U06. POO en PHP - Patrón MVC

1.- Características de la POO

La programación orientada a objetos (POO, o OOP en lenguaje inglés), es una metodología de programación basada en objetos. Un objeto es una estructura que contiene datos y el código que los maneja.

La estructura de los objetos se define en las clases. En ellas se escribe el código que define el comportamiento de los objetos y se indican los miembros que formarán parte de los objetos de dicha clase. Entre los miembros de una clase puede haber:

- **Métodos**. Son los miembros de la clase que contienen el código de la misma. Un método es como una función. Puede recibir parámetros y devolver valores.
- **Atributos** o **Propiedades**. Almacenan información acerca del estado del objeto al que pertenecen (y por tanto, su valor puede ser distinto para cada uno de los objetos de la misma clase).

A la creación de un objeto basado en una clase se le llama instanciar una clase y al objeto obtenido se le conoce como instancia de esa clase.

Los pilares fundamentales de la POO son:

- **Herencia**. Es el proceso de crear una clase a partir de otra, heredando su comportamiento y características y pudiendo redefinirlos.
- **Abstracción**. Hace referencia a que cada clase oculta en su interior las peculiaridades de su implementación, y presenta al exterior una serie de métodos (interface) cuyo comportamiento está bien definido. Visto desde el exterior, cada objeto es un ente abstracto que realiza un trabajo.
- **Polimorfismo**. Un mismo método puede tener comportamientos distintos en función del objeto con que se utilice (relacionado con la *Herencia* y *Sobreescritura* de métodos).
- **Encapsulación**. En POO se agrupan dentro del objeto los datos y el código que los manipula.

Las ventajas más importantes que aporta la programación orientada a objetos son:

- **Modularidad**. Permite dividir los programas en módulos más pequeños, independientes unos de otros y que pueden comunicarse entre si, permitiendo además su reutilización.
- **Extensibilidad**. Se facilita la ampliación de nuevas funcionalidades, bien añadiendo nuevos métodos a las clases existentes o añadiendo nuevas clases.
- **Mantenimiento**. Facilita el mantenimiento de las aplicaciones al estar el código modularizado y mejor organizado (clases, paquetes, etc.).

1.1.- Características de la POO en PHP

Aunque PHP soportaba orientación a objetos en versiones antiguas, fue a partir de la versión 5 cuando se incorporaron los principales conceptos de POO, tales como:

- Métodos estáticos.
- Métodos constructores y destructores.
- Herencia.
- Interfaces.
- Clases abstractas.

PHP no soporta:

- Herencia múltiple (tampoco JAVA)
- Sobrecarga de métodos
- Sobrecarga de operadores (tampoco JAVA)

Al igual que ocurre en la mayoría de lenguajes orientadas a objetos, cuando se pasa un parámetro tipo objeto a una función o método se hace por referencia, no por copia.

1.2.- Creación de clases en PHP

Para crear una nueva clase, utilizaremos la palabra reservada ***class***, por convención suelen ponerse primero las variables miembro y luego los métodos, por ejemplo:

```php
<?PHP
    class Producto {
        private $codigo;
        private $descripcion;
        private $pcompra;
        private $pventa;
        private $stock;

        function __construct($codigo,$descripcion,$pcompra,$pventa,$stock) {
            $this->codigo=$codigo;
            $this->descripcion=$descripcion;
            $this->pcompra=$pcompra;
            $this->pventa=$pventa;
            $this->stock=$stock;
        }
        function margen() {
            return $this->pventa-$this->pcompra;
        }
    }
    $prod=new Producto('h020','Barra acero 16mm. longitud 6m.',35.30,45.40,50);
    echo $prod->margen();
?>
```

Es buena costumbre poner cada clase en su propio fichero, y también seguir la convención de que los nombres de clase empiezan por letra mayúscula y están guardadas en un fichero del mismo nombre ***.php.*** Para usar la clase haremos un ***require_once*** y para instanciar un objeto nuevo haremos uso del operador ***new***, como se ve en el ejemeplo anterior.

Cuando se declara un atributo debe especificarse su modificador de acceso:
- ***public***. El atributo está accesible fuera de los métodos de la clase
- ***private***. El atributo está accesible sólo en los métodos de la clase
- ***protected***. El atributo está accesible en los métodos de la clase y en los métodos de las clases que heredan de la misma.

