cipsa.net

PHP & MySQL

Anexo 5.- 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

Fundamentos de POO

La programación orientada a objetos representa una nueva metodología en la estructuración y desarrollo de aplicaciones. Esta metodología centra el diseño de la aplicación en los datos manejados en vez de en las tareas a realizar.

Para ello se divide la aplicación en clases. Una clase puede representar un tipo de información (mensajes, personas, artículos), o componentes (una plantilla, una página, un acceso a una base de datos o a un fichero). Las tareas de la aplicación se definen mediante métodos en las clases que componen la aplicación.

Conceptos de Clase y Objeto

- *"Una clase es un tipo que representa el diseño de una información o componente de la aplicación."*

Si se supone una aplicación que administre la información de los *empleados* de una empresa debe definirse una clase *Empleado*. Esta clase debe indicar el conjunto de datos y tareas relacionadas con cada empleado almacenado en la aplicación (nombre, apellidos, edad…, etc).

De igual manera pueden definirse clases para representar cualquier tipo de componente: páginas, controles, conexiones a bases de datos, formularios… etc.

Clase EMPLEADO
- Nombre
- Apellido
- Edad
- Salario

- *"Un objeto es un dato o componente concreto. Los atributos contienen la información específica del dato o componente representado y sus métodos permiten manipularlo."*

Si se supone una aplicación que administra la información de los empleados de una empresa, ésta manejará tantos objetos de la clase *Empleado* como empleados haya en la empresa. Todos los objetos de la clase Empleado tendrán los atributos definidos en la clase, pero con sus propios valores nombre, apellido, edad…., etc.

Objeto EMPLEADO
- Nombre -> Roger
- Apellido -> Petrov
- Edad -> 25
- Salario -> 21.000€

Declaración de una clase

La declaración de una clase en PHP se realiza a partir de un script de PHP vacío indicando la palabra clave ***class*** seguida del nombre de la clase:

Ejemplo: Código del fichero "*cuenta.class.php*".

(*) El nombre de la clase se indica en minúsculas sin espacios ni guiones.

```php
<?php
class cuenta {
        // Atributos

        // Métodos
}
```

Es costumbre que el nombre del script coincida con el de la clase, y añadir una doble extensión **.class.php*, para diferenciarlo de otros scripts que no definen clases.

Atributos

Los atributos representan los datos que desean almacenarse por cada objeto. En el caso de una cuenta bancaria éstos serían: *titular*, *saldo*, *interés*…, etc. Estos se declaran al principio de la clase como una variable normal precedida de un *modificador de visibilidad* y un valor inicial si se quiere.

```php
<?php
class cuenta {

        public $titular;
        public $saldo = 0;
        public $interes = 0;
}
```

(*) La palabra clave "public" es un modificador de visibilidad. Se explica más adelante.

Métodos

A continuación se definen los métodos que representan el conjunto de operaciones que pueden invocarse contra un objeto de esa clase. Estos se declaran como funciones convencionales empleando la palabra clave ***function***:

```php
<?php
class cuenta {

        // Atributos
        public $titular;
        public $saldo = 0;
        public $interes = 0;

        // Métodos
        function ingresar( $_cantidad ) {}
        function retirar( $_cantidad ) {}
}
```

En el caso de una clase cuenta bancaria los métodos más obvios serían *ingresar()* y *retirar()* capital.

Constructores y Destructores

Un *constructor* es un tipo especial de método que se ejecuta al crear un objeto. La misión de los constructores es recibir valores para asignarlos a los atributos de un objeto desde el momento en que es creado. Un objeto cuyos atributos no tienen valor ni identifica ni representa nada.

El constructor se declara con el identificador: "***__construct***" y recibe típicamente tantos parámetros como atributos tiene la clase:

```php
<?php
class cuenta {

        // Atributos
        public $titular;
        public $saldo = 0;
        public $interes = 0;

         // Constructor
         function __construct( $_titular, $_saldo, $_interes) {
                 echo "OBJETO CUENTA INSTANCIADO<br/>";
                 echo "Titular: $_titular<br/>";
                 echo "Saldo: $_saldo<br/>";
                 echo "Interes: $_interes<br />";
         }
        // Métodos
        function ingresar( $_cantidad ) {}
        function retirar( $_cantidad ) {}
}
```

El *destructor* es un tipo especial de método que se ejecuta cuando un objeto es eliminado de la memoria. Por defecto *todos los objetos creados por un script de PHP son automáticamente eliminados al termina su ejecución con el envío de la página resultante al navegador del usuario*.

El método destructor se define con el identificador: *"__destruct"*, pero a diferencia del constructor *NO recibe ningún parámetro* ya que es invocado por el sistema:

```php
<?php
class cuenta {

        // Atributos
        public $titular;
        public $saldo = 0;
        public $interes = 0;

        // Constructor
        function __construct( $_titular, $_saldo, $_interes) {
                echo "OBJETO CUENTA INSTANCIADO<br/>";
                echo "Titular: $_titular<br/>";
                echo "Saldo: $_saldo<br/>";
                echo "Interes: $_interes<br />";
        }
         // Destructor
         function __destruct() {
                 echo "OBJETO CUENTA ELIMINADO<br/>";
         }
        // Métodos
        function ingresar( $_cantidad ) {}
        function retirar( $_cantidad ) {}
}
```

Instanciación

Una vez declarada una clase es posible crear objetos o *instancias* de la misma. Esta operación recibe el nombre de *instanciación*. Para ello se declara una variable y emplea la palabra clave *new* indicando a continuación el nombre de la clase.

(*) Si el script en el que se crea el objeto es distinto del que contiene la declaración de la clase, ésta debe incluirse empleando los comandos ***requiere***, ***include***.

Ejemplo: Instanciación de un objeto cuenta bancaria con "Roger Petrov" como titular, 1200.50€ de saldo y un interés fijado al 0.25%. La clase cuenta está definida en el script "*cuenta.class.php*":

```php
<?php
	include_once "cuenta.class.php";

	// Instanciacion de la clase cuenta.
	$obj = new cuenta("Roger Petrov", 1200.50, 0.25);
```

Para instanciar una clase *deben indicarse tantos argumentos como parámetros requiere el constructor* de la clase.

Ejemplo: Al ejecutar el código anterior, el resultado mostrado al navegador del usuario es el siguiente:

```
OBJETO CUENTA INSTANCIADO
Titular: Roger Petrov
Saldo: 1200.5
Interes: 0.25
OBJETO CUENTA ELIMINADO
```

Se instancia un objeto en la variable *$obj* provocando la ejecución del métodos constructor que muestra los valores indicados como argumentos. Al llegar la ejecución al fin del script se elimina el objeto creado ejecutándose el método destructor.

Instanciación sin constructor

Si la clase *no contase con constructor*, la instanciación tendría la siguiente forma:

```php
	$obj = new cuenta();
```

En este caso, el objeto cuenta bancaria creado no tendría ningún valor en sus atributos por lo que no se sabe que cuenta representa. Esto no se recomienda.

Acceso a los atributos

El acceso y manipulación de los atributos de una clase desde sus propios métodos se realiza indicando el puntero especial ***$this*** seguido del operador "**->**" y el nombre del atributo. Esto puede emplearse desde el constructor o cualquier otro método de la clase para obtener o modificar el valor de los atributos:

Código: *cuenta.class.php*

```php
<?php
class cuenta {
        // Atributos
        public $titular;
        public $saldo = 0;
        public $interes = 0;

        // Constructor
        function __construct( $_titular, $_saldo, $_interes) {
                $this->titular = $_titular;
                $this->saldo = $_saldo;
                $this->interes = $_interes;
                echo "Creado objeto cuenta de $this->titular<br/>";
        }
        // Destructor
        function __destruct() {
                echo "Eliminado objeto cuenta de $this->titular<br/>";
        }

        // Ingresar -> Incrementa el saldo en la cantidad indicada
        function ingresar( $_cantidad ) {
                $this->saldo += $_cantidad;
        }
        // Retirar --> Elimina del saldo la cantidad indicada
        function retirar( $_cantidad ) {
                $this->saldo -= $_cantidad;
        }
}
```

También es posible obtener y modificar el valor de los atributos de una clase desde fuera *siempre que estén declarados públicos (**public**)*. Para ello se indica la variable que representa el objeto, seguido del operador "->" y el nombre del atributo.

