



PHP & MySQL

Anexo 6.- Orientación a Objetos



DISTRIBUIDO POR:

CENTRO DE INFORMÁTICA PROFESIONAL S.L.

C/ URGELL, 100
08011 BARCELONA
TFNO: 93 426 50 87

C/ RAFAELA YBARRA, 10
48014 BILBAO
TFNO: 94 448 31 33

www.cipsa.net

RESERVADOS TODOS LOS DERECHOS. QUEDA PROHIBIDO TODO TIPO DE REPRODUCCIÓN TOTAL O PARCIAL DE ESTE MANUAL, SIN PREVIO CONSENTIMIENTO POR EL ESCRITOR DEL EDITOR

POO Avanzada

Miembros estáticos

Los métodos y atributos convencionales pertenecen a los objetos. Todos los poseen los mismos atributos y métodos declarados en su la clase, pero cada uno almacena sus propios valores y sus métodos actúan exclusivamente sobre éstos.

Por el contrario, los **atributos estáticos** son atributos cuyo valor es compartido por todos los objetos. Si un objeto modifica su valor, éste cambia para todos los demás también. Para declarar un atributo estático se añade la palabra clave **static** entre el modificador de visibilidad y el nombre del atributo:

Ejemplo: La siguiente clase *empleado* consta de un atributo estático *\$num_empleados* que almacena el número de objetos existentes de la clase. El atributo estático se incrementa con el constructor y decrementa con el destructor:

```
class empleado{
    // Atributo estático
    public static $num_empleados = 0;

    public function __construct() {
        self::$num_empleados++; // Incrementa contador de objetos
    }
    public function __destruct() {
        self::$num_empleados--; // Decrementa contador de objetos
    }
}
```

Para acceder a un atributo estático desde un método definido dentro de la propia clase deben emplearse la palabra clave **self** seguida del operador (**::**). Para acceder a un atributo estático desde un método fuera de la propia clase (siempre y cuando sea público), puede emplearse el propio nombre de la clase seguida del operador (**::**).

```
$emp01 = new empleado();
$emp02 = new empleado();

echo empleado::$num_empleados; // Muestra el valor 2
```

Los **métodos estáticos** son métodos que tampoco pertenecen a los objetos, por lo que no pueden emplear la referencia predefinida **\$this**, ni acceder a los atributos no estáticos de la clase. Estos métodos suelen emplearse para obtener o modificar de manera controlada los atributos estáticos de la clase:

```
class empleado{
    // Atributo estático
    private static $num_empleados = 0;

    // Método estático
    public static function obtenerConexiones() {
        return self::$num_empleados;
    }
}
```

Programación Web en PHP

Los métodos estáticos deben invocarse empleando el nombre de la clase seguido del operador (::):

```
$emp01 = new empleado();
$emp02 = new empleado();

echo empleado::obtenerConexiones(); // Llamada al método estático. Muestra 2
```

Los métodos estáticos también suelen emplearse para implementar operaciones entre objetos tales como sumar o comparar objetos de la clase.

Ejemplo: Supóngase una clase tiempo que permite almacenar un lapso de tiempo expresado en horas, minutos y segundos.

```
class tiempo {
    private $h;
    private $m;
    private $s;
    public function __construct($ h, $ m, $ s ) {
        $this->fijar($ h, $ m, $ s);
    }
    public function fijar( $ h, $ m, $ s ) {
        $this->h = $ h;
        if ( $ m >= 0 && $ m <= 59 ) $this->m = $ m;
        if ( $ s >= 0 && $ s <= 59 ) $this->s = $ s;
    }
    public function mostrar() {
        echo "$this->h:$this->m:$this->s";
    }
}
```

Supóngase que la clase dispone de los dos métodos siguientes: Un método convencional **sumar()** que recibe como argumento otro objeto *tiempo* (*\$tiempo*) y añade sus horas, minutos y segundos a las horas, minutos y segundos del objeto.

```
public function sumar( $tiempo ) {
    // Añade horas
    $this->h += $tiempo->h;
    // Añade minutos
    $this->m += $tiempo->m;
    // Control desbordamiento minutos
    if ( $this->m >= 60 ) {
        $this->h++;
        $this->m-=60;
    }
    // Añade segundos
    $this->s += $tiempo->s;
    // Control desbordamiento segundos
    if ( $this->s >= 60 ) {
        $this->m++;
        $this->s-=60;
    }
}
```

El método convencional se invoca a partir del objeto \$t1, para que le sume las horas, minutos y segundos del objeto \$t2:

```
$t1 = new tiempo(10,20,40);
$t2 = new tiempo(0,50,50);
$t1->sumar($t2); // Suma al objeto $t1, el tiempo del objeto $t2.
$t1->mostrar(); // Muestra 11:11:30
```

Supóngase ahora que implementamos un método estático **sumarTiempos()** que recibe de objetos tiempo **\$t1** y **\$t2** y retorna como resultado un objeto tiempo con las horas, minutos y segundos totales de ambos objetos:

```
public static function sumarTiempos( $t1, $t2 ) {  
    // Calcula horas totales  
    $h = $t1->h + $t2->h;  
    // Calcula minutos totales  
    $m = $t1->m + $t2->m;  
    if ( $m >= 60 ) {  
        $h++;  
        $m-=60;  
    }  
    // Calcula segundos totales  
    $s = $t1->s + $t2->s;  
    if ( $s >= 60 ) {  
        $m++;  
        $s-=60;  
    }  
    // Crea objeto tiempo resultante y lo retorna  
    return new tiempo($h, $m, $s);  
}
```

El método estático se invoca a partir de la propia clase y retorna un objeto tiempo como resultado:

```
$t1 = new tiempo(10,20,40);  
$t2 = new tiempo(0,50,50);  
$r = tiempo::sumarTiempos( $t1, $t2); // Devuelve un objeto tiempo con la suma de $t1 y $t2  
$r->mostrar();
```

Constantes de Clase

La programación orientada a objetos permite la definición de constantes en las clases. Estas reciben el nombre de constantes de clase y se declaran dentro de la clase empleando la palabra clave **const** sin modificador de visibilidad ni signo \$ como prefijo:

```
class calculos {  
    const PI = 3.1415927;  
}
```

Las constantes pueden emplearse dentro de los métodos de la propia clase empleando la palabra clave **self** seguida del operador (::):

```
class calculos {  
    const PI = 3.1415927;  
    // Función que calcula el area de un círculo dado su radio  
    public static function areaCirculo( $radio ) {  
        return self::PI * pow($radio, 2);  
    }  
}
```

Para obtener el valor de una constante desde un método fuera de la propia clase debe indicarse el nombre de la clase seguido del operador (::):

```
echo "El valor de PI es: ". calculos::PI;
```

Especificación/Identificación de tipos

PHP es un lenguaje *no tipado* ya que una variable puede ser de cualquier tipo en función del valor que almacena en cada momento (cadena, lógico, número, matriz...,etc).

En el caso de los objetos, una variable puede contener un objeto de cualquier clase. PHP dispone del operador condicional **instanceof** que permite identificar la clase/interfaz a la que pertenece un objeto. Su sintaxis es:

<obj> instanceof <clase/interfaz>

El operador **instanceof** devuelve un valor lógico cierto si:

- \$obj → Es un objeto de la clase indicada.
- \$obj → Es un objeto de una clase hija de la clase indicada.
- \$obj → Es un objeto de una clase que implementa el interfaz indicado.

Ejemplo: El siguiente código comprueba si el objeto referenciado por la variable *\$obj* pertenece a la clase *tiempo*, o a una clase hija de *tiempo*.

```
if ( $obj instanceof tiempo ) {  
    echo "Es un objeto de la clase tiempo";  
}
```

Especificación de tipos


PHP también permite especificar la clase o interfaz a la que debe pertenecer los parámetros de un método o función indicando ésta delante del parámetro.

