TEMA 2: LENGUAJES DE MARCAS (PHP)

Contenido

NOTA: Estos apuntes están basados en unos apuntes desarrollados por Jorge Sánchez Asenjo para el módulo Implantación de Aplicaciones Web (www.jorgesanchez.net).

1.- PHP

Las páginas web se crean mediante HTML, y este es un lenguaje muy limitado para atender a los requerimientos que actualmente se exigen. Por ello han aparecido numerosas extensiones al lenguaje que permiten enriquecer las páginas web.

Muchas mejoras están orientadas al cliente, es decir que se trata de código de otros lenguajes (llamados lenguajes de script) que se añaden al código HTML y que el ordenador que recibe la página debe interpretar a través del software apropiado. Por lo tanto el cliente de la página debe poseer el software apropiado. Y esto es un problema.

Por ello aparecieron lenguajes y mejoras en el lado del servidor. De modo que el programador añade al código HTML código de otro lenguaje script de la misma manera que el párrafo anterior. La diferencia es que este código no se le envía al cliente sino que es el servidor el que le interpreta. El cliente recibirá una página HTML normal y será el servidor el que traduzca el código script.

PHP es el lenguaje script de servidor más popular. Fue el primero en aparecer aunque realmente empezó a imponerse en torno al año 2000 por encima de ASP que era la tecnología de servidor reinante.

Hoy en día se pueden instalar módulos para interpretar PHP en casi todos los servidores de aplicaciones web. En especial PHP tiene una gran relación con Apache.

Es un lenguaje basado en C y en Perl, que se ha diseñado pensando en darle la máxima versatilidad y facilidad de aprendizaje, por encima de la rigidez y coherencia semántica.

PHP tiene como características:

1. Multiplataforma. A diferencia de otros lenguajes (especialmente de ASP y ColdFussion), se trata de un lenguaje que se puede lanzar en casi todas las plataformas de trabajo (Windows, Linux, Mac,…)
2. Abierto y gratuito. Pertenece al software licenciado como GNU, la licencia del sistema Linux; lo que permite su distribución gratuita y que la comunidad mejore el código.
3. Gran comunidad de usuarios. La popularidad de PHP, junto con la gran defensa que de él hacen los defensores del código abierto, permite tener una comunidad amplia y muy dinámica a la que acudir en caso de necesidad.
4. Apache, MySQL. Apache es el servidor web y de aplicaciones más utilizado en la actualidad. MySQL es el servidor de bases de datos relacionales más popular en Internet para crear aplicaciones web. Puesto que PHP tiene una gran relación y compatibilidad con ambos productos (está de hecho muy pensado para hacer tándem con ellos), esto se convierte en una enorme (y a veces determinante) ventaja.

5. Extensiones. Dispone de un enorme número de extensiones que permiten ampliar las capacidades del lenguaje, facilitando la creación de aplicaciones web complejas.
6. ¿Fácil? Es un punto muy controvertido. Los programadores PHP entusiastas, defienden esta idea; es indudable además que fue uno de los objetivos al crear este lenguaje. Sin embargo Microsoft defiende con energía que ASP permite crear aplicaciones web complejas con gran facilidad; y parece indudable que el lenguaje ColdFussion de Macromedia (ahora de Adobe) es más sencillo de aprender.
7. Las características de PHP correspondientes a la libertad de creación y asignación de valores a variables, tipos de datos poco restrictivos, y otras ausencias de reglas rígidas suelen ser los puntos que defienden los programadores de PHP para estimar su facilidad de aprendizaje. Sin embargo los programadores de lenguajes formales como C y Java, seguramente se encontrarán con más problemas que ventajas al aprender este lenguaje.

2.- HERRAMIENTAS Y SOFTWARE NECESARIO

Para crear aplicaciones web en PHP necesitamos disponer de software específico que complica los requisitos previos que tiene que tener la máquina en la que deseemos aprender y probar el lenguaje. Lo imprescindible es:

- Un servidor web compatible con PHP. La mayoría lo son, pero parece que la opción de usar Apache es la más recomendable por la buena relación que han tenido ambos productos. No obstante cada vez es más habitual utilizar PHP en servidores como IIS o ngnx.
- Motor PHP. Se trata del software que extiende al servidor web anterior para conseguir que se convierta en un servidor de aplicaciones web PHP. Necesitamos descargar e instalar dicho motor correspondiente al módulo de PHP compatible con el servidor web con el que trabajemos.
- IDE para PHP. Un IDE es un entorno de desarrollo integrado; es decir, un software que facilita la escritura de código en otro lenguaje. En realidad se puede escribir código PHP en cualquier editor de texto (como el Bloc de Notas de Windows por ejemplo); pero estos entornos facilitan la edición de código (con coloreado especial de las palabras de PHP, corrección del código en línea, abreviaturas de código, plantillas,…) y su prueba y depuración.