Código: home.php

```php
<?php

        include_once "classes/cuenta.class.php";
        // Instanciacion de la clase cuenta.
        $obj = new cuenta("Roger Petrov", 1200.50, 0.25);
?>
<!DOCTYPE html>
<html>
<head>
<title>Insert title here</title>
</head>
<body>
        <div>
                <p>Titular: <?php echo $obj->titular; ?></p>
                <p>Saldo: <?php echo $obj->saldo; ?></p>
                <p>Interes: <?php echo $obj->interes; ?></p>
        </div>
</body>
</html>
```

El resultado mostrado en el navegador al ejecutar la página "*home.php*" es:

```
Creado objeto cuenta de Roger Petrov

Titular: Roger Petrov

Saldo: 1200.5

Interes: 0.25

Eliminado objeto cuenta de Roger Petrov
```

Llamada a los métodos

Los métodos de una clase pueden invocarse también desde fuera de la misma indicando el objeto seguido del operador "**->**" y el método a llamar.

```php
<?php

        include_once "classes/cuenta.class.php";
        // Instanciacion de la clase cuenta.
        $obj = new cuenta("Roger Petrov", 1200.50, 0.25);
        // Saldo original
        echo "Saldo inicial: $obj->saldo<br/>";
        // Llamada al método ingresar
        $obj->ingresar(1050.20);
        echo "Saldo tras ingreso: $obj->saldo<br/>";
        // Llamada al método retirar
        $obj->retirar(500.0);
        echo "Saldo tras retirada: $obj->saldo<br/>";
```

```
Creado objeto cuenta de Roger Petrov
Saldo inicial: 1200.5
Saldo tras ingreso: 2250.7
Saldo tras retirada: 1750.7
Eliminado objeto cuenta de Roger Petrov
```

Encapsulación

El principio de encapsulación es un fundamento de la programación orientada a objetos que exige que el diseño de una clase sea tal que permita:

- Instanciar y manipular objetos de una clase conociendo únicamente sus atributos y métodos *pero no su programación*.

- Poder obtener y modificar los valores de sus atributos *de manera segura sin permitirse la asignación de valores erróneos o incongruentes*.

Para ello se emplean los *modificadores de visibilidad*.

Modificadores de visibilidad

Los modificadores de visibilidad son palabras claves que determinan desde dónde son accesibles los métodos y atributos de una clase:

Modificador	Accesibilidad
Público (*public*)	Los métodos y atributos públicos son accesibles desde los métodos de la propia clase, las clases hijas, y a través de sus objetos.
Privado (*prívate*)	Los métodos y atributos privados únicamente son accesibles desde el resto de métodos de la propia clase.
Protegido (*protected*)	Los métodos y atributos protegidos son accesibles desde los métodos de la propia clase y de las clases hijas.

- Tanto las funciones como los atributos pueden declararse indicando primero el modificador de visibilidad. ***Si no se indica ninguno se toma como público***.

- Los atributos deben declararse privados para impedir el acceso libre a los mismos, y declarar funciones públicas o funciones de acceso para permitir obtener y modificar de manera controlada el valor de los atributos.

- Las funciones deben declararse públicas cuando representar tareas principales del objeto. Las funciones que implementan tareas auxiliares deben declararse privadas.

Funciones de acceso

La encapsulación desaconseja la declaración de atributos públicos, ya que permite que asignarles cualquier valor sin control.

Sea la clase *empleado*

```php
<?php
class persona
{
        public $nombre;
        public $apellido;
        public $edad;
}
```

Al tener declarado como *público* el atributo *$edad*, es posible asignarle cualquier valor, como por ejemplo; un número negativo:

```
$obj = new persona();
$obj->edad = -100;          // Valor sin sentido!!!!
```

Para evitar estos problemas y encapsular correctamente la clase podemos definir funciones públicas que permitan obtener y modificar los atributos de manera correcta.

Ejemplo: El siguiente código muestra las funciones *setEdad()* y *getEdad()* que permiten modificar y obtener el valor del atributo *$edad*:

```
<?php
class persona
{
        public $nombre;
        public $apellido;
        private $edad;

        // Método de modificacion de edad.
        function setEdad( $_edad ) {
                if ( $_edad > 0 && $_edad < 100) {
                        $this->edad = $_edad;
                }
        }
        // Método de obtencion de edad
        function getEdad() {
                return $this->edad;
        }
}
```

La función *setEdad()* recibe como parámetro el valor que se desea asignar al atributo $edad. Si el valor está comprendido entre 0 y 100 pasa al atributo, en caso contrario se ignora protegiendo el atributo.

También pueden crearse *funciones de acceso*. Estas son dos funciones con nombre ***__get()*** y ***__set()*** que deben declararse del siguiente modo:

```
        // Obtenciόm de atributos
        function __get($atr) {

        }
        // Modificacion de atributos
        function __set($atr, $valor) {

        }
```

- La función ***__get()*** devuelve el valor correspondiente al atributo indicado como argumento a su parámetro *$atr*.

- La función ***__set()*** asigna al atributo indicado como argumento para el parámetro *$atr*, el valor dado al parámetro *$valor*.

Sea la clase *persona* anterior, el siguiente código muestra la implementación de las funciones de acceso ***__get()***, ***__set()*** para dar acceso a sus atributos.

```php
<?php
class persona
{
        private $nombre;
        private $apellido;
        private $edad;

        // Obtenciόm de atributos
        function __get($atr) {
                return $this->$atr;
        }

        // Modificacion de atributos
        function __set($atr, $valor) {
                switch($atr) {
                        case "edad": {          // Atributo edad
                                if ( $valor > 0 && $valor < 100) {
                                        $this->edad = $valor;
                                }
                        }; break;
                        case "nombre": {         // Atributo nombre
                                $this->nombre = $valor;
                        }; break;
                        case "apellido": {      // Atributo apellido
                                $this->apellido = $valor;
                        }; break;
                }
        }
}
```

El acceso a los atributos a través de las funciones de acceso se realiza empleando la misma sintaxis que para acceder directamente a los atributos:

```php
<?php
        include_once "classes/cuenta.class.php";
        include_once "classes/persona.class.php";

        $obj = new persona("Roger", "Petrov", 25);
         $obj->nombre = "Yulius";
         $obj->apellido = "Galligan";
         $obj->edad = 23;
```

Al realizar una operación de asignación sobre el atributo "*edad*", se ejecuta la función de acceso *__set()* recibiendo como parámetro *$atr* el nombre del atributo solicitado "*edad*", y en el parámetro "valor" el valor que se le pretende asignar.

Al realizar una operación de obtención del valor del atributo "*edad*", se ejecuta la función de acceso *__get()* recibiendo parámetro *$atr* el nombre del atributo. La función devuelve entonces el valor del atributo correspondiente.

(*) Las funciones de acceso ***__get()*** y ***__set()*** pueden emplearse para dar acceso a atributos inaccesibles declarados privados o protegidos. Sin embargo, estas funciones no se ejecutan (incluso aunque estén declaradas) al acceder a atributos públicos.

Tipificación de parámetros de entrada en métodos

Las variables que representan objetos también pueden utilizarse como parámetros de funciones para otros objetos. Esta es la base de la interacción entre los objetos que permite a su vez relacionar los diferentes tipos de datos de una aplicación web.