Ejemplo: El siguiente código declara una función *foo()* que requiere como parámetro de entrada un objeto de la clase *tiempo*:

```
function foo( tiempo $obj ) {  
    $obj->mostrar();  
}
```

Al llamar a esta función PHP emplea el operador **instanceof** para comprobar si el objeto dado como argumento pertenece a la clase/interfaz especificado en el método, en caso contrario se produce un error de tipo (*Uncaught TypeError*):

```
$objT = new tiempo(10,10,10);  
foo($objT); // Correcto: ( $objT es de tipo tiempo )  
$objE = new empleado("Rogén", "Petroviano");  
foo($objE); // Error: ( $objE no es de tipo tiempo )
```

 Fatal error: Uncaught TypeError: Argument 1 passed to foo() must be an instance of tiempo, instance of empleado given, called in C:\xampp7\htdocs\prueba\home.php on line 99 and defined in C:\xampp7\htdocs\prueba\home.php on line 92

Ejemplo 2: Supóngase ahora que las clases *tiempo* y *empleado* implementan el interfaz *visualizable* tal y como se muestra:

```
interface visualizable {
    function mostrar();
}

class empleado
    implements visualizable {
    private $nombre;
    private $apellido;
    public function __construct($ nombre, $ apellido) {
        $this->nombre = $ nombre;
        $this->apellido = $ apellido;
    }
    public function mostrar() {
        echo "Soy $this->nombre $this->apellido";
    }
}

class tiempo
    implements visualizable {
    private $h;
    private $m;
    private $s;
    public function __construct($ h, $ m, $ s ) {
        $this->h = $ h;
        if ( $ m >= 0 && $ m <= 59 ) $this->m = $ m;
        if ( $ s >= 0 && $ s <= 59 ) $this->s = $ s;
    }

    public function mostrar() {
        echo "$this->h:$this->m:$this->s";
    }
}
```

Si implementamos la función **foo()** anterior indicando como tipo del parámetro \$obj el interfaz *visualizable*:

```
function foo( visualizable $obj ) {
    $obj->mostrar();
}
```

Ahora las dos llamadas a la función *foo()* funcionan correctamente en ambos casos, ya que *\$objT* y *\$objE* pertenecen a clases que implementan el interfaz *visualizable*.

```
$objT = new tiempo(10,10,10);           // OK
foo($objT);
$objE = new empleado("Roger", "Petroviano"); // OK
foo($objE);
```

(*) La ventana de especificar parámetros de tipo interfaz o clases bases, es que el método permitirá recibir tanto objetos de las clases hijas, como de cualquier clase que implemente el interfaz. No obstante, en el interior de la función sólo podrán invocarse a los métodos y atributos definidos en la clase base e interfaz.

Esto significa que dentro de la función *foo()* sólo podemos invocar al método *mostrar()* sobre el objeto *\$obj*, aun cuando el objeto tenga más atributos y métodos.

Clases abstractas

Una clase abstracta es un tipo especial de clase de la que no pueden crearse objetos, y que sirve únicamente como modelo para sus subclases. Para definir una clase abstracta se indica la palabra clave **abstract** delante de la palabra clave **class** en su definición:

```
abstract class figura {
    protected $x;
    protected $y;

    function __construct($ x, $ y) {
        $this->x = $ x;
        $this->y = $ y;
    }

    // Método abstracto a sobrescribir por las subclases
    abstract function getArea();
}
```

Métodos abstractos

Un método abstracto es un método sin código que se define en una clase base para que esté presente y sea sobrescrito en todas sus clases hijas.

Ejemplo: La clase *figura* se declara abstracta ya que su método *getArea()* es abstracto al no poderse implementar para una figura cualquiera:

```
abstract class Figura {
    protected $x;
    protected $y;
    function __construct($ x, $ y) {
        $this->x = $ x;
        $this->y = $ y;
    }
    abstract function getArea();
}
```

Las subclases de las clases abstractas como *figura* heredan los métodos abstractos sobrescribiéndolos en cada caso con el código necesario:

```
class circulo extends figura {
    private $radio;
    function __construct( $ x, $ y, $ radio) {
        parent::__construct($ x, $ y);
        $this->radio = $ radio;
    }
    function getArea() {
        return "Area circulo: " . (3.1415927 * $this->radio * $this->radio);
    }
}

class cuadrado extends figura {
    private $lado;
    function __construct( $ x, $ y, $ lado) {
        parent::__construct($ x, $ y);
        $this->lado = $ lado;
    }
    function getArea() {
        return "Area cuadrado: " . ($this->lado * $this->lado);
    }
}
```


Diferencias en el uso entre *Interfaces* y clases *Abstractas*

- Una clase abstracta define un tipo no instanciable (no pueden crearse objetos) que sirve de base para otras clases hijas. Un interfaz representa una funcionalidad aplicable a cualquier clase que lo implemente.
- Una clase sólo puede heredar de una clase padre pero puede implementar múltiples interfaces al mismo tiempo.
- La herencia se aplica entre clases que representan subtipos de datos (figura -> circulo, cuadrado, triángulo). Las interfaces pueden ser implementadas por cualquier clase en la que se necesite incluir su funcionalidad.

Herencia e implementación pueden combinarse de modo que una clase puede heredar de una clase base e implementar uno o varios interfaces sin problemas.

Ejemplo: El siguiente código muestra la clase *cuadrado* que hereda de la clase abstracta *figura* sobrescribiendo su método *calculaArea()* e implementa al mismo tiempo el método *getHTML()* del interfaz *visualizable*.

```
// Clase abstracta Figura
abstract class Figura {
    protected $x;
    protected $y;

    function __construct($x, $y) {
        $this->x = $x;
        $this->y = $y;
    }
    // Método abstracto a sobrescribir por las clases hijas
    public abstract function calcularArea();
}

// Interfaz visualizable que devuelve el código HTML.
interface visualizable {
    function getHTML();
}

// Clase cuadrado hija de figura que implemente visualizable
class cuadrado extends Figura implements visualizable {
    private $lado;

    function __construct( $x, $y, $lado ) {
        parent::__construct($x,$y);
        $this->lado = $lado;
    }

    // Sobrescritura del método abstracto de la clase figura
    public function calcularArea() {
        return $this->lado * $this->lado;
    }

    // Implementación del método del interfaz visualizable
    public function getHTML() {
        $area = $this->calcularArea();
        $resultado = "<table><thead><td>X</td><td>Y</td><td>AREA</td></thead>";
        $resultado = $resultado . "<tr><td>$this->x</td><td>$this->y</td><td>$area</td></tr></table>";
        return $resultado;
    }
}
```

Clonado

Los valores de tipo numérico, decimal, cadenas de texto y lógico son tipos por valor. Esto significa que cada variable posee su propio valor y este no puede ser compartido ni modificado desde otra variable.

En el siguiente ejemplo de código se declaran dos variable \$a y \$b con valores de tipo numérico. Cada variable contiene su valor, de modo que cuando se hace una asignación (\$b = \$a) **el valor de \$a se copia en \$b**. **Como cada variable tiene su propio valor, la modificación del valor de \$a no afecta en ningún caso al valor de \$b**.

```
$a = 10;           // Asignacion valor $a
$b = $a;          // Valor de $a se copia en $b
$a = 20;          // Modificacion valor $a
echo "$a<br />";   // Muestra 20
echo "$b<br />";   // Muestra 10
```

Los objetos sin embargo son tipos por referencia. Esto significa que cada variable posee una referencia a un objeto en vez del propio objeto.