Métodos mágicos

Son métodos cuyo nombre empieza por dos guiones bajos y que están asociados a acciones relacionadas con la clase, por ejemplo:

- ***__construct***. Está reservado para el constructor de la clase
- ***__set***. Para asignar a un atributo un valor
- ***__get***. Para obtener el valor de un atributo
- ***__toString***. Es llamado automáticamente cuando se imprime un objeto directamente

Siguiendo con el ejemplo anterior:

```php
<?PHP
    class Producto {
        static $porcentMargen=10;
        private $codigo;
        private $descripcion;
        private $pcompra;
        private $pventa;
        private $stock;

        function __construct($codigo,$descripcion,$pcompra,$pventa,$stock) {
            $this->codigo=$codigo;
            $this->descripcion=$descripcion;
            $this->pcompra=$pcompra;
            $this->pventa=$pventa;
            $this->stock=$stock;
        }
        function getMargen() {
            return $this->pventa-$this->pcompra;
        }
        function __toString() {
            return "('$this->codigo','$this->descripcion',$this->pcompra,$this->pventa,$this-
>stock)";
        }
        function __set($atributo,$valor) {
            if ($atributo=='pcompra' && $valor>=$this->pventa) {
                $this->pventa=$valor*(1+Producto::$porcentMargen/100);
            } elseif ($atributo=='codigo' && !ctype_alpha($valor[0])) {
                throw new Exception("Error el código debe empezar por una letra");
            }
            $this->$atributo=$valor;
        }
        function __get($atributo) {
            if ($atributo=='margen') {
                return $this->getMargen();
            } else {
                return $this->$atributo;
            }
        }
    }
    $prod=new Producto('h020','Barra acero 16mm. longitud 6m.',35.30,45.40,50);
    echo $prod->getMargen(),"\n";
    echo "$prod\n";
    $prod->codigo='h025';
    echo "$prod\n";
    $prod->pcompra=47.25;
    echo "$prod\n";
    echo "Margen: $prod->margen. \n";
    try {
        $prod->codigo='1235';
    } catch (Exception $ex) {
        echo "Se ha producido una excepción: ",$ex->getMessage(),"\n";
    }
?>
```

Al ejecutar dicho código obtenemos por pantalla:

10.1
('h020','Barra acero 16mm. longitud 6m.',35.3,45.4,50)
('h025','Barra acero 16mm. longitud 6m.',35.3,45.4,50)
('h025','Barra acero 16mm. longitud 6m.',47.25,51.975,50)
Margen: 4.725.
Se ha producido una excepción: Error el código debe empezar por una letra

Como se ve en el ejemplo podemos utilizar los métodos mágicos ***__set*** y ***__get*** para "simular" que los atributos de la clase son públicos, haciéndoles asignaciones directas y accediendo a sus valores, pero en realidad se está controlando la asignación de valores a dichos atributos mediante el método ***__set*** y el acceso a los atributos mediante el método ***__get***.

Cuando se llama a un método de un objeto, éste recibe una referencia al objeto mediante la variable ***$this***, como se ve en el ejemeplo anterior.

Cuando se instancia un objeto haciendo uso del operador ***new*** se nos devuelve una referencia al objeto. Si asignamos a una variable un objeto, ambas variables apuntarán al mismo objeto (al igual que ocurría en JAVA).

Además de métodos y propiedades, en una clase también se pueden definir constantes, utilizando la palabra reservada ***const***. Es importante no confundir con los atributos. Son conceptos distintos: las constantes no pueden cambiar su valor (obviamente, de ahí su nombre), no usan el carácter ***$*** y, además, su valor va siempre entre comillas y está asociado a la clase, es decir, no existe una copia del mismo en cada objeto. Por tanto, para acceder a las constantes de una clase, se debe utilizar el nombre de la clase y el operador **::**, llamado operador de resolución de ámbito, en el ejemplo anterior se vio su uso para acceder a una variable estática o de clase: **Producto:: $porcentMargen**. Ejemplo: *const IVA = 5;*

La diferencia entre una constante y una variable estática es que la primera no puede cambiar su valor bajo ninguna circunstancia y la segunda sí. A las variables ***static*** también se les llama variables de clase y su valor es compartido por todos los objetos de la clase.