La *tipificación* de parámetros *permite indicar la clase a la que debe pertenecer el objeto pasado como argumento para un determinado parámetro de una función*.

Supóngase las siguientes clases.

La clase *coche* define los datos que almacena un programa de cada coche gestionado:

```
class coche{
        private $matricula;
        private $marca;
        private $modelo;
        public function __construct( $ matricula, $ marca, $ modelo) {
                $this->matricula = $ matricula;
                $this->marca = $ marca;
                $this->modelo = $ modelo;
        }
}
```

La clase *empleado* representa a su vez los datos que se almacenan de cada empleado de la empresa, incluyendo los datos de su coche. Es por ello que la clase define un atributo *$coche* que hace referencia al objeto coche del empleado.

```
class empleado {
        public $dni;
        public $nombre;
        private $edad;
        private $coche;
        public function  __construct( $ dni, $ nombre, $ edad, Coche $ coche ){
                $this->dni = $ dni;
                $this->nombre = $ nombre;
                $this->setEdad( $ edad );
                $this->coche = $ coche;
        }
        public function getCoche() {
                return $this->coche;
        }
}
```

El constructor de la clase *empleado* recibe un parámetro *$_coche* tipificado que sólo admite como argumentos objetos de la clase *coche*.

El hecho de que la clase *empleado* tenga un atributo *$coche* con un objeto de la clase *coche* establece una asociación entre ambos tipos de datos, tal que; cada empleado se entiende está relacionado con uno o ningún coche.

Espacios de nombres

Una aplicación Web puede estar dividida en una gran cantidad de clases para representar tanto datos de la aplicación como componentes de la misma: páginas, controles, conexiones a base de datos…, etc.

Los espacios de nombres permiten organizar clases relacionadas del mismo modo que se emplean las carpetas para organizar los ficheros en Windows. Por ejemplo; una aplicación web provista de múltiples clases dedicadas a la conexión con base de datos (conexión, comando, lector…, etc), podría definirlas dentro de un espacio de nombres "basedatos" para separarlas del resto de clases.

Los espacios de nombres se definen mediante la palabra clave *namespace* seguida de un identificador. Todas las clases definidas a continuación son consideras pertenecientes a ese espacio de nombres.

Ejemplo: El siguiente script (*punto.class.php*) define una clase *punto* contenida en el espacio de nombre *geometría*:

```
namespace geometria;
class punto {
        private $x;
        private $y;
        public function __construct( $_x, $_y) {
                $this->x = $_x;
                $this->y = $_y;
        }
}
```

Una vez definida la clase como parte de un espacio de nombres, éste debe indicarse al crear un objeto de los siguientes modos:

Empleando el nombre completo de la clase, es decir; el espacio de nombres, contrabarra (****), y el nombre de la clase:

```
<?php
require "punto.class.php";

$obj = new geometria\punto(10,20);
```

Empleando la sentencia ***use*** indicando que queremos instanciar la clase *punto* perteneciente al espacio de nombres *geometría* (*geometría**punto*):

```
<?php
require "punto.class.php";

use geometria\punto;

$obj = new punto(10,20);
```

Los espacios de nombres pueden anidarse empleando la contrabarra (****) para crear sub-espacios y mejorar la organización de las clases.

Ejemplo: fichero *punto.class.php*

```php
<?php
namespace geometria;
class punto {
        private $x;
        private $y;
        function __construct( $_x, $_y) {
                $this->x = $_x;
                $this->y = $_y;
        }
}
```

Ejemplo: fichero *cuadrado.class.php*

```php
<?php
namespace geometria\figuras;

require_once "punto.class.php";

use geometria\punto;

class cuadrado{
        private $punto;
        private $lado;
        function __construct( punto $_punto, $_lado ) {
                $this->punto = $_punto;
                $this->lado = $_lado;
        }
}
```

Para crear un objeto cuadrado emplearíamos el siguiente código:

```php
<?php
require_once "punto.class.php";
require_once "cuadrado.class.php";

use geometria\punto;
use geometria\figuras\cuadrado;

// Creación de un objeto punto
$p = new punto(10,10);

// Creacion de un objeto cuadrado
$c = new cuadrado($p, 10);
```

Recomendaciones sobre organización de scripts de clases y espacios de nombres.

Se recomienda guardar las clases en scripts con el nombre de la clase y extensión ".class.php". Ej: Sea la clase ***Persona*** → El fichero debe llamarse: ***persona.class.php***

Si se usan espacios de nombres, los scripts con las clases deben organizarse en una estructura de carpetas que imite los espacios/subespacios de nombres indicados. Ej: Sea la clase ***punto*** en el espacio de nombres ***geometría*** → El fichero *punto.class.php* debería encontrarse almacenado dentro de una carpeta *geometría*.

Se recomienda almacenar todos los scripts con clases dentro de una carpeta "*clases*" para separarlos del resto de ficheros, scripts, páginas y recursos de la aplicación.

Autocarga de clases.

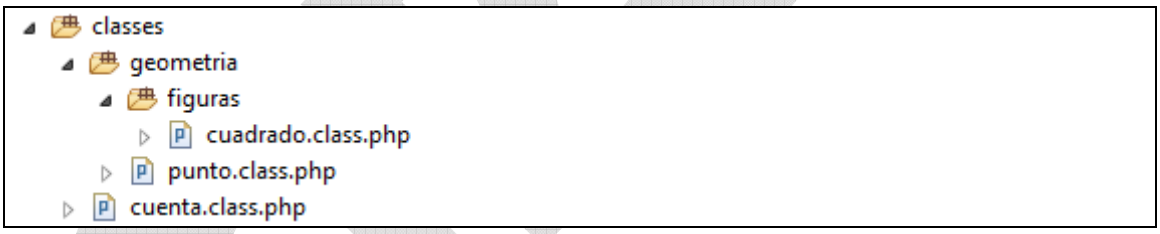
La declaración de clases y espacios de nombres en ficheros separados y carpetas permite organizar los scripts en el servidor. Sin embargo, para poder crear un objeto de cualquier clase es obligatorio importar primero el script con su declaración mediante las funciones ***require, include, require_once, include_once***….

Cuando se emplean gran número de clases puede ser complicado importar manualmente todos los scripts. Para automatizar este proceso pueden emplearse la función de PHP ***spl_autoload_register()***.

```
bool spl_autoload_register ([ callable $autoload_function
          [, bool $throw = true
          [, bool $prepend = false ]]] )
```

La función *spl_autoload_register()* registra *funciones de autocarga* que importan el script correspondiente a una clase/interfaz a partir de su nombre. Una *función de autocarga* recibe como único parámetro el nombre de la clase que se instancia, y su código debe importar el script correspondiente a partir del nombre de la clase.

Ejemplo: Supóngase una aplicación web provista de tres clases vistas anteriormente (*cuenta*, *punto* y *cuadrado*), almacenadas en scripts con la siguiente estructura:



El siguiente código registra una función de autocarga capaz de cargar cualquiera de las tres clases anteriores desde un script o página web *situada en la raíz del sitio web*.

```
spl_autoload_register(function($clase){
        $ruta = "classes/$clase.class.php";
        require $ruta;
});

use geometria\punto;
use geometria\figuras\cuadrado;

// Creación de una cuenta
$obj = new cuenta("pepe", 1000.45, 0.56);
// Creación de un objeto punto
$p = new punto(10,10);
// Creacion de un objeto cuadrado
$c = new cuadrado($p, 10);
```

Para que la función de autocarga funcione correctamente al margen de la ubicación del script deberíamos modificar la función de manera que obtenga la ruta al script con la clase a partir de la raíz del sitio web:

```
spl_autoload_register(function($clase){
        $ruta = $_SERVER['DOCUMENT_ROOT']."/classes/$clase.class.php";
        require $ruta;
});
```

Prácticas

1. Crea una clase llamada *persona* que conste de un atributo privado $*nombre*, un constructor que reciba el nombre de la persona como parámetro de entrada, y un método *Saludar*() que muestre por pantalla el mensaje: "*Hola <nombre>*" al ser invocado.