En el siguiente ejemplo de código se crea un objeto tiempo referenciado por \$objT1. Al realizar la asignación (\$objT2 = \$objT1) **ambas variables pasan a hacer referencia al mismo objeto**. De este modo **cualquier modificación realizada en objeto a través de una es visible desde la otra**.

```
// Instanciacion del objeto $objT1
$objT1 = new tiempo( 10,10,10);
// Asignacion del objeto $objT1 a $objT2
$objT2 = $objT1;
// Modificacion del atributo $m del objeto referenciado por $objT2.
$objT2->m = 30;

$objT1->mostrar();           // Muestra '10:30:10' -> apunta al mismo objeto
$objT2->mostrar();           // Muestra '10:30:10' -> apunta al mismo objeto
```

Para crear una copia (no una referencia) de un objeto debe emplearse el operador **clone** mediante la siguiente sintaxis:

\$objClonado = clone \$objOriginal

Tras la llamada a **clone**, la variable \$objClonado obtiene la referencia a un objeto duplica del objeto referenciado por \$objOriginal. Ambos objetos son distintos, de modo que la modificación de uno no afectará al otro.

Esta operación se denomina **clonado de objetos**.

El siguiente código muestra el clonado del objeto *\$objT1* en *\$objT2*, de modo que la modificación de sus atributos no afecta a los de *\$objT1*; puesto que son objetos distintos:

```
// Instanciación del objeto $objT1
$objT1 = new tiempo( 10,10,10);
// Duplicación de $objT1 en $objT2
$objT2 = clone $objT1;
// Modificación del atributo $m del objeto referenciado por $objT2.
$objT2->m = 30;

$objT1->mostrar();           // Muestra '10:10:10' -> son objetos distintos
$objT2->mostrar();           // Muestra '10:30:10' -> son objetos distintos
```

Clonado superficial y en profundidad

Se denomina clonado superficial a aquel que sólo duplica los atributos por valor de un objeto, pero **no** los atributos por referencia. Esto significa que si el objeto tiene objetos como atributos, éstos serán referenciados por el objeto clonado.

Ejemplo: Supóngase la clase *geopos* que representa una medida de grados, minutos y segundos, y la clase *posicion* que representa una posición dada una longitud y latitud expresadas como objetos *geopos* y un identificador numérico:

```
class geopos {
    public $g;
    public $m;
    public $s;
    public function __construct($ g, $ m, $ s ) {
        $this->g = $ g;
        if ( $ m >= 0 && $ m <= 59 ) $this->m = $ m;
        if ( $ s >= 0 && $ s <= 59 ) $this->s = $ s;
    }
    public function mostrar() {
        echo "$this->g:$this->m:$this->s<br/>";
    }
}

class posicion {
    public $id;
    public $lat;
    public $lon;
    public function __construct($ id, geopos $ lat, geopos $ lon){
        $this->id = $ id;
        $this->lat = $ lat;
        $this->lon = $ lon;
    }
    public function mostrar() {
        echo "ID: $this->id<br />";
        echo "LAT: ";
        $this->lat->mostrar();
        echo "LON: ";
        $this->lon->mostrar();
    }
}
```

Los objetos de la clase *posicion* tiene por tanto como atributos *\$lat* y *\$lon* objetos de la clase *geopos*.

Ejemplo: Sea el siguiente código en el que se crea un objeto \$pos de la clase posicion, y se clona sobre \$pos2. Posteriormente se modifica el atributo \$id, y los grados de la latitud del objeto \$pos2, y se muestran ambos por pantalla:

```
$lat = new geopos(3,40,50);
$lon = new geopos(40,50,32);
$origen = new posicion( 5, $lat, $lon );
$copia = clone $origen;
$copia->id = 10; // Modificación atributo $id del objeto clonado
$copia->lat->g = 0; // Modificación de grados del atributo $lat de objeto clonado

echo "Original: ";
$origen->mostrar();
echo "Copia: ";
$copia->mostrar();
```

Original: ID: 5
LAT: 0:40:50
LON: 40:50:32
Copia: ID: 10
LAT: 0:40:50
LON: 40:50:32

Los grados de latitud del objeto copia coincide con el objeto original

La copia superficial a duplicado los objetos, por lo que la modificación del atributo \$id del objeto \$copia, no afecta a \$origen; son objetos diferentes. Sin embargo, la modificación de los grados del atributo \$lat de \$copia, ha afectado al atributo \$lat de origen. Eso significa que ambos referencian al mismo objeto en el atributo \$lat.

Este problema es producido por el clonado en superficie, que no duplica los objetos referenciados por los atributos.

Clonado en profundidad (método __clone())

El clonado en profundidad consiste en añadir un método **__clone()** a la clase que se desea clonar para implementar el clonado de sus atributos. Este método se ejecuta automáticamente al invocar el operador **clone** sobre un objeto de la clase.

```
class posicion {
    public $id;
    public $lat;
    public $lon;
    public function __construct($id, geopos $lat, geopos $lon){
        $this->id = $id;
        $this->lat = $lat;
        $this->lon = $lon;
    }
    public function __clone(){
        $this->lat = clone $this->lat; // Clonado del objeto atributo $lat
        $this->lon = clone $this->lon; // Clonado del objeto atributo $lon
    }
    public function mostrar() {
        echo "ID: $this->id<br />";
        echo "LAT: ";
        $this->lat->mostrar();
        echo "LON: ";
        $this->lon->mostrar();
    }
}
```

Si ahora volvemos a ejecutar el código de ejemplo anterior, vemos que la modificación de los grados del atributo *\$lat* del objeto *\$copia*, no afecta a *\$origen*. Ambos objetos han sido clonado, así como los objetos de sus correspondientes atributos *\$lat* y *\$lon*:

```
$lat = new geopos(3,40,50);
$lon = new geopos(40,50,32);
$origen = new posicion( 5, $lat, $lon );
$copia = clone $origen;
$copia->id = 10; // Modificación atributo $id del objeto clonado
$copia->lat->g = 0; // Modificación de grados del atributo $lat de objeto clonado

echo "Original: ";
$origen->mostrar();
echo "Copia: ";
$copia->mostrar();
```

```
Original: ID: 5
LAT: 3:40:50
LON: 40:50:32
Copia: ID: 10
LAT: 0:40:50
LON: 40:50:32
```

Los grados de los atributos \$lat de ambos objetos no coinciden.

Recorrido de objetos (métodos generadores)

Es muy común emplear objetos para almacenar colecciones de valores empleando matrices como atributos. Para obtener los valores de estas matrices pueden recorrerse empleando bucles *foreach()*:

Ejemplo: La clase *números* posee como atributo una matriz en la que se almacenan una serie de valores dados en el constructor:

```
class numeros {
    public $valores;
    public function __construct( array $valores ) {
        $this->valores = $valores;
    }
}
```

El siguiente código crea un objeto *\$obj* de la clase *números* y recorre después los valores almacenados en su matriz mediante un bucle *foreach()*. Sin embargo, esto requiere que el atributo *\$valores* sea público:

```
$obj = new numeros([1,2,3,4,5,6,7]);
// Recorrido de los valores almacenados accediendo a la matriz
foreach( $obj->valores as $valor ) {
    var_dump($valor);
}
```

Para poder recorrer los valores contenidos en la matriz atributo *\$valores* sin necesidad de declararla pública es necesario emplear **métodos generadores**.

Un método generador es un método especial que ofrece una secuencia de recorrible mediante un bucle **foreach()** del mismo modo que si se tratase de una matriz. Para ello el método implementa dentro un bucle que recorre la matriz retornando cada valor de la secuencia mediante la partícula **yield**. Esta partícula retorna el valor indicado pero sin finalizar la ejecución del método generador.