Los nombres de constantes suelen escribirse en mayúsculas y se puede acceder a las mismas sin que exista ningún objeto de la clase previamente instanciado (igual que ocurre con las variables ***static)***. La forma de acceder sería, al igual que en el caso de las variables ***static***, ***NombreClase::CONSTANTE***.

En el caso de métodos ***static*** ocurre lo mismo, la forma de invocarlos es mediante el operador ***::*** y, al igual que ocurre en JAVA, estos métodos no pueden trabajar con variables no ***static***.

Los métodos y variables ***static*** pueden ser referenciadas dentro de un objeto haciendo uso de la palabra reservada ***self***. ***self*** hace referencia a la clase, mientras que ***$this*** hace referencia al objeto.

Constructores

El constructor de una clase, como ya vimos antes, es un *método mágico* (los métodos mágicos tienen un significado especial y su nombre comienza con un doble guion bajo). Tiene reservado el nombre ***__construct.*** Su misión es poder inicializar el estado del objeto. **Sólo puede haber un método constructor en cada clase**. Los métodos constructores, al igual que cualquier otro método, pueden tener parámetros opcionales, lo cual evita tener que sobrecargarlos por motivo de la diferencia de parámetros.

Podemos tener también un método ***__destruct*** que será ejecutado cuando se elimine el objeto, por ejemplo:

```
class Producto {
    private static $num_productos = 0;
    private $codigo;
    public function __construct($codigo) {
        $this->$codigo = $codigo;
        self::$num_productos++;
    }
    public function __destruct() {
        self::$num_productos--;
    }
    ...
}
```

```
    $p = new Producto('h399');
```

En el ejemplo llevamos el control del número de productos creados mediante la variable estática ***$num_productos***, cuando creamos un nuevo producto la incrementamos y cuando lo destruimos la decrementamos.

Para ver si un objeto pertenece a una clase podemos utilizar el operador ***instanceof***, por ejemplo: *if ($p instanceof Producto) { .... }*

Además a partir de PHP5 existen una serie de funciones útiles para trabajar con POO tales como get_class, class_exists, get_declared_classes, class_alias, get_class_methods, method_exists, get_class_vars, get_object_vars y property_exists (ver documentación de PHP (https://www.php.net/docs.php)

A partir de PHP5 se puede especificar en las funciones y métodos que reciban objetos como parámetros, la clase del objeto, p.ej.:

```
function margenBeneficio(Producto $p):float {
        return $p->getPrecioVenta()-$p->getPrecioCompra();
}
```

También a partir de PHP5 los objetos son referenciados. Cuando a una variable se le asigna otra que referencia a un objeto, ambas quedarán referenciando ("apuntando") al mismo objeto. Si lo que queremos es crear una copia del primer objeto deberemos llamar a la función clone:

```
$p2=clone($p); // ahora p2 "apunta" a un objeto distinto
```

Disponemos del método mágico ***__clone*** para particularizar la clonación de un objeto, por ejemplo si no queremos que se clonen exactamente todos los atributos y el código del producto quisiéramos que fuese único para cada objeto.

A la hora de comparar objetos hay una gran diferencia entre el operador == y el operador ===, el primero comparará si ambos objetos tienen sus atributos iguales y el segundo si ambas referencias "apuntan" al mismo objeto.

1.3.- Mecanismos de mantenimiento del estado

En una unidad anterior hablamos de la variable superglobal ***$_SESSION*** en la que almacenábamos el estado de la sesión. Si necesitamos que un objeto ***persista*** podemos utilizar la función ***serialize*** que nos devolverá el objeto convertido en un "***string"*** que contiene una serie de bytes que representan el estado del objeto. Por ejemplo:

```
$p=new Producto();
$a=serialize($p);
```

Esta cadena se puede almacenar en cualquier parte, como puede ser la sesión del usuario, o una base de datos. A partir de ella, es posible reconstruir el objeto original utilizando la función unserialize.

```
$p = unserialize($a);
```

Las funciones ***serialize*** y ***unserialize*** se utilizan mucho con objetos, pero sirven para convertir en una cadena cualquier tipo de dato, excepto el tipo ***resource***. Cuando se aplican a un objeto, convierten y recuperan toda la información del mismo, incluyendo sus atributos privados a excepción de los atributos estáticos de las clases.