2. Modifica la clase *persona* del ejercicio anterior añadiendo los métodos que sean necesarios para poder obtener y modificar el valor del atributo $nombre de un objeto Persona una vez creado.

3. Modifica la clase *persona* añadiendo un segundo atributo *$edad*. El valor del atributo debe recibirse como parámetro en el constructor de la clase. Añade los métodos necesarios para obtener la edad e incrementarla en la unidad cuando la persona cumpla años.

4. Crea una script PHP (*automovil.class.php*) que defina una clase *automovil*. Esta clase debe poseer los siguientes atributos: *$matricula, $marca, $modelo* y *$combustible*. La clase debe incluir los siguientes métodos: Un constructor que reciba como parámetros los valores correspondientes para todos los atributos, y un método *mostrar()* que muestre una fila de una tabla HTML con una columna por cada atributo indicando el nombre y su correspondiente valor:

Matricula	Marca	Modelo	Combustible
4383-SA	Seat	Toledo	24.56

Para probar la clase crea una página PHP que defina una matriz de objetos *vehículo* con los siguientes atributos:

Matricula	Marca	Modelo	Combustible
4383-SA	Seat	Toledo	24.56
3949-SS	Citron	Saxo	20.345
7644-GH	Peugeot	307	14.56
4955-FF	Fiat	Punto	37.54

A continuación crea una función *mostrarVehiculos()* que reciba como parámetro la matriz anterior y muestre una tabla HTML con los datos de todos los vehículos

Herencia

La herencia es un mecanismo de la programación orientada a objetos que permite la definición de una clase (*subclase*) a partir de otra ya existente (*clase base*). La subclase representa un subtipo de la clase base.

Herencia de atributos y métodos

Supongamos una aplicación Web encargada de gestionar la compra y recogida de teléfonos móviles nuevos y averiados. En principio podrían crearse dos clases para representar tanto a los móviles en venta, como los averiados:

```php
class movil_venta {
        public $marca;
        public $modelo;
        public $precio;
        public $fecha_ingreso;
}

class movil_averiado {
        public $marca;
        public $modelo;
        public $dueño;
        public $motivo;
}
```

Ambas clases comparten atributos como *$marca* y *$modelo*, es decir; todos los móviles tienen una marca y modelo. Aplicando la herencia podría crearse una clase *movil* con los atributos *$marca* y *$modelo*, y las clases *móvil_venta* y *móvil_averiado* pueden declararse como subclases de *móvil*.

```php
// Clase base
class movil
{
        public $marca;
        public $modelo;
}
```

```php
// Subclase 1
require_once "movil.class.php";
class movil_venta extends movil {
        public $precio;
        public $fechaVenta;
}
```

```php
// Subclase 2
require_once "movil.class.php";
class movil_averiado extends movil {
        public $dueño;
        public $motivo;
}
```

La herencia se implementa en PHP al definir las clases relacionada. Para que indicar que una clase hereda de otra debe indicarse la palabra clave ***extends*** en su definición seguida del nombre de la clase base.

Las clases *móvil_venta y movel_averiado* contienen los atributos *$marca* y *$modelo* heredados de la clase base móvil, a parte de los suyos propios.

Ejemplo: El siguiente código muestra la instanciación de objetos de las clases *móvil_venta* y *móvil_averiado* inicializando tanto los atributos heredados de la clase base *móvil*, como los específicos de cada tipo:

```php
$obj = new movil_venta();
$obj->marca = "NOKIA";
$obj->modelo = "LUMIA";
$obj->fechaVenta = "20/08/2015";
$obj->precio = 220.45;
var_dump($obj);

$obj = new movil_averiado();
$obj->marca = "NOKIA";
$obj->modelo = "LUMIA";
$obj->dueño = "Roger Petrov";
$obj->motivo = "Pantalla negra";
var_dump($obj);

$obj = new movil();
$obj->marca = "NOKIA";
$obj->modelo = "LUMIA";
var_dump($obj);
```

```
C:\xampp7\htdocs\prueba\home.php:53:
object(movil_venta)[1]
  public 'precio' => float 220.45
  public 'fechaVenta' => string '20/08/2015' (length=10)
  public 'marca' => string 'NOKIA' (length=5)
  public 'modelo' => string 'LUMIA' (length=5)

C:\xampp7\htdocs\prueba\home.php:60:
object(movil_averiado)[2]
  public 'dueño' => string 'Roger Petrov' (length=12)
  public 'motivo' => string 'Pantalla negra' (length=14)
  public 'marca' => string 'NOKIA' (length=5)
  public 'modelo' => string 'LUMIA' (length=5)

C:\xampp7\htdocs\prueba\home.php:65:
object(movil)[1]
  public 'marca' => string 'NOKIA' (length=5)
  public 'modelo' => string 'LUMIA' (length=5)
```

(*) La herencia es una relación unidireccional. Las subclases heredan los atributos de la clase base, pero la clase base no hereda los de las subclases.

Los métodos de las clases bases también son heredados por las subclases, de modo que éstos pueden ser llamados contra objetos de las subclases como si fueran suyos.

Ejemplo: El siguiente código muestra la clase *cuenta* y la subclase *cuenta_bonificada* que añade únicamente un atributo *$bonificación*.

```php
class cuenta {
        public $titular;
        private $saldo = 0;

        public function ingresar( $_cantidad) {
                $this->saldo += $_cantidad;
        }
        public function retirar( $_cantidad) {
                $this->saldo -= $_cantidad;
        }
        public function obtenerSaldo() {
                return $this->saldo;
        }
}
```

```php
class cuenta_bonificada extends cuenta {
        public $bonificacion;
}
```

Ejemplo: El siguiente código muestra la instanciación de objetos de la clase *cuenta* y *cuenta_bonificada* y el uso de los métodos heredados *ingresar()*, *retirar()* y *obtenerSaldo()* en ambos:

```php
// Instanciacion de la clase cuenta
$objCuenta = new cuenta();
$objCuenta->titular = "Roger Petrovich";
$objCuenta->ingresar(1000.0);
$objCuenta->retirar(500.0);
echo $objCuenta->obtenerSaldo()."<br />";

// Instanciacion de la subclase cuenta_bonificada
$objCBonif = new cuenta_bonificada();
$objCBonif->titular = "Ivan Mendelevief";
$objCBonif->ingresar(2000.0);
$objCBonif->retirar(250.0);
echo $objCBonif->obtenerSaldo()."<br />";
```

```
500
1750
```

Sobrescritura de métodos

A veces los métodos heredados por una subclase de su clase base no sirven ya que no tienen en cuenta los atributos adicionales de la subclase.

Ejemplo 1: Supóngase la clase *cuenta* que incluye el método *calcularBeneficio()* heredado por la clase subclase *cuenta_bonificada*:

```php
class cuenta {
        public $saldo;
        public $interes;

        public function calcularBeneficio() {
                return $this->saldo * ($this->interes/100.0);
        }
}

class cuenta_bonificada extends cuenta {
        public $bonificacion;
}
```

El método *calcularBeneficio()* es correcto en la clase *cuenta*, pero no tiene en cuenta el atributo bonificación en la subclase *cuenta_bonificada*. Para resolverlo puede redefinirse el método heredado para adaptarlo a la subclase. Esta operación recibe el nombre de *sobrescritura* o *redefinición*.