Ejemplo: El siguiente código muestra la implementación de los métodos generadores **todo()** y **pares()**. El método **todos()** devuelve una secuencia con todos los valores contenidos en la matriz. El método **pares()** devuelve una secuencia compuesta por los valores pares contenidos en la matriz:

```
class numeros {
    public $valores;

    public function __construct( array $valores ) {
        $this->valores = $valores;
    }

    // Devuelve secuencia con todos los valores de la matriz
    public function todos() {
        $indx = 0;
        while( $indx < count($this->valores) ) {
            yield $this->valores[$indx]; // Valor devuelto por iteración
            $indx++;
        }
    }

    // Devuelve secuencia con los valores pares de la matriz
    public function pares() {
        foreach( $this->valores as $valor ) {
            if ( $valor % 2 == 0 ) {
                yield $valor; // Valor devuelto por iteración
            }
        }
    }
}
```

La secuencia de valores devuelta por los métodos generadores puede recorrerse empleando un bucle **foreach()** sin problemas:

```
$obj = new numeros([1,2,3,4,5,6,7]);
// Recorrido de todos los valores almacenados en objeto $obj
foreach( $obj->todos() as $valor ) {
    echo "$valor ";
}
echo "<br />";
// Recorrido de los valores pares almacenados en objeto $obj
foreach( $obj->pares() as $valor ) {
    echo "$valor ";
}
```

1 2 3 4 5 6 7
2 4 6

Metadatos

Se denominan metadatos a los datos que pueden obtenerse de un objeto y que reflejan la clase a la que pertenece, sus atributos, sus métodos..., etc.

PHP dispone para estas operaciones de las siguientes funciones: Las siguientes funciones permiten obtener información sobre los atributos y métodos presentes en un determinado objeto:

- `string get_class ([object $object = NULL])`

Devuelve el nombre de la clase a la que pertenece un objeto

- `array get_object_vars (object $object)`

Devuelve una matriz asociativa con el nombre y valor de los atributos accesibles del objeto indicado como argumento. Si se invoca desde un método externo a la clase sólo muestra los atributos públicos. Si se invoca desde un método dentro de la clase muestra también los atributos privados.

- `bool property_exists (mixed $class , string $property)`

Devuelve un valor lógico indicando si el objeto o clase indicada como argumento del primer parámetro (*\$class*), posee el atributo con el nombre indicado en el segundo parámetro (*\$property*).

- `bool method_exists (mixed $object , string $method_name)`

Devuelve un valor lógico indicando si el objeto o clase indicada como argumento del primer parámetro (*\$object*) posee el método de nombre indicado como segundo parámetro (*\$method_name*).

- `array get_class_methods (mixed $class_name)`

Devuelve una matriz con los nombres de los métodos de la clase indicada como argumento.

- `$obj instanceof $class/$interfaz`

Devuelve un valor lógico indicando si el objeto indicado a la izquierda pertenece a la clase indicada a la derecha, o hereda de ella, o implementa el interfaz.

Ejemplo: El siguiente código muestra la declaración de la clase *cuenta*, la instanciación de un objeto de la clase, y el uso de las funciones de obtención de metadatos para obtener información de sus propiedades y métodos.

```
class cuenta {
    public $titular;
    private $saldo = 0;
    public function __construct($ titular, $ saldo) {
        $this->titular = $ titular;
        $this->saldo = $ saldo;
    }
    public function ingresar( $ cantidad) {
        $this->saldo += $ cantidad;
    }
    public function retirar( $ cantidad) {
        $this->saldo -= $ cantidad;
    }
    public function obtenerSaldo() {
        return $this->saldo;
    }
    public function test() {
        var_dump(get_object_vars($this));
    }
};

$obj = new cuenta("Roger Petrov", 1200);

echo "get_class()<br/>";
var_dump(get_class($obj)); // Es $obj de la clase cuenta?
echo "get_object_vars() externo <br />";
var_dump(get_object_vars($obj)); // Obtener atributos externos de $obj
echo "get_object_vars() interno <br />";
$obj->test(); // Obtener atributos internos de $obj
echo "get_class_methods()<br />";
var_dump(get_class_methods($obj)); // Obtiene los métodos de %obj
echo "get_property_exists() titular<br />";
var_dump(property_exists($obj, "titular")); // Posee $obj la propiedad titular?
echo "get_property_exists() saldo<br />";
var_dump(property_exists($obj, "saldo")); // Posee $obj la propiedad saldo?
echo "get_method_exists() retirar<br />";
var_dump(method_exists($obj, "retirar")); // Posee $obj el método retirar()?

get_class()
C:\xampp7\htdocs\prueba\home.php:221:string 'cuenta' (length=6)

get_object_vars() externo
C:\xampp7\htdocs\prueba\home.php:224:
array (size=1)
    'titular' => string 'Roger Petrov' (length=12)

get_object_vars() interno
C:\xampp7\htdocs\prueba\home.php:214:
array (size=2)
    'titular' => string 'Roger Petrov' (length=12)
    'saldo' => int 1200

get_class_methods()
C:\xampp7\htdocs\prueba\home.php:228:
array (size=5)
    0 => string '__construct' (length=11)
    1 => string 'ingresar' (length=8)
    2 => string 'retirar' (length=7)
    3 => string 'obtenerSaldo' (length=12)
    4 => string 'test' (length=4)

get_property_exists() titular
C:\xampp7\htdocs\prueba\home.php:230:boolean true

get_property_exists() saldo
C:\xampp7\htdocs\prueba\home.php:232:boolean true

get_method_exists() retirar
C:\xampp7\htdocs\prueba\home.php:234:boolean true
```


Objetos anónimos

Los objetos convencionales se crean a partir de una clase, la cual define los atributos y métodos del objeto. Todos los objetos de una clase poseen los mismos atributos y métodos definidos en la clase a la que pertenecen. Lo que diferencia a cada objeto es por tanto el valor de sus atributos.

Los objetos anónimos son aquellos que se crean a partir de la clase *stdClass* y se definen sus atributos y valores contra el propio objeto. No se crean a partir de ninguna clase concreta.

```
// Creacion de un objeto anónimo
$obj = new stdClass();

// Declaracion de atributos y valores del objeto anónimo.
$obj->x = 10;
$obj->y = 20;

// Obtención de valores de los atributos del objeto anónimo.
echo $obj->x;           // Muestra 10
echo $obj->y;           // Muestra 20
```

También es posible crear los objetos anónimos a partir de matrices asociativas interpretándose cada valor como un atributo donde la clave es el identificador del atributo:

```
$matriz = ["x"=>10, "y"=>20];           // Matriz asociativa
$objeto = (object)$matriz;               // Conversión a objeto anónimo

var_dump($objeto);
echo "$objeto->x<br/>"; // Muestra 10
echo "$objeto->y<br/>"; // Muestra 20

C:\xampp7\htdocs\prueba\prueba.php:66:
object(stdClass) [1]
  public 'x' => int 10
  public 'y' => int 20

10
20
```

Este tipo de objetos se emplean cuando es necesario crear objetos auxiliares para contener conjuntos de datos sin métodos no definidos en ninguna clase, o para los que no se desea crear ninguna. (por ejemplo; resultados de consultas).

Serialización

Se llama *serialización* al proceso por el que toda la información de un objeto se convierte en texto para poder ser almacenado. Dicho objeto pueden luego recuperarse a partir del texto mediante el proceso inverso denominado *deserialización*.

No debe confundirse serializar un objeto con almacenar sus valores en una matriz, un fichero o una cadena. La serialización NO sólo almacena los valores, también la estructura del propio objeto de modo que pueda recuperarse íntegramente al deserializarlo

Existen múltiples modos de serializar y deserializar que difieren en el modo en el que el objeto es convertido a texto.

Serialización de PHP.

Función *serialize()*

PHP dispone de un mecanismo de serialización y deserialización basado en el uso de las funciones *serialize()* y *unserialize()*:

```
string serialize ( mixed $value )
```

La función *serialize()* recibe una variable como parámetro y retorna un texto con la serialización de dicha variable. Se pueden serializar cualquier tipo de valor, incluidos objetos y matrices de cualquier tipo escalares y asociativas. Únicamente las variables de tipo recurso como los manejadores de ficheros NO pueden serializarse.