Si simplemente queremos almacenar un objeto en la sesión del usuario, deberíamos hacer por tanto:

```
session_start();
$_SESSION['producto'] = serialize($p);
```

Pero esto no hace falta, si recuerdas del ejercicio del *carrito* de un tema anterior, guardábamos directamente el objeto en la sesión y éste permanecía en la misma, es serializado y deserializado automáticamente, por lo que no tenemos que preocuparnos de ello. Existen dos métodos mágicos relacionados con ***serialize*** y ***unserialize***: ***__sleep*** (llamado antes de serializar) y ***__wakeup*** (llamado después de deserializar), respectivamente.

1.4.- Herencia

```php
class Producto {
    private $codigo;
    private $nombre;
    private $nombreCorto;
    private $PVP;
    public function __construct($codigo,$nombre,$nombreCorto,$PVP) {
        $this->codigo=$codigo; $this->nombre=$nombre; $this->nombreCorto=$nombreCorto; $this->PVP=$PVP;
    }
    public function __toString() {
        return "$this->codigo, $this->nombre, $this->nombreCorto, $this->PVP";
    }
    public function __get($atributo) {
        return $this->$atributo;
    }
}
class Televisor extends Producto {
    private $pulgadas;
    private $tecnologia;
    function __construct($codigo,$nombre,$nombreCorto,$PVP,$pulgadas,$tecnologia) {
        parent::__construct($codigo,$nombre,$nombreCorto,$PVP);
        $this->pulgadas=$pulgadas; $this->tecnologia=$tecnologia;
    }
    function __toString() {
        return parent::__toString().", Televisor $this->pulgadas\", $this->tecnologia";
    }
}
$t=new Televisor("TV3255","Televisor Samsung 3255","TVSAMS3255","1200","50","LED");
if ($t instanceof Producto) {
    echo "$t, es instancia de Producto\n";
}
if ($t instanceof Televisor) {
    echo "$t, también es instancia de Televisor\n";
}
```

>>> ***get_parent_class***($t)
=> "Producto"
>>> ***is_subclass_of***($t,'Producto')
=> true

Si una clase tiene método constructor y una clase que hereda no, se llamará automáticamente al método constructor de la clase padre al instanciar un objeto de la clase hija. Si una clase que hereda tiene constructor habrá que llamar explícitamente al constructor del padre mendiante *parent::__construct(…)*.

Si a un método de una clase se le aplica la palabra clave ***final*** las clases que hereden no podrán sobreescribirlo, en el ejemplo anterior si hubiéramos declarado ***public final function __toString*** en la clase *Producto*, la clase *Televisor* no hubiera podido sobreescribirlo. También se puede aplicar ***final*** a una clase, con lo cual no podrá haber clases que hereden de la misma.

También podemos tener una clase ***abstract***, que sería aquella que tiene definido algún método ***abstract***. De esta clase no se pueden instanciar objetos y es necesario que haya alguna clase que herede de la misma para que sea de utilidad y podamos instanciar objetos, siempre que la clase que herede defina todos los métodos ***abstract*** declarados en la clase padre.

1.5.- Interfaces

```php
<?PHP
interface Automovil {
    public function getModelo();
    public function getMarca();
    public function getCilindrada();
    public function getNumRuedas();
    public function __toString();
}
class Coche implements Automovil {
    private $modelo, $marca, $cilindrada, $numRuedas;
    public function __construct($modelo,$marca,$cilindrada) {
        $this->modelo=$modelo; $this->marca=$marca; $this->cilindrada=$cilindrada; $this->numRuedas=4;
    }
    public function __get($atributo) {
        return $this->$atributo;
    }
    public function getModelo() { return $this->modelo; }
    public function getMarca() { return $this->marca; }
    public function getCilindrada() { return $this->cilindrada; }
    public function getNumRuedas() { return $this->numRuedas; }
    public function __toString() { return "[Coche, $this->marca, $this->modelo, $this->cilindrada]"; }
}
class Moto implements Automovil {
    private $modelo, $marca, $cilindrada, $numRuedas;
    public function __construct($modelo,$marca,$cilindrada) {
        $this->modelo=$modelo; $this->marca=$marca; $this->cilindrada=$cilindrada; $this->numRuedas=2;
    }
    public function __get($atributo) {
        return $this->$atributo;
    }
    public function getModelo() { return $this->modelo; }
    public function getMarca() { return $this->marca; }
    public function getCilindrada() { return $this->cilindrada; }
    public function getNumRuedas() { return $this->numRuedas; }
    public function __toString() { return "[Moto, $this->marca, $this->modelo, $this->cilindrada]"; }
}
$moto=new Moto('Ténéré','Yamaha',1000);
$coche=new Coche('Focus','Ford',1500);
echo $moto,"\n";
echo $coche,"\n";
?>
```