Ejemplo El siguiente código muestra el beneficio de una cuenta bonificada de 1200€ de saldo con un interés del 10% y 100€ de bonificación. El método *calcularBonificacion()* devuelve sin embargo el valor de 120€ sin tener en cuenta la bonificación.

```php
// Instanciacion de la subclase cuenta_bonificada
$objCBonif = new cuenta_bonificada();
$objCBonif->saldo = 1200;
$objCBonif->interes = 10;
$objCBonif->bonificacion = 100;
echo "El beneficio es de: ".$objCBonif->calcularBeneficio()."<br />";
```

La sobrescritura puede ser parcial, o completa.

- Se dice que la sobrescritura es parcial cuando se sobrescribe un método con el fin de ampliar su código reutilizando la versión heredada del método. *Para llamar la versión original del método debe emplearse la referencia **parent** seguida del operador "::", y el método correspondiente*

- Se dice que la sobrescritura es completa cuando se sobrescribe el método sin reutilizar la versión heredada del método

Ejemplo: El siguiente código muestra la sobrescritura parcial del método *calcularBeneficio()* heredado de la clase cuenta. Para que el nuevo método sea correcto se invoca a la versión original heredada de la clase base y se le suma el valor del atributo *$bonificiacion*.

```
class cuenta_bonificada extends cuenta {
        public $bonificacion;

         public function calcularBeneficio() {
                return parent::calcularBeneficio() + $this->bonificacion;
         }
}
// Instanciacion de la subclase cuenta_bonificada
$objCBonif = new cuenta_bonificada();
$objCBonif->saldo = 1200;
$objCBonif->interes = 10;
$objCBonif->bonificacion = 100;
echo "El beneficio es de: ".$objCBonif->calcularBeneficio()."<br />";
```

Ahora el resultado devuelto por el método calcularBeneficio() del objeto *$objCBonif* es 220€ (1200 x 10% + 100) teniendo en cuenta la cantidad de *$bonificacion*.

Constructores en herencia

Los métodos constructores también actúan en la herencia. Sea la clase *punto:*

```
class punto {                    // Clase padre con Constructor
        private $x;
        private $y;
        function  __construct($_x, $_y) {
                $this->x = $_x;
                $this->y = $_y;
        }
}
```

Supongamos una subclase *punto_coloreado* que hereda de *punto*. Esta clase hereda los atributos *$x*, e *$y*, y añade un atributo *$color*. El constructor de la subclase debe recibir tantos parámetros como atributos posee (tanto propios como heredados):

```
class punto_coloreado extends Punto{     // Clase hija
        private $color;
         function __construct($_x, $_y, $_color) {

         }
}
```

El constructor de la subclase *debe llamar al constructor de la clase base pasando los parámetros correspondientes a sus atributos para que se inicialicen en ella*. Los atributos propios se inicializan normalmente.

Para invocar al constructor de la clase base desde el constructor de la subclase debe emplearse la referencia predefinida ***parent*** seguida del operador (***::***), de igual modo que en una sobrescritura parcial.

```
class punto_coloreado extends Punto{     // Clase hija
        private $color;
         function __construct($_x, $_y, $_color) {
                parent::__construct($_x, $_y);            // LLamada a constructor padre
                $this->color = $_color;                   // Inicializacion atributo propio
         }
}
```

Ejemplo 2: Los siguientes códigos muestran la declaración completa de las clases *punto* y *punto_coloreado* con sus respectivos constructores y el método *getDatos()* que devuelve una cadena con los datos de cada objeto sobrescrito:

```
class punto {
        private $x;
        private $y;
        function  __construct($_x, $_y) {
                $this->x = $_x;
                $this->y = $_y;
        }
        function getDatos() {
                return "X: ".$this->x." Y:".$this->y;
        }
}
```

```
class punto_coloreado extends Punto{
        private $color;
         // Constructor subclase
         function __construct($_x, $_y, $_color) {
                parent::__construct($_x, $_y);
                $this->color = $_color;
         }

         // Sobrescritura parcial método getDatos()
         function getDatos() {
                return parent::getDatos() . " Color:".$this->color;
         }
}
```

El siguiente código instancia y muestra los datos de dos objetos de ambas clases:

```
// Instanciacion y visualizacion de objeto punto
$obj = new punto(2,3);
echo $obj->getDatos()."<br/>";

// Instanciacion y visualizaicon de objeto punto_coloreado
$objc = new punto_coloreado(2,3,"rojo");
echo $objc->getDatos()."<br/>";
```

```
X: 2 Y:3
X: 2 Y:3 Color:rojo
```

Encapsulación en herencia

Los modificadores privado (*prívate*) y protegido (*protected*) pueden emplearse para controlar qué atributos y métodos de una clase base son accesibles en las subclases.

Ejemplo: Supóngase las siguientes clase *base* y la clase *sub* que hereda de ésta.

```php
class base {
        public function fooA() {
                echo "funcion publica llamada<br/>";
        }
        private function fooB() {
                echo "funcion privada llamada<br/>";
        }
        protected function fooC() {
                echo "función protegida llamada<br/>";
        }
}
```

```php
class sub extends base {
        function __construct() {
                $this->fooA();          // Llamada a función público
                $this->fooC();          // Llamada a función protegido
                $this->fooB();          // Llamada a funcion privada
        }
}
```

Al crear un objeto de la clase *sub*, se ejecuta su constructor el cual intenta llamar a los tres métodos de su clase base. El resultado mostrado sería el siguiente:

```php
$obj = new sub();
```

funcion publica llamada
función protegida llamada

Fatal error: Uncaught Error: Call to private method base::fooB() from context 'sub' in C:\xampp7\htdocs\prueba\home.php on line *47*

Error: Call to private method base::fooB() from context 'sub' in C:\xampp7\htdocs\prueba\home.php on line *47*

Call Stack

#	Time	Memory	Function	Location
1	0.0156	357552	{main}( )	...\home.php:0
2	0.0156	357576	sub->__construct( )	...\home.php:52

La llamada a los método *fooA()* y *fooB()* desde el constructor de la subclase son posibles y se ejecutan normalmente. Sin embargo, *la llamada al método fooC() no es posible dado que está declarado como privado en la clase base*.

Por otro lado, si creamos un objeto de la clase *base* e invocamos sus métodos *sólo fooA() resulta accesible al ser el único declarado público*:

```php
$obj = new base();
$obj->fooA();           // FUNCIONA OK
$obj->fooC();           // NO FUNCIONA ( Método privado – inaccesible )
$obj->fooB();         // NO FUNCIONA ( Método protegido – inaccesible )
```

Resumen:

El modificador de visibilidad protegido (*protected*); define atributos y métodos no accesibles desde fuera de la clase (tal y como si fuesen privados); pero si desde las subclases. En comparación; el modificador *private no permite el acceso desde las subclases*:

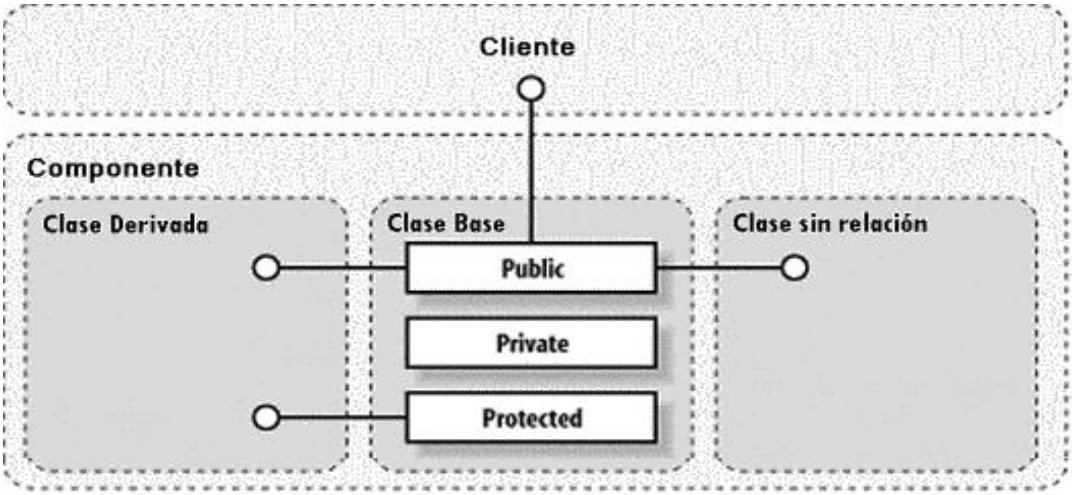


Diagrama de accesibilidad de los modificadores de visibilidad.