Ejemplo: Serialización de un valor simple

```
$x = 10;
$s = serialize($x);
var_dump($s);
```

```
'i:10;'
```

Ejemplo: Serialización de una matriz de valores simples:

```
$valores = [1, 2.45, "HOLA"];
$serializado = serialize($valores);
var_dump($serializado);
```

```
'a:3:{i:0;i:1;i:1;d:2.4500000000000002;i:2;s:4:"HOLA";}'
```

Ejemplo: Serialización de una matriz asociativa de valores simples

```
$valores = ["edad" => 20, "altura" => 1.84, "nombre" => "HOLA"];
$serializado = serialize($valores);
var_dump($serializado);
```

```
'a:3:{s:4:"edad";i:20;s:6:"altura";d:1.8400000000000001;s:6:"nombre";s:4:"HOLA";}'
```

Función *unserialize()*

Esta función permite deserializar una cadena de texto devuelta por la función *serialize()* y recuperar el valor, matriz u objeto serializado.

mixed unserialize (*string \$str* [, *array \$options*])

La función recibe como parámetro la cadena devuelta *serialize()* (*\$str*), y retorna el valor deserializado, que puede ser un entero, decimal, lógico, cadena, objeto, o una matriz de cualquiera de ellos. Si no se puede deserializar la cadena (está corrupta) la función retorna un valor lógico *FALSO*.

Serialización y deserialización básica contra fichero

Habitualmente la serialización y deserialización se emplea para almacenar datos complejos en ficheros para su posterior recuperación.

Ejemplo: Supóngase que contamos con la siguiente clase *alumno* declarada en el fichero “/clases/alumnos.class.php” que representa los datos de un alumno tales como *nombre*, *apellido1*, *apellido2*, *cuenta* y *calificación* obtenida:

```
class alumno {
    private $nombre;
    private $apellido1;
    private $apellido2;
    private $cuenta;
    private $calificacion;
    public function __construct($ nombre, $ apellido1, $ apellido2, $ cuenta,
    $ calificacion ) {
        $this->nombre = $ nombre;
        $this->apellido1 = $ apellido1;
        $this->apellido2 = $ apellido2;
        $this->cuenta = $ cuenta;
        $this->calificacion = $ calificacion;
    }
}
```

A continuación definimos dos funciones que nos ayuden a serializar y deserializar cualquier valor hacia/desde un fichero dado:

- La función *serializeToFile()* serializa y almacena el valor indicado en el parámetro *\$_valores*, en el fichero indicado en *\$_file*. Si no se puede crear el fichero retorna un valor lógico falso, o cierto si todo va bien.

```
function serializeToFile( $ file, $ valores ) {
    // Obtencion ruta raiz
    $document_root = $_SERVER["DOCUMENT_ROOT"];
    // Apertura del fichero
    @$res = fopen("$document_root/$ file", "w");
    if ( $res ) {
        // Serialización
        $serializado = serialize($ valores);
        // Guardado de la cadena serializada
        fwrite($res, $serializado);
        fclose($res);
        return true;
    } else return false;    // EL fichero no pudo crearse
}
```

Programación Web en PHP

- La función `unserializeFromFile()` deserializa el valor almacenado en el fichero indicado en `$_file` y lo retorna como resultado. Si el fichero no se encuentra, o el contenido está corrupto devuelve un valor lógico falso.

```
function unserializeFromFile( $ file ) {  
    // Obtencion ruta raiz  
    $document_root = $_SERVER["DOCUMENT_ROOT"];  
    // Apertura de fichero  
    @$res = fopen("$document_root/$ file", "r");  
    if ( $res ) {  
        // Lectura contenido del fichero  
        $serializado = fread($res, filesize("$document_root/$ file"));  
        // Deserializacion del contenido del fichero y obtención de datos  
        $valor = unserialize($serializado);  
        fclose($res);  
        return $valor;  
    } else return false; // El fichero no pudo abrirse  
}
```

Supóngase que deseamos almacenar en un fichero `"/datos/alumnos.dat"` los datos de una serie de alumnos almacenados en una matriz de objetos `alumno` para recuperarlos posteriormente en un fichero: `"/prueba/puntos.dat"`:

```
require_once "classes/alumno.class.php";  
  
$alumnos[]=new alumno('Alberto','Polanco','Sánchez','albe',7.5);  
$alumnos[]=new alumno('Alvaro','Jiménez','Lopez','alva',6.0);  
$alumnos[]=new alumno('Aranzazu','Pérez','Laborda','aran',3.5);  
$alumnos[]=new alumno('Aritz','Hernández','García','arit', 6);  
$alumnos[]=new alumno('Asier','Pérez','Tamayo','asie', 7);  
$alumnos[]=new alumno('Eder','Moledo','Villalba','eder', 6.5);  
$alumnos[]=new alumno('Eneko','de la Torre','Sánchez','enek', 5.5);  
$alumnos[]=new alumno('Francisco Borja','Navarro','Río','fran', 5);  
$alumnos[]=new alumno('Gaizka','Lorenzo','Jiménez','gaiz', 4.5);  
$alumnos[]=new alumno('Iñaki','Angulo','Tarancón','inak', 2);  
$alumnos[]=new alumno('Iñaki','Demón','Fernandez','inai',7);  
$alumnos[]=new alumno('Iñigo','Casado','Rico','inig', 8);  
$alumnos[]=new alumno('Iratxe','Urriolabeitia','Zabala','irat',9.5);  
$alumnos[]=new alumno('Isaac Jacob','Pazos','López','isaa', 7.5);  
$alumnos[]=new alumno('Jon','Aróstegui','Brizuela','jona', 6);  
$alumnos[]=new alumno('Jon','Elorriaga','Bilbao','jone', 6.5);  
$alumnos[]=new alumno('Lander','Eguskiza','Lamelas','land', 3);  
$alumnos[]=new alumno('Luis Fernando','Coca','Cabrera','luis', 4);  
$alumnos[]=new alumno('Miguel','Rodrigo','Ortega','migu', 5.5);  
$alumnos[]=new alumno('Sergio','Castro','Bravo','serg', 6.5);  
$alumnos[]=new alumno('Xabier','Isla','Rodriguez','xabi', 7);  
// Serializacion de los datos y almacenamiento en fichero.  
$ok = serializeToFile("/datos/alumnos.dat", $alumnos);  
// Comprobacion: Todo OK?  
if ( $ok ) echo "Guardado";
```

Para recuperar los datos almacenados desde otro script podríamos emplear el siguiente código:

```
require_once "classes/alumno.class.php";  
  
$recuperados = unserializeFromFile("/datos / alumnos.dat");  
if ( $recuperados ) {  
    // La matriz $recuperados contiene todos los objetos alumno almacenados previamente.  
} else {  
    echo "No se pudo abrir fichero";  
}
```

Serialización con JSON

JSON es un formato de intercambio de datos (*Javascript Object Notation*) que permite representar estructuras complejas de datos tales como matrices y objetos mediante texto. Este se basa en la sintaxis del lenguaje de programación Javascript, y se emplea como medio de intercambio y almacenaje de datos multiplataforma entre ficheros, bases de datos y servicios web. De ahí la importancia de saber codificar y decodificación datos en JSON.

A partir de la versión 5.2 de PHP, el soporte de JSON forma parte de PHP y no es necesario instalar ninguna extensión adicional. Puede comprobarse si la extensión de JSON está presente en PHP por la página de información (*phpinfo()*)

json

json support	enabled
json version	1.4.0

Vista de extensión JSON instalada en la ventana de información de PHP

Función *json_encode()*

Esta función devuelve la cadena con el código JSON correspondiente el valor indicado como argumento (*\$value*) que puede ser de cualquier tipo exceptuando de tipo recurso (*resource*).

```
string json_encode ( mixed $value
                    [, int $options = 0
                    [, int $depth = 512 ]] )
```

El parámetro *\$options* permite configurar el modo en que se codifican los datos. Los valores posibles están definidos como constantes. Uno de esos valores es **JSON_PRETTY_PRINT** que fuerza el formateo del código JSON generado para facilitar su visualización.