Todos los métodos que se declaran en un *interface* deben ser públicos, además pueden declararse constantes pero no atributos. Una clase puede implementar varios *interface* basta con declararlos tras la palabra clave *implements* separados por comas. La única restricción es que si se implementan varios interfaces los nombres de los métodos a implementar tengan nombres distintos, es decir que no exista un mismo nombre en dos interfaces distintos.

Los ***interfaces*** pueden heredar de otros interfaces.

Una de las dudas más comunes en POO, es qué solución adoptar en algunas situaciones: interfaces o clases abstractas. Ambas permiten definir reglas para las clases que los implementen o hereden respectivamente. Y ninguna permite instanciar objetos. Las diferencias principales entre ambas opciones son:

- En las clases abstractas, los métodos pueden contener código. Si van a existir varias subclases con un comportamiento común, se podría programar en los métodos de la clase abstracta. Si se opta por un interface, habría que repetir el código en todas las clases que lo implementen.
- Las clases abstractas pueden contener atributos, y las interfaces no.
- No se puede crear una clase que herede de dos clases abstractas, pero sí se puede crear una clase que implemente varios interfaces.

En PHP existen las funciones ***get_declared_interfaces***, que devuelve un array con los nombres de los interfaces declarados, e ***interface_exists*** que nos indica si está declarado un *interface*.

Ejercicio 1. ¿En el ejemplo anterior del interface, no sería mejor que Automovil fuera una clase ***abstract?***

2.- Programación en capas

https://desarrolloweb.com/articulos/que-es-mvc.html
https://twitter.com/midesweb
Vídeo PHP frameworks: https://www.youtube.com/watch?v=JsWQsucdSzM

En muchas ocasiones el código propio de la lógica de la aplicación se mezcla con el código necesario para crear el interfaz web que se presenta a los usuarios.

Existen varios métodos que permiten separar la lógica de presentación (en nuestro caso, la que genera las etiquetas HTML) de la lógica de negocio, donde se implementa la lógica propia de cada aplicación. El más extendido es el patrón de diseño ***Modelo – Vista – Controlador (MVC)***. Este patrón divide el código en tres partes, dedicando cada una a una función definida y diferenciada de las otras:

> ***Modelo***. Es el encargado de manejar los datos propios de la aplicación. Debe proveer mecanismos para obtener y modificar la información del mismo. Si la aplicación utiliza algún tipo de almacenamiento para su información (como un SGBD), tendrá que encargarse de almacenarla y recuperarla.
>
> ***Vista***. Es la parte del modelo que se encarga de la interacción con el usuario. En esta parte se encuentra el código necesario para generar el *interface de usuario* (en nuestro caso en HTML, CSS, JavaScript,...), según la información obtenida del modelo.
>
> ***Controlador***. En este módulo se decide qué se ha de hacer, en función de las acciones del usuario con su vista. Con esta información, interactúa con el modelo para indicarle las acciones a realizar y, según el resultado obtenido, envía a la vista las instrucciones necesarias para generar el nuevo interface.

La gran ventaja de este patrón de programación es que genera código muy estructurado, fácil de comprender y de mantener.

Es muy común utilizar el patrón MVC en conjunción con un framework. Existen numerosos frameworks disponibles en PHP, como por ejemplo Laravel, Codeigniter o Symfony, los cuales están basados en ***MVC***.

2.1. Separación de la lógica de negocio.

Un mecanismo muy extendido para separar la lógica de presentación y la de negocio son los motores de plantillas (template engines).

Un motor de plantillas web es una aplicación que genera una página web a partir de un fichero con la información de presentación, denominada plantilla o template, que vendría a ser la vista de ***MVC***. De esta forma es sencillo dividir el trabajo de programación de una aplicación web en dos perfiles: un programador, que se encargará del ***Modelo*** y la lógica de la aplicación, o sea el ***Controlador***, y un diseñador que se encargará de la elaboración de las plantillas que conforma la vista.