El modificador de visibilidad protegido se emplea normalmente con atributos y métodos que se desea que sean accesibles desde los métodos de las subclases.

imitar la herencia

Impedir la sobrescritura de métodos

Para impedirse que un método pueda sobrescribirse en una subclase añadiendo la palabra clave ***final*** en su declaración:

```php
class cuenta {
        public $saldo;
        public $interes;

        public final function calcularBeneficio() {
                return $this->saldo * ($this->interes/100.0);
        }
}
```

Con esto el método *calcularBeneficio()* no puede ser sobrescrito en las subclases. En caso de intentarlo se genera un error:

```
Fatal error: Cannot override final method cuenta::calcularBeneficio() in C:\xampp7\htdocs\prueba\home.php on line 42
```

Impedir la sobrescritura se realiza en métodos que llevan a cabo operaciones muy delicadas, *por lo que no se desea que puedan modificarse en las subclases.*

Impedir la herencia

Para impedir que una clase sirva clase base para la definición de otras subclases basta con añadir la palabra clave ***final*** delante de su declaración:

```
final class punto_coloreado extends Punto{
        private $color;

}
```

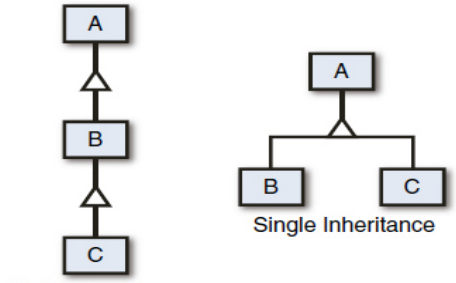
Esto impide que puedan crearse otras clases a partir de ésta. A estas clases que no admiten la herencia se les denomina ***clases finales***.

Herencia simple

En PHP sólo está permitida la *herencia simple*:

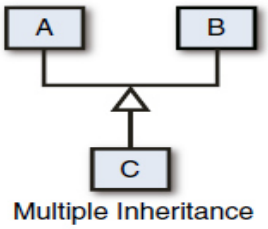
- Una clase puede ser base para infinitas subclases.
- Una subclase solo puede heredar *directamente* de una única clase base.

La herencia es *transitiva*. Esto significa que una clase C puede heredar de B y ésta a la vez de A. En tal caso, la subclase C hereda los atributos y métodos tanto de B directamente, como de A indirectamente.



Izquierda) Esquema de una jerarquía de herencia simple donde C hereda de B y A indirectamente.
Derecha) Esquema de herencia simple donde A tiene B y C como subclases.
B y C sólo pueden heredar directamente de A.

Existe otro tipo de herencia denominada *herencia múltiple* en la que las subclases pueden heredar de varias clases base directamente.



Representación de herencia múltiple, donde la subclase C hereda directamente de las clases base A y B

Este tipo de herencia ***no está permitida en PHP***.

Interfaces

Una *clase* describe las *características* (atributos) y *operaciones* (métodos) que representan un *determinado tipo de dato* (persona, coche, alumno, articulo…., etc). Un *interfaz* sin embargo define *métodos* que describen una determinada *funcionalidad*.

Una interfaz se define mediante la palabra clave ***interface***. Su declaración consta únicamente de métodos pero sin incluir código (sólo declaración, no implementación).

Ejemplo: El código siguiente muestra la declaración del interfaz *Visualizable* representa la capacidad de obtener el código HTML para representar algo. (*No define qué algo, sólo la funcionalidad; por esto tiene como identificador un adverbio y no un sustantivo*):

```
interface Visualizable {
	function getHTML();
}
```

Los interfaces son *implementados* por las clases. *Una clase implementa un interfaz al declarar e implementar todos los métodos del interfaz como suyos*. Con ello, la clase obtiene la funcionalidad del interfaz.

Para indicar que una clase implementa una determinada interfaz debe indicarse la palabra clave ***implements*** seguida del nombre del interfaz. *La clase debe definir con código todos los métodos declarados en la interfaz*:

Ejempo: La clase *punto* implementa la interfaz *Visualizable* dado que define e implementa su método *getHTML()*.

```
class punto implements Visualizable{
	private $x;
	private $y;

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

	 public function getHTML()
	 {
echo <<<END
			<table>
			<thead><td>X</td><td>Y</td></thead>
			<tr><td>$this->x</td><td>$this->y</td></tr>
			</table>
END;
	}
}
```

Mediante el uso del interfaz *Visualizable* se establece que los objetos de las clases que lo implementan pueden mostrarse como parte de una página web llamando al método *getHTML()* declarado por el interfaz.

Los métodos implementador del interfaz se llaman del mismo modo que el resto.

```
$p = new punto(2,3);
$p->getHTML();
```

```
X Y
2 3
```

Una clase puede implementar tantos interfaces como se desee. Para ello debe indicarse separados por comas los nombres de los interfaces tras la partícula ***implements***, e implementar todos los métodos declarados.

Ejemplo: Supóngase los interfaces *Visualizable* y *Comparable*:

```
interface Visualizable {
        function getHTML();
}

interface Comparable {
        function getValor();
}
```

Ambos interfaces pueden ser implementados al mismo tiempo por la clase tiempo:

```
class tiempo implements Visualizable, Comparable{
        private $h;
        private $m;
        private $s;

        function  __construct($_h, $_m, $_s) {
                $this->h = $_h;
                $this->m = $_m;
                $this->s = $_s;
        }

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

         public function getValor() {
                 return $this->h * 3600 + ( $this->m * 60 ) + $this->s;
         }
}
```

Con ello, los objetos de la clase tiempo son:

- Visualizables → Pueden visualizarse como parte de una página web llamando al método *getHTML()* definido en el interfaz ***Visualizable***.

- Comparables → Pueden compararse unos con otros mediante el valor devuelto por el método *getValor()* definido en la interfaz ***Comparable***.

Traits

Un ***Trait*** define métodos *ya implementados* que pueden agregarse a las clases para su reutilización. En cierto modo son semejantes a los interfaces, pero con la diferencia de que los métodos ya están implementados y se reutilizan en las clases tal cual están declarados, al contrario que en los interfaces; donde cada clase los debe implementar según sea necesario.

Un trait se define como un clases o interfaz pero indicando la palabra clave ***trait***:

Ejemplo: El siguiente código implementa un trait llamado ***Logger*** que define un único método ***log()*** que registra en un fichero de registro "*error.log*" el mensaje indicado como parámetro *$message*:

```
trait Logger {
        private function log($message) {
                error_log( date("D, d M Y H:i:s")."\t$message\r\n", 3, "errors.log");
        }
}
```

Los métodos definidos en un trait puede añadirse a una clase indicando en la declaración la palabra clave ***use*** seguida del identificador del trait. Con ello, los métodos del trait pasan a formar parte de la clase y sus objetos.

```
class tiempo implements Visualizable, Comparable{
        private $h;
        private $m;
        private $s;

        use Logger;                        // Agregado de métodos de Logger a la clase.

        function  __construct($_h, $_m, $_s) {
                $this->h = $_h;
                $this->m = $_m;
                $this->s = $_s;
                $this->log("Objeto tiempo creado.");   // Llamada al método del Trait
        }
}
```

Los métodos obtenidos del trait se invocan como si fueran de la propia clase según sus modificadores de visibilidad. En el ejemplo, el método *log()* es invocado desde el interior de la clase *tiempo* como un método privado más para registrar la instanciación de cada objeto.