Ejemplo: Codificación de valores simples:

<pre>\$numero = 10; \$nombre = "pepe"; \$altura = 1.77; \$activo = true; var_dump(json_encode(\$numero)); var_dump(json_encode(\$nombre)); var_dump(json_encode(\$altura)); var_dump(json_encode(\$activo));</pre>
10
"pepe"
1.77
true

Ejemplo: Codificación de matrices escalares

```
$valores = [ 1, 3, 4, 10, 20];  
$nombres = [ "Roger", "Ivan", "Petrovich"];  
var_dump( json_encode($valores));  
var_dump( json_encode($nombres));
```

```
[1,3,4,10,20]
```

```
["Roger","Ivan","Petrovich"]
```

Ejemplo: Codificación de objetos. Supóngase una clase alumno con la siguiente definición:

```
class alumno {  
    public $nombre;  
    public $apellido1;  
    public $apellido2;  
    public $cuenta;  
    public $calificacion;  
  
    public function __construct($ nombre, $ apellido1, $ apellido2, $ cuenta,  
    $ calificacion ) {  
        $this->nombre = $ nombre;  
        $this->apellido1 = $ apellido1;  
        $this->apellido2 = $ apellido2;  
        $this->cuenta = $ cuenta;  
        $this->calificacion = $ calificacion;  
    }  
}
```

El siguiente código codifica en JSON un objeto de la clase alumno (\$obj);

```
$obj = new alumno("Roger", "Petrov", "Demidovich", "petrus", 6.5);  
var_dump(json_encode($obj));
```

```
{  
    "nombre": "Roger",  
    "apellido1": "Petrov",  
    "apellido2": "Demidovich",  
    "cuenta": "petrus",  
    "calificacion": 6.5  
}
```

Ejemplo: Codificación de matrices asociativas:

```
$valores = [ "Ivan" => 20, "Yuri" => 23, "Antonov" => 45, "Pavel" => 12];  
var_dump( json_encode($valores));
```

```
{  
    "Ivan": 20,  
    "Yuri": 23,  
    "Antonov": 45,  
    "Pavel": 12  
}
```

La codificación en JSON de las matrices asociativas y los objetos es idéntica:

- Las matrices asociativas codifican cada valor indicando la clave entre comillas dobles seguida de dos puntos y el valor correspondiente. Los diferentes valores se separan por comas.
- Los objetos codifican cada atributo público indicando el nombre del atributo entre comillas dobles seguido de dos puntos y el valor correspondiente. Los diferentes atributos se separan por comas excepto el último.
- En ambos casos la declaración va contenida entre llaves {}

Función `json_decode()`

La función `json_decode()` realiza el proceso inverso convirtiendo la cadena codificada en JSON dada como parámetro en el valor codificado:

```
mixed json_decode ( string $json  
                    [, bool $assoc = false  
                    [, int$depth = 512  
                    [, int $options = 0 ]]] )
```

El parámetro `$json` representa la cadena codificada en JSON.

Ejemplo: Decodificación de valores simples:

```
$cadena1 = '10'; // entero en JSON  
$cadena2 = '"hola"'; // cadena en JSON  
$cadena3 = '20.45'; // decimal en JSON  
$cadena4 = 'true'; // lógico en JSON  
var_dump(json_decode($cadena1));  
var_dump(json_decode($cadena2));  
var_dump(json_decode($cadena3));  
var_dump(json_decode($cadena4));
```

```
C:\xampp7\htdocs\prueba\prueba.php:40:int 10
```

```
C:\xampp7\htdocs\prueba\prueba.php:41:string 'hola' (length=4) ▶
```

```
C:\xampp7\htdocs\prueba\prueba.php:42:float 20.45
```

```
C:\xampp7\htdocs\prueba\prueba.php:43:boolean true
```

Ejemplo: Decodificación de matrices escalares (*no asociativas*)

```
$cadena5 = '[1,2,3,4,5,6,7,8]'; // matriz escalar de enteros en JSON  
$cadena6 = '["uno","dos","tres"]'; // matriz escalar de cadenas en JSON  
var_dump(json_decode($cadena5));  
var_dump(json_decode($cadena6));
```

```
C:\xampp7\htdocs\prueba\prueba.php:44:  
array (size=8)  
    0 => int 1  
    1 => int 2  
    2 => int 3  
    3 => int 4  
    4 => int 5  
    5 => int 6  
    6 => int 7  
    7 => int 8
```

```
C:\xampp7\htdocs\prueba\prueba.php:45:  
array (size=3)  
    0 => string 'uno' (length=3)  
    1 => string 'dos' (length=3)  
    2 => string 'tres' (length=4)
```

Ejemplo: Decodificación de matrices asociativas/objetos

La decodificación de matrices asociativas y objetos es equivalente puesto que ambos tipos de datos se codifican igual en JSON. Para indicar si deseamos obtener el valor como objeto o matriz asociativa se emplea el parámetro *\$assoc*.

Si se indica un valor lógico **cierto** para el parámetro *\$assoc*, se fuerza la decodificación de los datos como una matriz asociativa:

```
$cadena = '{"x":10, "y":20}'; // objeto en JSON
// Decodificación como matriz asociativa
$matriz = json_decode($cadena, true);
var_dump($matriz);
echo $matriz['x'];
echo $matriz['y'];

C:\xampp7\htdocs\prueba\prueba.php:55:
array (size=2)
  'x' => int 10
  'y' => int 20
```

Si se indica un valor lógico **falso** para el parámetro *\$assoc*, o se omite; la decodificación se realiza devolviendo un objeto anónimo con los atributos indicados:

```
$cadena = '{"x":10, "y":20}'; // objeto en JSON
// Decodificación como objeto
$objeto = json_decode($cadena);
var_dump($objeto);
echo $objeto->x;
echo $objeto->y;

C:\xampp7\htdocs\prueba\prueba.php:49:
object(stdClass) [1]
  public 'x' => int 10
  public 'y' => int 20
```

(*) A diferencia de la serialización predeterminada de PHP, la decodificación de objetos en JSON no requiere la importación de la clase original, ya que los objetos son decodificados siempre como anónimos.

Diferencias entre Javascript y JSON

El formato JSON se basa en la sintaxis de Javascript, no obstante; existen ciertas construcciones que si bien son válidas en Javascript no lo son en JSON.

- Los valores de tipo cadena se codifican con comillas dobles. Las comillas simples no son válidas.
- Los nombres de atributos/claves deben indicarse con comillas dobles seguidas de dos puntos y el valor correspondiente.
- Cada valor se separa del anterior mediante una coma. El último valor de la secuencia no debe llevarla.

Manejo de errores

La codificación a JSON puede presentar problemas en ciertas ocasiones. JSON únicamente soporta la codificación de cadenas en formato UTF8. En caso de no ser así, la función `json_encode()` retorna un valor lógico falso como resultado.

Para evitar este problema puede emplearse la función `utf8_encode()` para obtener la codificación en UTF-8 de una cadena con otra codificación.

La decodificación de JSON también puede presentar problemas si el formato no es correcto. En esa situación la función `json_decode()` retorna un valor NULL.