Existen varios motores de plantillas con diferentes características, nosotros vamos a utilizar uno de código abierto y disponible bajo licencia LGPL, llamado smarty (https://www.smarty.net/).

Entre las características de Smarty cabe destacar:
- Permite la inclusión en las plantillas de una lógica de presentación compleja.

- Acelera la generación de la página web resultante. Uno de los problemas de los motores de plantillas es que su utilización influye negativamente en el rendimiento. Smarty “compila” internamente las plantillas a scripts PHP equivalentes, y posibilita el almacenamiento del resultado obtenido en memoria temporal.

Al ser usado por una amplia comunidad de desarrolladores, existen multitud de ejemplos y foros para la resolución de los problemas que vayan apareciendo.

Para instalar Smarty, debemos descargar la última versión desde su sitio web, y seguir los siguientes pasos:

1. Descomprimir los archivos de Smarty en */usr/share/php/*, debes hacerlo como ***root*** pues esa carpeta pertenece a ***root***.
2. Modificar el fichero */etc/php/x.x/apache/php.ini (x.x es la versión de PHP que estás usando)* para que se incluya en la variable ***include_path*** de PHP la ruta en que acabas de descomprimir Smarty, y reiniciar Apache para que se apliquen los cambios, por ejemplo:
   include_path = ".:/usr/share/php:/usr/share/php/smarty-4.0.0/libs"
3. Crear la estructura de directorios necesaria para Smarty. Concretamente debes crear cuatro directorios, con nombres ***templates***, ***templates_c***, ***configs*** y ***cache***. Es conveniente que estén ubicados en un lugar no accesible por el servidor web, y en una ubicación distinta para cada aplicación web que programes. Por ejemplo, puedes crear en la carpeta */var/www/phpdata* una carpeta llamada ***smarty***, y bajo ella otra con el nombre de cada aplicación que será la que contendrá las carpetas mencionadas. El usuario con el que se ejecuta ***apache2*** debe tener permisos de lectura/escritura en dichas carpetas.

Para utilizar Smarty, haremos un ***require_once(“Smarty.class.php”);*** y luego debemos instanciar una objeto de la clase y configurar la ruta de las carpetas mencionadas anteriormente, por ejemplo:

```
require_once('Smarty.class.php');
$smarty = new Smarty();
$smarty->template_dir = '/var/www/phpdata/smarty/gestisimal/templates/';
$smarty->compile_dir = '/var/www/phpdata/smarty/gestisimal/templates_c/';
$smarty->config_dir = '/var/www/phpdata/smarty/gestisimal/configs/';
$smarty->cache_dir = '/var/www/phpdata/smarty/gestisimal/cache/';
```

Podemos evitar tener que hacer eso de la siguiente forma:

```
<?php
        require_once("Smarty.class.php");
        class miSmarty extends Smarty {
                private string $app, $rutaBase;
                public function __construct($app) {
                        parent::__construct();
                        $this->app=$app;
```

```
                $this->rutaBase=$rutaBase="/var/www/phpdata/smarty/".$app;
                $this->template_dir="$rutaBase/templates/";
                $this->compile_dir="$rutaBase/templaces_c/";
                $this->config_dir="$rutaBase/configs/";
                $this->cache_dir="$rutaBase/cache/";
                $this->assign('css_dir','../Vista/css');
                $this->assign('imgs_dir','../Vista/imagenes');
        }
        public function getRutaBase() {
                return $this->rutaBase;
        }
        public function getApp() {
                return $this->app;
        }

}
```

Hemos creado nuestra propia clase ***miSmarty*** que hereda de la clase ***Smarty***, y aprovechamos el constructor para poner los parámetros de forma automática, sólo necesitamos pasarle el nombre de la aplicación. Además hemos creado dos nuevas variables de plantilla *css_dir* e *imgs_dir*, que usaremos en las plantillas para localizar las hojas de estilo y las imágenes.

2.2. Generación de vistas con *Smarty*

Las plantillas de Smarty son ficheros con extensión `tpl`, en los que se puede incluir cualquier contenido propio de una página web. Se intercalarán delimitadores para indicar la inclusión de datos y de lógica propia de la presentación.