  Los IDEs más populares son:

  - Basados en Eclipse. Eclipse es un entorno de programación de aplicaciones pensado para Java pero que dispone de muchas extensiones que permiten programar para diferentes lenguajes, convirtiéndose así en la plataforma de programación más popular de la actualidad. Las extensiones de Eclipse más populares para programar en PHP son:
    - Aptana. Sea la versión sólo para PHP o la completa Studio Pro con posibilidad de usar en varios lenguajes es una de las más populares. Muy famosa para crear código Javascript, se ha adaptado con grandes resultados para PHP ya que permite casi todo lo necesario sobre este lenguaje.
    - PHP Designer Toolkit (PDT). Considerada la extensión oficial de Eclipse para programar en PHP, es quizá la más utilizada en la actualidad.
  - Zend Studio. Dispone de un framework (una plataforma) muy famosa para crear PHP usando plantillas. Su entorno compite con los anteriores en prestaciones.
  - Netbeans. Se trata del entorno libre de programación competidor con Eclipse. Ambos se crearon para programar en Java, pero han extendido su uso a otros lenguajes. NetBeans dispone de una extensión para PHP, incluso se puede descargar una versión ligera de NetBeans sólo para PHP. Es una de las mejores opciones para programar en PHP.
  - Adobe Dreamweaver. Se trata del software comercial de creación de páginas web más famoso del planeta. Tiene capacidad para escribir código PHP e incluso facilidades gráficas para hacerlo. Es inferior a los anteriores en cuanto a escritura de

código, pero muy superior cuando queremos concentrarnos en el diseño del sitio.

- Microsoft Expression Web. Software comercial competidor del anterior (aunque por ahora mucho menos popular) y con capacidad de usar con PHP.

- Depurador PHP. Se trata de un software que se añade al módulo PHP para darle la capacidad de depurar el código. Los más utilizados por su potencia son Zend Debugger y XDebug.

3.- BASE DE PHP

3.1.- Como escribir PHP

Cuando en un documento web queremos añadir código php se indica la etiqueta <?php …. ?>:

```
<?php
//codigo php
?>
```

El código PHP se coloca en la zona de la página web donde más nos interese hacerlo. Un primer documento PHP podría ser:

```
<?php
echo "<!DOCTYPE html>";
?>
<html>
<head>
      <meta charset="UTF-8">
      <title> Ejemplo</title>
</head>
<body>
<?php
      Echo "Hola";
?>
</body>
</html>
```

Así en el ejemplo anterior podemos ver cómo hemos usado PHP para escribir HTML. En este caso se ha usado la función ***echo*** para enviar código HTML al cliente. Como la salida la entiende el cliente directamente como HTML (o javascript, css, pdf, etc), es posible devolverle cualquier elemento en este lenguaje.

```
echo "<span>esto es html</span><br>".PHP_EOL;
```

Dentro del código PHP se pueden hacer tres tipos de comentarios:

- **Comentarios de varias líneas**. Al igual que en lenguaje C, comienzan por**/*** yterminan por ***/.**

```
/*
    * Este comentario ocupa varias lineas
    */
```

- **Comentarios de una línea estilo C++**. Se ponen tras la barra doble //.

```
echo "Texto html"; //comentario en línea
```

- Comentarios de una línea estilo **ShellScript**. Se ponen tras la almohadilla.

```
echo "Texto html"; #Otro comentario en línea
```

Cuando escribamos PHP hemos de tener en cuenta que:

- Todas las líneas de código deben de finalizar con un punto y coma.
- Se puede agrupar el código en bloques que se escriben entre llaves
- Una línea de código se puede partir o sangrar (añadir espacios al inicio) a voluntad con el fin de que sea más legible, siempre y cuando no partamos una palabra o un valor.
- PHP obliga a ser estricto con las mayúsculas y las minúsculas en algunos casos como el nombre de las variables; sin embargo, con las palabras reservadas del lenguaje o las funciones no es estricto. Es decir, PHP entiende que ***WHILE***, ***while***es lo mismo al ser una palabra reservada. Sin embargo $var y $VAR no son iguales al ser el nombre de una variable.

3.2.- Las variables

En todos los lenguajes de programación (y PHP no es una excepción) Las variables son contenedores que sirven para almacenar los datos que utiliza un programa. Dicho más sencillamente, son nombres que asociamos a determinados datos. La realidad es que cada variable ocupa un espacio en la memoria RAM del servidor que ejecute el código para almacenar el dato al que se refiere. Es decir, cuando utilizamos el nombre de la variable realmente estamos haciendo referencia a un dato que está en memoria.

Las variables tienen un nombre (un **identificador**) que tiene que cumplir estas reglas:

- Tiene que empezar con el símbolo **$**. Ese símbolo es el que permite distinguir a una variable de otro elemento del lenguaje PHP.
- El segundo carácter puede ser el guion bajo (_) o bien una letra.
- A partir del tercer carácter se pueden incluir números, además de letras y el guión bajo
- No hay límite de tamaño en el nombre
- Por supuesto el nombre de la variable no puede tener espacios en blanco (de ahí la posibilidad de utilizar el guión bajo)

Es conveniente que los nombres de las variables indiquen de la mejor forma posible su función. Es decir: ***$saldo*** es un buen nombre, pero ***$x123*** no lo es aunque sea válido.