Para conocer el tipo de error sucedido en la decodificación pueden emplearse las funciones `json_last_error()` y `json_last_error_msg()`.

```
int json_last_error ( void )
```

La función `json_last_error()` no recibe ningún parámetro y devuelve un valor entero indicando el último error de decodificación sucedido. Los posibles valores están definidos por las siguientes constantes:

Constante	Significado	Disponibilidad
JSON_ERROR_NONE	No ocurrió ningún error	
JSON_ERROR_DEPTH	Se ha excedido la profundidad máxima de la pila	
JSON_ERROR_STATE_MISMATCH	JSON con formato incorrecto o inválido	
JSON_ERROR_CTRL_CHAR	Error del carácter de control, posiblemente se ha codificado de forma incorrecta	
JSON_ERROR_SYNTAX	Error de sintaxis	
JSON_ERROR_UTF8	Caracteres UTF-8 mal formados, posiblemente codificados de forma incorrecta	PHP 5.3.3
JSON_ERROR_RECURSION	Una o más referencias recursivas en el valor a codificar	PHP 5.5.0
JSON_ERROR_INF_OR_NAN	Uno o más valores <code>NAN</code> o <code>INF</code> en el valor a codificar	PHP 5.5.0
JSON_ERROR_UNSUPPORTED_TYPE	Se proporcionó un valor de un tipo que no se puede codificar	PHP 5.5.0

Adicionalmente, también es posible obtener un mensaje descriptivo del último error sucedido mediante la función `json_last_error_msg()`.

```
string json_last_error_msg ( void )
```

Esta función no recibe tampoco ningún parámetro y devuelve una cadena con la descripción del último error de decodificación sucedido.

Prácticas

Vamos a crear una aplicación web que permita la creación y almacenamiento de facturas de un comercio de alimentación delicatessen.

Los datos de los productos.

Los productos a la venta están definidos en un fichero codificado en formato JSON que debe almacenarse con el nombre *productos.json* dentro de una carpeta *datos*. El código del fichero es el siguiente:

```
{
  "36114": {
    "id": 36114,
    "producto": "T\u00e9 Dharamsala",
    "categoria": "Bebidas",
    "precio": 18
  },
  "36242": {
    "id": 36242,
    "producto": "Cerveza tibetana Barley",
    "categoria": "Bebidas",
    "precio": 19
  },
  ...
}
```

El fichero codifica una matriz asociativa de productos empleando como clave el identificador del producto. Cada producto es representado a su vez por una matriz asociativa con el identificador del producto (*id*), el nombre (*producto*), la categoría a la que pertenece (*categoría*), y el precio de venta por unidad (*precio*).

Crear una clase **producto** (*producto.class.php*). Esta debe almacenarse en una subcarpeta *entidades* con los siguientes miembros:

```
class producto {
    public $id;
    public $nombre;
    public $categoria;
    public $precio;
    public $unidades;
    public $importe;

    public function __construct( $ id, $ producto, $ categoria, $ precio )

    public function compra( $ unidades ) {
        $this->unidades = $ unidades;
        $this->importe = $this->unidades * $this->precio;
    }
}
```

El método **calcularImporte()** recibe el nº de unidades adquiridas y calcular el importe de venta almacenándolo en el atributo *\$importe* y retornando el resultado.

Crea una clase **datos** (*datos.class.php*). Esta debe almacenarse en la carpeta *datos* y debe constar por el momento de una función estática *obtenerProducto(\$_id)* que recibe como argumento el identificador de un producto y devuelva el objeto producto correspondiente registrado en el fichero *productos.json* o NULL si no existe.

```
class datos {
    public static function obtenerProducto( $ _id ) {
        $producto = null;
        $document_root = $_SERVER["DOCUMENT_ROOT"];
        $json = file_get_contents("$document_root/prueba/data/datos.json");
        // decodificación de json como matriz asociativa
        $datos = json_decode($json, true);
        // Existe la clave indicada como identificador?
        if ( array_key_exists($ _id, $datos) ) {
            // Obtengo array con los valores del objeto producto
            $array_producto = $datos[$ _id];
            // Obtengo el objeto producto
            $objeto_producto =
                new producto($array_producto['id'],
                            $array_producto['producto'],
                            $array_producto['categoria'],
                            $array_producto['precio']);
        }
        return $objeto_producto;
    }
}
```

Los controles de formulario

Recupera las clases **textbox** y **validatebox** y el interfaz **htmlrenderable** creados en ejercicios anteriores y almacénalas en los ficheros *textbox.class.php* y *validatebox.class.php* de la carpeta *controls* respectivamente.

```
class textbox implements htmlrenderable {
    protected $id;
    protected $valor;
    public function __construct($ id, $ valor = "") {
        $this->id = $ id;
        $this->valor = (isset($_REQUEST[$this->id]))?$_REQUEST[$this->id]:$ valor;
    }
    public function esVacio() { return $this->valor == ""; }
    public function getValor() {return $this->valor;}
    public function setValor( $ valor) {$this->valor = $ valor; }
    public function html() {
        echo "<input type='text' name='$this->id' value='$this->valor'>";
    }
}

class validatebox extends textbox {
    protected $error_longitud = false;
    protected $error_numerico = false;
    protected $longitud;

    public function __construct($ id, $ longitud = 9999, $ valor = "") {
        parent::__construct($ id, $ valor);
        $this->longitud = $ longitud;
        if ( isset($_REQUEST[$this->id]) ) {
            $this->error_longitud = (strlen( $this->valor ) > $this->longitud);
            $this->error_numerico = !is_numeric($this->valor);
        }
    }
    public function esValido() {return !$this->error_longitud && !$this->error_numerico; }
    public function html() {
        parent::html();
        if ( $this->error_longitud ) echo "LONGITUD NO VALIDA";
        else if ( $this->error_numerico) echo "VALOR NO VALIDO";
    }
}
```

Crea un nuevo control llamado **tabla** (*tabla.class.php*) que implemente la interfaz *htmlrenderable* y almacénalo en la carpeta *controls*. Este control debe representar una tabla HTML con los datos contenidos en una matriz de objetos indicando el nombre de los atributos que deseamos ver en cada columna:

```
class tabla implements htmlrenderable {
    protected $id;
    protected $columnas = array();
    protected $datos = array();

    public function __construct( $ _id, $ _datos, $ _columnas)
    public function html()
}
```

El constructor debe recibir como parámetros un identificador (*\$ _id*), una matriz escalar de objetos a mostrar(*\$ _datos*), y una matriz escalar de cadenas con los atributos que desean mostrarse en cada columna (*\$ _columnas*).

Ejemplo: si queremos mostrar en la tabla tres objetos *producto* almacenados en una matriz, el código sería el siguiente:

```
require_once 'classes/controls.class.php';
require_once 'entidades/producto.class.php';
$productos[] = new producto(1023, "producto_1", "categoria_A", 100.40);
$productos[] = new producto(3045, "producto_2", "categoria_A", 220.50);
$productos[] = new producto(5905, "producto_3", "categoria_A", 195.45);
// Parámetros
// 1.- ID de la tabla
// 2.- Matriz de nombres de columnas coincidente con atributos de la clase producto
// 3.- Matriz de objetos producto a representar
$tbl = new tabla("tbl", ["id", "producto", "categoria", "precio"], $productos);
$tbl->html();
```

id	producto	categoria	precio
1023	producto_1	categoria_A	100.4
3045	producto_2	categoria_A	220.5
5905	producto_3	categoria_A	195.45

Crea un control **boton** (*boton.class.php*) que implemente la interfaz *htmlrenderable* y almacénalo en la carpeta *controls*. Este control debe representar un botón de envío de formulario `<input type='submit'>`. La clase debe constar de los siguientes miembros:

- Un constructor que recibe un identificador (*\$ _id*) cuyo valor se indica en el atributo *name* del elemento HTML, y el parámetro (*\$ _valor*) con el valor mostrado p el botón.
- Un método *fuePulsado()* que devuelve un valor lógico que indica si el botón ha sido pulsado tras recargarse el formulario comprobando para ello si existe la clase dada como identificador en la matriz *\$ _REQUEST*.

```
class boton implements htmlrenderable {
    protected $id;
    protected $valor;
    protected $pulsado;
    public function __construct( $ _id, $ _valor = null )
    public function fuePulsado()
}
```

Crea un nuevo control llamado **contexto** (*contexto.class.php*) que implemente igualmente la interfaz *htmlrenderable* y almacénalo en la carpeta *controls*.

