>Han salido " . $mi_dado->dame_nueva_puntuacion() . " puntos";
}
```

Hemos tenido que crear un nuevo método público para operar con el dado, porque si es todo privado no hay manera de hacer uso de él. El mencionado método es dame_nueva_puntuación(), que realiza la acción de tirar el dado y devolver el valor que ha salido.

Modificador protected

Este indica un nivel de acceso medio y un poco más especial que los anteriores. Sirve para que el método o atributo sea público dentro del código de la propia clase y de cualquier clase que herede de aquella donde está el método o propiedad protected. Es privado y no accesible desde cualquier otra parte. Es decir, un elemento protected es público dentro de la propia clase y en sus heredadas.

Sobrecarga de constructores en PHP

Al notar la falta de sobrecarga de métodos en la Programación Orientada a Objetos implementada en PHP, uno se puede suponer qué no existe en PHP? ¿no se trata de una POO devaluada debido a la carencia de sobrecarga? ¿cómo puedo hacer clases que tengan constructores que acepten varios juegos de parámetros?

Podemos observar que en PHP sufrimos la carencia de sobrecarga debido a que es un lenguaje poco estricto. En el caso de las funciones, PHP tiene la particularidad de permitir invocarlas enviando un juego de parámetros diferente al que fue declarada. Sin invocamos una función sin enviarle todos los parámetros, PHP dará mensajes de advertencia, pero el código se ejecuta. Si al invocar una función enviamos más parámetros de los que tocaba, ni siquiera nos da un mensaje de advertencia.

Es por ello que no pueden coexistir funciones con el mismo nombre y distintos juegos de parámetros, porque PHP siempre va a utilizar la función que se declare (una única vez), enviando los parámetros de que disponga en el momento se su invocación.

¿Cómo simular la sobrecarga de métodos constructores?
La opción para aprovechar las posibilidades de la sobrecarga de métodos es programarla manualmente. Se trata de un proceso sencillo de incorporar, aunque no cabe duda de que es más fácil de comprender cuando está admitido por el lenguaje de manera nativa.

El desarrollo se basa en un método que no recibe parámetros y un método específico para cada número de parámetros que pensemos aceptar. Este método "genérico" lo declaramos sin indicar ningún parámetro y dentro de su código utilizamos la función de PHP func_get_args(), que nos permite extraer todos los parámetros que pueda estar recibiendo la función. Una vez que obtuvimos el número de parámetros que nos han enviado en tiempo de ejecución, podemos invocar a la función específica que tiene el código a ejecutar cuando se recibe ese número concreto de parámetros.

Un código del siguiente ejemplo y sus comentarios ayudan a su comprensión:

```
class jugador
{
	private $nombre;
	private $equipo;
	function __construct()
	{
		//obtengo un array con los parámetros enviados a la función
		$params = func_get_args();
		//saco el número de parámetros que estoy recibiendo
		$num_params = func_num_args();
		//cada constructor de un número dado de parámetros tendrá un nombre de
		función
		//atendiendo al siguiente modelo __construct1() __construct2()...
		$funcion_constructor ='__construct'.$num_params;
		//compruebo si hay un constructor con ese número de parámetros
		if (method_exists($this,$funcion_constructor)) {
			//si existía esa función, la invoco, reenviando los parámetros que recibí en el
constructor original
			call_user_func_array(array($this,$funcion_constructor),$params);
		}
```

```
	}
	//ahora declaro una serie de métodos constructores que aceptan diversos números de
	parámetros
	function __construct0()
	{
		$this>__construct1("Anónimo");
	}
	function __construct1($nombre)
	{
		$this>__construct2($nombre, "Sin equipo");
	}
		function __construct2($nombre, $equipo)
	{
		$this>nombre = $nombre;
		$this>equipo = $equipo;
	}
}
```

Seguramente si conoces otros lenguajes de programación como JAVA, este apartado te resulte extraño, ya que PHP sobrecarga los métodos de forma muy diferente a como se hace de forma común a otros lenguajes.

Otra opción.
Para ello se utiliza un método llamado __call() al que se le pasa como argumentos; el nombre del método y una lista con los argumentos que este método necesitaría. Este método __call, recibe aquellos argumentos que no están definidos de forma independiente, comprobando que estén definidos en su interior.