Una clase puede agregar tantos Traits como sean necesarios indicando la palabra clave use seguida de sus identificadores separados por comas:

```
        use TraitA, TraitB;          // Agregado de métodos de TraitA y TraitB a la clase.
```

Ejercicio

Se pide crear una aplicación web basada en un único formulario que permita calcular el coste de una pizza en función del tipo de pizza, ingredientes y ofertas seleccionadas.

Para hacer los cálculos se suministran los siguientes datos:

Tipos de Pizzas:

Tipos	Pequeña	Mediana	Familiar
Napolitana	4.00€	4.50€	5.20€
New York	4.20€	4.80€	5.50€
Pizza a Taglio	5.20€	5.80€	6.50€
Argentina	6.00€	6.80€	7.20€
Chicago	4.20€	4.50€	5.00€
Sfincione	5.80€	6.20€	6.50€

Los tipos de pizzas son representados por la clase ***pizza*** que debe constar de los siguientes atributos: nombre, precio pequeña, precio mediana, y precio familiar. La clase debe definirse en el fichero *clases/pizza.class.php*.

Todos los precios deben estar contenidos en una matriz escalar de objetos pizza donde la clave sea el propio nombre de la pizza. La matriz debe llamarse ***$_pizzas*** y declararse en el fichero *datos/pizzas.dat.php*.

Ingredientes

Ingrediente	Sencillo	Doble
Anchoas	0.25€	0.3€
Atún	0.25€	0.35€
Bacon	0.2€	0.4€
Carne Vacuno	0.2€	0.45€
Cebolla	0.1€	0.15€
Cerdo	0.4€	0.65€
Champiñón	0.2€	0.4€
Gambas	0.3€	0.45€
Pepperoni	0.15€	0.3€
Pimientos	0.2€	0.25€
Jalapeños	0.15€	0.25€
Pollo marinado	0.35€	0.45€
Queso cheddar	0.25€	0.5€
Queso provolone	0.3€	0.45€
Queso Suizo	0.4€	0.65€
Salchicha	0.25€	0.5€
Tomate natural	0.15€	0.25€
Jamón york	0.15€	0.25€

Los ingredientes son representados por la clase ***ingrediente*** que debe constar de los siguientes atributos: nombre, precio sencillo, precio doble. La clase debe declararse en el fichero *clases/ingrediente.class.php*.

El listado de todos los ingredientes debe definirse en una matriz escalar de objetos ingrediente donde se almacene como clave el nombre del ingrediente. Esta matriz debe llamarse ***$_ingredientes*** y declararse en el fichero *datos/ingredientes.data.php*.

Las ofertas deben definirse como clases que implementen el interfaz ***calculable***. Este interfaz define un único método *computarPrecio()* que recibe los siguientes parámetros:

- *Precio pizza* →Indica el precio bruto de la pizza, es decir; el sumatorio del precio base de la pizza y el de los diferentes ingredientes añadidos.
- *Tamaño de pizza* → Indica el tipo de pizza adquirida: (pequeña/mediana/familiar).
- *Nº de pizzas* → Indica el nº de pizzas adquiridas.
- *VISA* → Indica si el usuario efectúa el pago con VISA o no.

Deben definirse 4 tipos de ofertas implementadas como 4 clases:

- ***ofertaWeekend*** → aplica un 10% de descuento si es fin de semana + 5% adicional si se paga con VISA.
- ***ofertaFamily*** → aplica un 20% de descuento para pizzas grandes a partir de 2 unidades. Se computa un 4$ de descuento adicional por cada unidad extra. Se añade un 2% adicional si se paga con VISA.
- ***ofertaFriends*** → aplica un 2% de descuento por cada pizza mediana o pequeña únicamente durante los fines de semana.

Estas clases y el interfaz debe deben definirse en el fichero *clases/ofertas.class.php*.

Finalmente definir una clase ***compra*** encargada de calcular el importe final de la compra en función de la pizza, tamaño, unidades, ingredientes y oferta seleccionada. Esta clase debe definir los siguientes miembros:

- Un constructor que reciba como parámetros los siguientes datos:
  - Pizza → Objeto de la clase pizza que representa la pizza seleccionada.
  - Tamaño → Cadena que indica el tamaño de la pizza: “pequeña”, “mediana”, “familiar”.
  - Cantidad → Valor que indica la cantidad de pizzas adquiridas
  - VISA → Valor lógico que indica si el usuario paga con VISA.

- Un método *añadirIngrediente()* agrega un ingrediente a la compra. Este método puede llamarse varias veces para añadir varios ingredientes a la compra. Debe recibir los siguientes parámetros:
  - Ingrediente → Objeto de la clase ingrediente que indica el ingrediente seleccionado.
  - Doble → Valor lógico que indica si se añade doble cantidad (valor lógico cierto), o simple (valor lógico falso) del ingrediente.

- Un método *añadirOferta()* agrega una oferta a la compra. Debe recibir como parámetro un objeto de cualquiera de las clases que implementan el interfaz *calculable*. Si se invoca este método varias veces sólo se almacena la última oferta añadida, es decir; las ofertas no son acumulables.
- Un método *calcularPrecio()* que calcula y retorna el importe final de la compra.

La clase compra debe definirse en el fichero *clases/compra.class.php*.

La aplicación debe constar de dos páginas.

- ***formulario.php*** → Se trata de un formulario generado mediante PHP que debe mostrar los siguientes elementos:

  - Una lista desplegable con el nombre de cada tipo de pizza definido en la matriz *$_pizzas*.
  - Un conjunto de tres botones de opción que permitan elegir uno de los tres tamaños posibles: "*pequeño*", "*mediano*", "*familiar*".
  - Una caja de texto que permite indicar el nº de pizzas. Debe contener un valor numérico válido comprendido entre 1 y 9 incluidos.
  - Por cada ingrediente en la matriz $_ingredientes debe mostrarse:
  - El nombre del ingrediente seguido de dos botones de opción que permitan seleccionar entre cantidad simple y doble.
  - Una casilla de verificación que permita indicar si el usuario pagará con VISA.
  - Un conjunto de tres botones de opción que permitan indicar la oferta a aplicar a la compra: *weekend*, *family* o *friends*.

Los datos deben enviarse mediante POST a la siguiente página *compra.php*.

- ***compra.php*** → Esta página muestra los datos del pedido realizado:

  - El tipo de pizza, tamaño y nº de unidades pedidas.
  - Una tabla con los ingredientes añadidos y su cantidad
  - El importa bruto del pedido sin oferta alguna. (el sumatorio del precio de la pizza y los ingredientes añadidos únicamente).
  - El nombre de la oferta seleccionada
  - El importa final del pedido una vez aplicada la oferta seleccionada.

Práctica

Crea una interfaz llamada *htmlrenderable* que conste únicamente de una función llamada *html()*:

```
interface htmlrenderable {
	function html();
}
```

Este interfaz lo vamos a emplear para identificar a aquellas clases que pueden representar sus objetos en HTML llamando a la función *html()*.

1.- Crea una clase ***textbox*** que representa una caja de texto de un formulario. La clase debe poseer un constructor con los siguientes atributos:

- *id* → Cadena identificadora de la caja de texto. (su valor debe aplicarse al atributo *name* de la etiqueta *<input>*).
- *valor* → Cadena que representa el valor mostrado por la caja de texto

La clase debe implementar el interfaz *htmlrenderable*. El método *html()* debe generar el código HTML que represente la caja de texto.