Los delimitadores de Smarty son llaves, si algún trozo de código llevase llaves habría que encerrarlo entre {literal}...{/literal}, esto indicará a *Smarty* que lo que va enmedio de ambas etiquetas debe ser copiado literalmente en la salida, por ejemplo código *JavaScript* (ver ejemplo más abajo). De los distintos elementos que puedes incluir entre las llaves están:

- Comentarios. Van encerrados entre asteriscos.

  ```
  {* Este es un comentario de plantilla en Smarty *}
  ```
- Variables. Se incluye simplemente su nombre, precedido por el símbolo `$`. También se pueden especificar modificadores, separándolos de la variable por una barra vertical. Existen varios modificadores para, por ejemplo, dar formato a una fecha (`date_format`) o mostrar un contenido predeterminado si la variable está vacía (`default`). Ejemplos:

  ```
        - {$producto→codigo}
        - $smarty->assign('yesterday', strtotime('-1 day')); //esto estaría
en el controlador
          {$yesterday|date_format} // y esto en la vista
        - {$myTitle|default:'no hay título'}
  ```

- Estructura de procesamiento condicional: `if`, `elseif`, `else`. Permite usar condiciones, de forma similar a PHP, para decidir si se procesa o no cierto contenido.

  ```
  {if empty($productoscesta)}
        <p>Cesta vacía</p>
  {else}
      …
  {/if}
  ```
- Bucles: `foreach`. Son muy útiles para mostrar varios elementos, por ejemplo en una tabla. Deberás indicar al menos con `from` el array en el que están los elementos, y con `item` la variable a la que se le irán asignado los elementos en cada iteración.

  ```
  {foreach from=$productoscesta item=producto}
      <p>{$producto->codigo}</p>
  {/foreach}
  ```

- Inclusión de otras plantillas. Smarty permite descomponer una plantilla compleja en trozos más pequeños y almacenarlos en otras plantillas, que se incluirán en la actual utilizando la sentencia `include`.

```
<div id="cesta">
    {include file="productoscesta.tpl"}
</div>
<div id="productos">
    {include file="listaproductos.tpl"}
</div>
```

La documentación completa de *Smarty* se puede consultar en https://smarty-php.github.io/smarty/.

Ejemplo de utilización:
En el controlador cuando queramos cargar una vista haremos lo que se muestra en el siguiente ejemplo:

```
require_once("../Modelo/DAOProducto.php");
require_once("../../../miSmarty.php");
$smarty=new miSmarty("gestisimal");
$listaProductos=DAOProducto::listaProductos();
$metaInfo=DAOProducto::getLongitudCampos();
$smarty->assign('productos',$listaProductos);
$smarty->assign('metaInfo',$metaInfo);
$smarty->display("../Vista/vistaProducto.tpl"); //aquí “cargamos” la vista
```

Un ejemplo de plantilla sería (fichero vistaProducto.tpl):

```
<!DOCTYPE html>
<!-- Desarrollo Web en Entorno Servidor -->
<html>
	<head>
		<meta http-equiv="content-type" content="text/html; charset=UTF-8">
		<title>Ejemplo Tema 6: Listado de Productos con Plantillas</title>
		<link href="{$css_dir}/productos.css" rel="stylesheet" type="text/css">
		{literal}
			<script>
				function despliega(capa) { capa.style.height='auto'; }
				function comprime(capa) { capa.style.height='27px'; }
				function mensaje(mensa) { alert(mensaje); }
			</script>
		{/literal}
	</head>
	<body>
		<header id="encabezado">
			<h1>Listado de productos</h1>
		</header>
		<main>
			<!-- Dividir en varios templates -->
			<div id="productos">
				{include file="./listaproductos.tpl"}
			</div>
		</main>
		<footer id="pie">
			<span class='amarilla'>
			GESTISIMAL, S.A. - Camino del Cedro Alto, 35. - Tenerife
			</span>
		</footer>
	</body>
</html>
```

ListaProductos.tpl:

```
{foreach from=$productos item=producto}
  <p><form id='{$producto->codigo}' action='productos.php' method='post'>
  <input type='hidden' name='cod' value='{$producto->codigo}'/>
  <input type='submit' name='enviar' value='Añadir'/>
  {if ($producto instanceof Ordenador) }
    <a href="detalleOrdenador.php?codProd={$producto->codigo}">{$producto->nombre_corto}: {$producto->PVP} euros.</a>
```

```
    {else}
        {$producto->nombre_corto}: {$producto->PVP} euros.
    {/if}
    </form></p>
{/foreach}
```