```
function __call($metodo, $argumentos)
{
 ...
}
```

De este modo imaginemos que queremos introducir una sobrecarga en nuestro método set_name, donde podría introducirse el nombre y los apellidos. Deberemos de crear nuestro método __call en relación a este objetivo.
Lo primero que habría que discernir es si el método pasado por argumento es el correcto, haciéndolo con un if:

```
function __call($metodo, $argumentos)
{
 if ($metodo = 'set_nombre'){
  ...
 }
}
```

De este modo controlamos que el método tenga el nombre correcto, pero ahora hay que comprobar el número de argumentos y que se debe de hacer en cada caso, ya que como se ha dicho $argumentos es una lista o array.

```
function __call($metodo, $argumentos)
{
```

```
 if ($metodo = 'set_nombre'){
  if (count($argumentos) == 1){
   $this->nombre = $argumentos[0];
  }
  if (count($argumentos) == 2){
   $this->nombre = $argumentos[0];
   $this->apellido = $argumentos[1];
  }
 }
}
```

El código comprueba el número de elementos enviados en la lista y dependiendo de cuantos haya crea el objeto con o sin apellido. (Recordemos que los arrays tienen como primer elemento el cero.)

De este modo ahora podemos comprobar como funciona con el siguiente código, donde se crean dos objetos, cada uno con un número diferente de argumentos:

```
<?php
 class Person
 {
  var $nombre;
  var $apellido;
  var $edad;

  //Funcion __call que recoge el nombre sobrecargado
  function __call($metodo, $argumentos)
  {
   if ($metodo = 'set_nombre'){
    if (count($argumentos) == 1){
     $this->nombre = $argumentos[0];
    }
    if (count($argumentos) == 2){
     $this->nombre = $argumentos[0];
     $this->apellido = $argumentos[1];
    }
   }
  }

  //Aquí comienzan los geters y seters de las propiedades.
  function get_nombre()
  {
   return $this->nombre;
  }
  function get_apellido()
  {
   return $this->apellido;
  }

  function set_edad($dato)
  {
   $this->edad = $dato;
  }
```

```
 function get_edad()
 {
  return $this->edad;
 }
}
$persona1 = new Person;
$persona1->set_nombre('Antonio');
$persona1->set_edad('20');

$persona2 = new Person;
$persona2->set_nombre('Manuel', 'Gonzalez');
$persona2->set_edad('28');

 echo '<h3>Persona1</h3>';
 echo '<p>Mostrando con $persona1->get_nombre: ' . $persona1->get_nombre() . '</p>';
 echo '<p>Mostrando con $persona1->get_edad: ' . $persona1->get_edad() . '</p>';
 echo '<h3>Persona2</h3>';
 echo '<p>Mostrando con $persona2->get_nombre: ' . $persona2->get_nombre() . '</p>';
 echo '<p>Mostrando con $persona2->get_apellido: ' . $persona2->get_apellido() . '</p>';
 echo '<p>Mostrando con $persona2->get_edad: ' . $persona2->get_edad() . '</p>';
?>
```

En este código podemos ver como esta incluido nuestro método sobrecargado, y como creamos los dos objetos, el que solo lleva un argumento y el que lleva los dos.
Después solicitamos los datos de ambos objetos y obtenemos el siguiente resultado:

Persona1
Mostrando con $persona1->get_nombre: Antonio
Mostrando con $persona1->get_edad: 20

Persona2
Mostrando con $persona2->get_nombre: Manuel
Mostrando con $persona2->get_apellido: Gonzalez
Mostrando con $persona2->get_edad: 28

Los atributos de los objetos pueden ser otros objetos

Las características de los objetos, que se almacenan por medio de los llamados atributos o propiedades, pueden ser de diversa naturaleza. La clase persona puede tener distintos tipos de atributos, como la edad (numérico), el nombre propio (tipo cadena de caracteres), color de pelo (que puede ser un tipo cadena de caracteres o tipo enumerado, que es una especie de variable que sólo puede tomar unos pocos valores posibles). También puede tener una estatura o un peso (que podrían ser de tipo float o número en coma flotante).

En general, podemos utilizar cualquier tipo para los atributos de los objetos, incluso podemos utilizar otros objetos. Por ejemplo, podríamos definir como atributo de la clase persona sus manos. Dada la complejidad de las manos, estas podrían definirse como otro objeto. Por ejemplo, las manos tendrían como características la longitud de los dedos, un coeficiente de elasticidad. Como funcionalidades o métodos, podríamos definir agarrar algo, soltarlo, pegar una bofetada, o cortarse las uñas. Así pues, uno de los atributos de la clase persona podría ser un nuevo objeto, con su propias características y funcionalidades. La complejidad de las manos no le importa al desarrollador de la clase persona, por el principio de encapsulación, dado que este conoce sus propiedades (o aquellas declaradas como public) y los métodos (también los que se hayan decidido declarar como públicos) y no necesita preocuparse sobre cómo se han codificado.

Nota: *Como vemos, los objetos pueden tener algunos atributos también de tipo objeto. En ese caso pueden haber distintos tipos de relaciones entre objetos. Por ejemplo, por pertenencia, como en el caso de la clase persona y sus manos, pues a un persona le pertenecen sus manos. También se pueden relacionar los objetos por asociación, como es el caso de los clientes y los juegos que alquilan, pues en ese caso un cliente no tiene un juego determinado, sino que se asocia con un juego momentáneamente.*

Atributos de la clase cliente
Hemos definido una serie de atributos para trabajar con los clientes que alquilan juegos de computadoras o consolas. Tenemos los siguientes:

```
public $nombre;
private $numero;
private $soportes_alquilados;
private $num_soportes_alquilados;
private $max_alquiler_concurrente;
```

Como se puede ver, se han definido casi todos los atributos como private, con lo que sólo se podrán acceder dentro del código de la clase.

El atributo nombre, que guarda el nombre propio del cliente, es el único que hemos dejado como público y que por tanto se podrá referenciar desde cualquier parte del programa, incluso desde otras clases. El atributo numero se utiliza para guardar el identificador numérico del cliente. Por su parte, soportes alquilados nos servirá para asociar al cliente los juegos cuando este los alquile. El atributo num_soportes_alquilados almacenará el número de juegos que un cliente tiene alquilados en todo momento. Por último, max_alquiler_concurrente indica el número máximo de soportes que puede tener alquilados un cliente en un mismo instante, no permitiéndose alquilar a ese cliente, a la vez, más que ese número de juegos.

El atributo a analizar es soportes_alquilados, que contendrá un array de soportes. En cada casilla del array se introducirán juegos que un cliente vaya alquilando, para asociar esos soportes al cliente que los alquiló. El array contendrá tantas casillas como el max_alquiler_concurrente, puesto que no

tiene sentido asignar mayor espacio al array del que se va a utilizar. Para facilitar las cosas, cuando un cliente no tiene alquilado nada, tendrá el valor null en las casillas del array.

Constructor de la clase cliente

```
function __construct($nombre,$numero,$max_alquiler_concurrente=3){
  $this->nombre=$nombre;
  $this->numero=$numero;
  $this->soportes_alquilados=array();
  $this->num_soportes_alquilados=0;
  $this->max_alquiler_concurrente = $max_alquiler_concurrente;
  //inicializo las casillas del array de alquiler a "null"
  //un valor "null" quiere decir que el no hay alquiler en esa casilla
  for ($i=0;$i<$max_alquiler_concurrente;$i++){
    $this->soportes_alquilados[$i]=null;
  }
}
```

El constructor de la clase cliente recibe los datos para inicializar el objeto. Estos son $nombre, $numero y $max_alquiler_concurrente. Si nos fijamos, se ha definido por defecto a 3 el número máximo de alquileres que puede tener un cliente, de modo que, en el caso de que la llamada al constructor no envíe el parámetro $max_alquiler_concurrente se asumirá el valor 3.

El atributo soportes_alquilados, como habíamos adelantado, será un array que tendrá tantas casillas como el máximo de alquileres concurrentes. En las últimas líneas se inicializan a null el contenido de las casillas del array.