Solución:

```
class textbox implements htmlrenderable {
	protected $id;
	protected $valor;
	public function __construct($_id, $_valor = null) {
		$this->id = $_id;
		$this->valor = $_valor;
	}

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

Prueba:

```
<?php
	$txtArgumento = new textbox("argumento");
?>
<!DOCTYPE html>
<html>
	<head>
		<meta charset="ISO-8859-1">
		<title>Insert title here</title>
	</head>
	<body>
	<form>
		Nombre<?php $txtArgumento->html(); ?>
		<input type="submit" value="procesar">
	</form>
	</body>
</html>
```

Nombre [ ] procesar

Modifica a continuación la clase *textbox* de modo que pueda recuperarse su valor al pulsar el botón "*procesar*" empleando la matriz superglobal *$_REQUEST*.

```
class textbox implements htmlrenderable {
        protected $id;
        protected $valor;
        public function __construct($_id, $_valor = null) {
                $this->id = $_id;
                if ( $_valor == null ) {
                        // Recupera valor de matriz superglobal $_REQUEST
                        if ( isset ($_REQUEST[$this->id])) {
                                $this->valor = $_REQUEST[$this->id];
                        }
                } else {
                        $this->valor = $_valor;
                }
        }

        public function getValor() {
                return $this->valor;
        }

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

2.- Crea una clase ***listbox*** que representa una lista desplegable implementando la interfaz *htmlrenderable*. La clase debe constar de los siguientes miembros:

```
class listbox implements htmlrenderable {
        protected $id;
        protected $opciones = array();

        public function __construct($_id )

        public function opcion( listboxopcion $opcion )

        public function getValor()

        public function html()}
```

El método *opcion()* permite agregar opciones a la lista. Estas opciones son representadas por objetos de la clase ***listboxopcion***, la cual también implementa la interfaz *htmlrenderable* con los siguientes miembros:

```
class listboxopcion implements htmlrenderable {
        protected $titulo;
        protected $valor;
        protected $seleccionado;

        public function __construct( $_titulo, $_valor, $_seleccionado = NULL)

        public function estaSeleccionado( $_seleccionado)

        public function html()
}
```

El método *html()* de la clase *listboxopcion* genera un elemento HTML *<option>* cuyo atributo *value* debe corresponderse con el de la propiedad *$valor*, y el contenido el de la propiedad $*titulo*. El elemento debe contener el atributo *selected* si la propiedad *$seleccionado* es true.

El código de la clase *listboxoption* sería el siguiente:

```
class listboxopcion implements htmlrenderable {
        protected $titulo;
        protected $valor;
        protected $seleccionado;

        public function __construct( $_titulo, $_valor, $_seleccionado = NULL){
                $this->titulo = $_titulo;
                $this->valor = $_valor;
                $this->seleccionado = $_seleccionado;
        }

        public function estaSeleccionado( $_seleccionado) {
                $this->seleccionado = $_seleccionado;
        }

        public function html() {
                echo "<option value='$this->valor' ".
                     (($this->seleccionado)?"selected":"").">$this->titulo</option>";
        }
}
```

El método *html()* de la clase *listbox* debe mostrar un elemento HTML *<select>* cuyo atributo *name* se corresponde con el valor de la propiedad $id, y que incluye tantos elementos *<option>* como objetos *listboxoption* agregados.

```
class listbox implements htmlrenderable {

        protected $id;
        protected $opciones = array();

        public function __construct($_id ) {
                $this->id = $_id;
        }

        public function opcion( listboxopcion $opcion ) {
                array_push($this->opciones, $opcion );
                return $this;
        }

        public function getValor() {
                return ( isset($_REQUEST[$this->id])?$_REQUEST[$this->id]:NULL);
        }

        public function html() {
                echo "<select name='$this->id'>";
                foreach ( $this->opciones as $opcion ) {
                        if ( isset( $_REQUEST[$this->id])) {
                                $opcion->estaSeleccionado
                                        (( $_REQUEST[$this->id] == $opcion->valor));
                        }
                        $opcion->html();
                }
                echo "</select>";
        }
}
```

Para recuperar el estado de selección previo de cada opción de la lista se comprueba el valor de la matriz superglobal $_REQUEST asociado a la propiedad $id, y se marca como seleccionada aquella opción cuyo valor coincide.

Prácticas

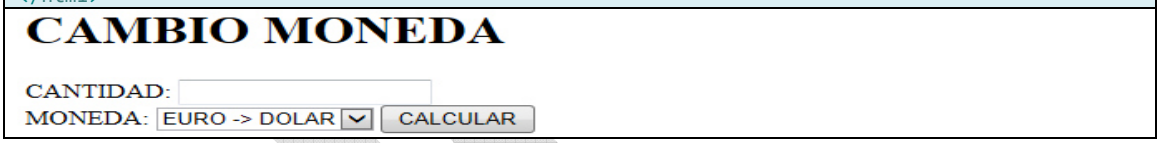
1.- Crea un formulario web que permita calcular el cambio de moneda de euros a dólares y viceversa. Para ello debes emplear las clases definidas anteriormente contenidas en un fichero *controles.class.php* que debe almacenarse en la carpeta *clases* del proyecto:

```php
<?php
require_once 'classes/controls.class.php';
$txt_cifra = new textbox("txt_cifra");
$lst_moneda = new listbox("lst_moneda");
$lst_moneda->opcion(new listboxopcion("EURO -> DOLAR", 1));
$lst_moneda->opcion(new listboxopcion("DOLAR -> EURO", 2));

EL CODIGO DE VALIDACION & CONVERSION DE VALORES DEBERÍA IR AQUI

?>
```

```html
<!DOCTYPE html>
<html>
<head>
<title>Selecciona una imagen para subirla al servidor</title>
</head>
<body>
<form>
        <h1>CAMBIO MONEDA</h1>
        CANTIDAD: <?php $txt_cifra->html(); ?><br/>
        MONEDA: <?php $lst_moneda->html(); ?>
        <input type="submit" value="CALCULAR">
</form>
</body>
</html>
```



- Para obtener tanto el valor introducido en la caja de texto representada por el objeto ***textbox***, como la opción seleccionada en la lista representada por el objeto ***listbox***, debe emplearse en ambos casos el método *getValor()*.

1.B.- Crea una clase ***validatebox*** que extienda de ***textbox*** e implemente una caja de texto con validación integrada. La clase debe tener los siguientes miembros:

```php
class validatebox extends textbox {

        protected $error_longitud = false;
        protected $error_numerico = false;
        protected $longitud;

        public function __construct($_id, $_valor = null, $_longitud = 9999)
        public function esValido()
        }
```

La función *esValido()* devuelve un valor lógico indicando si el contenido de la caja de texto es válido o no. Si el valor indicado no es un número debe mostrarse al margen derecho de la caja de texto un mensaje "*VALOR NO VALIDO*". Si el contenido excede la longitud debe mostrarse en su lugar el mensaje "*LONGITUD EXCEDIDA*". Deberá redefinirse el método *html()* del interfaz *htmlrenderable* para ello.

2.- Crea una clase *dni* que permita comprobar si un valor de DNI es válido. Los miembros deben ser los siguientes:

```
class dni {

        public function __construct($_dni)

        public function getNumero()

        public function getLetra()

        public function validar()
}
```

El DNI se recibe en el constructor como una cadena de la que debe separarse número y letra en los atributos correspondientes (*$numero* y *$letra*). Los métodos *getNumero()* y *getLetra()* devuelven el valor de los atributos correspondientes.

El método *validar()* devuelve un valor lógico cierto si la cadena dada como argumento para el constructor consta de 9 caracteres, los 8 primeros son cifras, el último una letra, y además la letra y el número coinciden.

(*) La letra de un DNI se obtiene a partir del resto del número del DNI entre 23 según las siguientes tablas:

RESTO	0	1	2	3	4	5	6	7	8	9	10	11
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B

RESTO	12	13	14	15	16	17	18	19	20	21	22
LETRA	N	J	Z	S	Q	V	H	L	C	K	E


CIPSA