El constructor recibe como atributos el identificador (*\$_id*) y opcionalmente un valor (*\$_valor*) que puede ser cualquier tipo de dato (entero, cadena, decimal, lógico, matriz, objeto... etc). Los métodos miembros *getValor()* y *setValor()* permite obtener y asignar el valor a almacenar en el control respectivamente.

```
class contexto implements htmlrenderable {
    protected $id;
    protected $valor;
    public function __construct($ id, $ valor = null)

    public function getValor()
    public function setValor( $ valor)
}
```

Este control debe mantener su valor a través de las recargas del formulario igual que los controles **textbox** o **listbox** implementados anteriormente. Para ello deben seguirse los siguientes pasos:

- En la sobrescritura del método *html()* del interfaz *htmlrenderable*, debe serializarse (función *serialize()*) el contenido del atributo *\$valor* e inscribirlo como valor del atributo 'value' del elemento *<input type='hidden'>*.

```
public function html() {
    echo "<input type='hidden' name='$this->id'
        value='".serialize($this->valor)."'>";
}
```

- En el constructor debe obtenerse el valor del campo de la matriz *\$_REQUEST* si es que existe, y deserializarlo (función *unserialize()*) en el atributo *\$valor* para recuperar el valor original. Si no existe debe tomar el indicado como parámetro al constructor:

```
public function __construct($ id, $ valor = null) {
    $this->id = $ id;
    $this->valor = (isset($_REQUEST[$this->id]))
        ?unserialize($_REQUEST[$this->id])
        :$ valor;
}
```

El formulario de compra

Crea una página **formulario_pedido.php** situada en la raíz del proyecto que muestre un formulario en el que el vendedor indique los productos y unidades adquiridas y se vaya mostrando el importe parcial y total de la compra.

El formulario debe constar de los siguientes elementos:

- Un control **textbox** que implemente una caja de texto para indicar el nombre del comprador.
- Un control **textbox** que implemente una caja de texto para indicar su dirección de correo electrónico.
- Un control **validatebox** que implemente una caja de texto que admita un valor numérico de máximo 5 cifras para indicar el identificador del producto adquirido.
- Un control **validatebox** que implemente una caja de texto que admita un valor numérico de máximo 2 cifras para indicar la cantidad de unidades adquiridas.
- Un control **tabla** que implemente una tabla donde se mostrarán los datos de los productos adquiridos hasta el momento señalando:
 - Nombre del producto
 - Unidades adquiridas
 - Precio por unidad
 - Importe
- El valor del importe total de la compra.
- Un control **botón** con el texto “AÑADIR” para agregar el producto indicado comprobando que el identificador y nº de unidades sean valores válidos y exista el producto indicado.
- Un control **botón** con el texto “REGISTRAR” para registrar la compra comprobando que se ha indicado el nombre del comprador y su dirección de correo electrónico.

COMPRADOR

NOMBRE:

E-MAIL:

PEDIDO

PRODUCTO:

CANTIDAD:

producto	categoria	precio	unidades	importe
Salsa de arándanos Northwoods	Condimentos	40	2	80
Sirope de regaliz	Condimentos	10	1	10
Postre de merengue Pavlova	Repostería	17	1	17

IMPORTE FINAL: 107 €

Programación Web en PHP

Se recomienda estructurar el código del formulario del siguiente modo:

```
<?php
    $error_producto_desconocido = ""; // mensaje de error de identificador de producto
    desconocido
    $error_comprador = ""; // mensaje de error de comprador no indicado
    $error_mail = ""; // mensaje de error de email no válido
    $sumatorio = 0; // sumatorio de importe final de compra

    // Declaracion de controles de formulario
    $txt_comprador = new textbox("txt_comprador");
    $txt_email = new textbox("txt_email");
    $txt_id = new validatebox("txt_id", 5);
    $txt_cantidad = new validatebox("txt_cantidad", 2);
    $context = new contexto("ctx");
    $tbl_compra = new tabla("tbl_compra",
        ["producto", "categoria", "precio", "unidades", "importe"]);
    $btn_add = new boton("btn_add", "AÑADIR");
    $btn_save = new boton("btn_registrar", "REGISTRAR");

    // Recuperacion de la matriz de compra
    if ( $context->getValor() == NULL ) {
        // Ningun producto -> matriz vacia
        $lista_compra = array();
    } else {
        // Recuperacion
        $lista_compra = $context->getValor();
    }

    // ¿Se pulso el botón añadir?
    if ( $btn_add->fuePulsado() ) {
        //
        // AGREGADO DE OBJETO PRODUCTO INDICADO A $lista_compra
        //
    }

    // ¿Se pulso el botón registrar?
    if ( $btn_save->fuePulsado() ) {
        //
        // CREACION DE OBJETO COMPRA Y REGISTRO.
        //
    }

    // Guardado de la matriz de compra actualizada
    $context->setValor($lista_compra);
    // Visualizacion de la matriz de compra
    $tbl_compra->setValores($lista_compra);
?>

<!DOCTYPE html>
<html>
<head>
<title>PEDIDO</title>
</head>
<body>
    <form method='post'>
        <?php $context->html(); ?>
        <h1>COMPRADOR</h1>
        NOMBRE: <?php $txt_comprador->html(); echo $error_comprador; ?><br/>
        E-MAIL: <?php $txt_email->html(); echo $error_mail; ?>
        <h1>PEDIDO</h1>
        PRODUCTO: <?php $txt_id->html(); echo $error_producto_desconocido; ?><br/>
        CANTIDAD: <?php $txt_cantidad->html(); ?><br />
        <?php $tbl_compra->html(); echo "<h2>IMPORTE FINAL: $sumatorio €</h2>"; ?>
        <?php $btn_add->html(); ?><?php $btn_save->html(); ?>
    </form>
</body>
</html>
```

Registro de compras

Crea una clase **compra** (*compra.class.php*) en la carpeta *entidades*. La clase debe disponer de los siguientes miembros:

```
class compra{
    public $comprador;    // Nombre del comprador
    public $mail;         // Dirección de correo electrónico
    public $productos;    // Matriz de objetos producto adquiridos

    public function __construct($ _comprador, $ _mail, $ _productos)
    public function calcularImporte()
    public function calcularImporteIVA( $ _iva )
```

- El constructor recibe como parámetros el nombre del comprador (*\$_comprador*), su dirección de correo electrónico (*\$_mail*), y la matriz de objetos producto adquiridos (*\$_productos*).
- La función **calcularImporte()** devuelve el importe bruto de la compra, esto es; el sumatorio del importe de productos contenidos en la matriz *\$productos*. La función **calcularImporteIva()** devuelve el importe incluyendo el porcentaje de impuesto indicado como parámetro *\$_iva*.

En la clase **datos** añade una nueva función *estática registrarCompra()*:

```
public static function registrarCompra( compra $_compra )
```

Esta función recibe como parámetro un objeto de la clase compra y lo almacena en un fichero codificado en *JSON*. Cada compra debe registrarse en un archivo cuyo nombre será el valor devuelto por la función **time()** de PHP y extensión *".json"*. Los archivos deben almacenarse en una carpeta *compras*:

```
{
    "comprador": "Roger Piskunova",
    "mail": "none@none.com",
    "productos": [
        {
            "id": 32718,
            "producto": "Salsa de ar\u00e9ndanos Northwoods",
            "categoria": "Condimentos",
            "precio": 40,
            "unidades": "2",
            "importe": 80
        },
        {
            "id": 39022,
            "producto": "Sirope de regaliz",
            "categoria": "Condimentos",
            "precio": 10,
            "unidades": "1",
            "importe": 10
        },
        {
            "id": 35089,
            "producto": "Postre de merengue Pavlova",
            "categoria": "Reposter\u00eda",
            "precio": 17,
            "unidades": "1",
            "importe": 17
        }
    ]
}
```


CLIPSA