También es conveniente poner a nuestras variables un nombre en minúsculas. Si consta de varias palabras el nombre, podemos separar las palabras con un guión bajo en vez del espacio o empezar cada nueva palabra con una mayúscula. Por ejemplo: ***$saldo_final*** o ***$saldoFinal***.

La primera sorpresa para los programadores de lenguajes estructurados es que en PHP no es necesario declarar una variable. Simplemente se utiliza y ya está. Es decir, si queremos que la variable ***$edad*** valga ***15***, haremos:

```
$edad=15;
```

Y no será necesario indicar de qué tipo es esa variable. Esto es tremendamente cómodo pero también nos complica la tarea de depurar nuestros programas al no ser nada rígido el lenguaje y permitir casi todo.

PHP pone a disposición del programador variables de sistema predefinidas llamadas ***superglobales***. La mayoría son simplemente informativas y contienen información que podemos manejar en nuestra aplicación.

- ***$_SERVER***: Da información relevante tanto del servidor como del cliente. Incorpora las cabeceras HTTP. Por ejemplo $_SERVER[“DOCUMENT_ROOT”] (raíz del sitio web en el servidor) o $_SERVER[“HTTP_USER_AGENT”] (cabecera USER AGENT de la solicitud).
- ***$_GET, $_POST y $_REQUEST***. Contienen los elementos del formulario enviados.
- ***$_COOKIE***. Cookies almacenadas en el cliente.
- ***$_FILES***. Archivos subidos desde el cliente.
- ***$_ENV***. Información de variables de entorno.

Estas variables se verán a lo largo de este curso.

La asignación se realiza mediante el operador = y consiste en darle un valor a una variable.

```
$cadena = 15;    //asigno un entero
$cadena = 12.7;    //asigno un real
$cadena = "hola";  //asigno una cadena
$cadena = 100;     //cambio el tipo de $cadena a entero
```

La variable tendrá el tipo correspondiente a su valor. Como se ve $cadena ha tenido sucesivamente tipo entero, real, cadena y de nuevo entero.

Las variables se declaran en el momento de la asignación. Un problema surge cuando queremos utilizar variables a las que no se les ha asignado ningún valor. Como:

```
echo $x; //error: la variable no se ha asignado previamente
$y=$x+2; //error: $x no está declarada
```

Ocurrirá un error al hacer ese uso de la variable. Aunque en realidad la directiva del archivo **php.ini, error_reporting** permite modificar los errores de aviso que se lanzan. Si bajamos su nivel de aviso, no detectará estos fallos. También podemos usar esa función al inicio de nuestro código para indicar fallos. Sería:

```
error_reporting(E_ALL); //mostrar todos los errores
```

Si deseamos que no nos avise de estos fallos. Habría que pasar a error_reporting otra constante; por ejemplo,**E_USER_ERROR** avisa sólo si hay errores de nivel usuario (o más graves) en el código

A las variables sin declarar se les da valores por defecto, que dependerán del contexto en que se usen. Hay una función llamado **isset** a la que se le pasa una variable e indica si la variable tenía asignado un valor; en cualquier otro caso devuelve falso. Ejemplo:

```
echo isset($x); //devuelve falso, no estaba declarada previamente
```

A su vez, si queremos "eliminar" una variable se usa la función **unset**.

```
$x=12; //declaro la variable x
echo isset($x); //devuelve true porque ya está declarada
unset($x);      //elimino la variable x
echo isset($x); //devuelve false, porque ya no está declarada
```

Una variable puede contener un valor de cualquier tipo. Además, y a diferencia de otros lenguajes, se puede extender al concepto de variable a variable-variable, variable-función y variable-clase:

- ***Variable-variable***: En este caso el contenido de una variable es el nombre de otra variable que podemos usar.

```
$nombre='Profesor'; //defino la variable $nombre
$var="nombre";   //defino una variable cuyo contenido será
                 //el nombre de otra variable
echo $$var;      //accedo mediante mecanismo variable-variable.
                 //$$var => $nombre => 'Profesor'
```

- ***Variable-función***: El contenido de una variable es el nombre de una función o una función anónima. Esta variable podría ejecutarse directamente.

```
//forma 1
function f1($x)  //creo la función
{
      echo $x;
}

$var="f1";       //asigno a la variable el nombre de la funcion
$var(12);        //llamo a la funcion usando la variable-funcion

//forma2
$var=function ($x){  //creo la variable-funcion desde una
      echo $x;       //funcion anonima
```

```
};
$var(34);       //llamo a la funcion usando la variable-funcion
```

- ***Variable-clase***: El contenido de una variable es el nombre de una clase. Esta variable se puede usar directamente como si fuera la clase.

```
class MiClase{   //defino la clase
      ......
}

$var="MiClase";   //la variable contiene el nombre de la clase

$objeto=new $var();  //creo una nueva instancia de la clase
$var::......;        //accedo a un elemento estatico de la clase
```

3.3.- Tipos de datos.

Como se ha dicho antes, las variables no se declaran con un tipo especificado sino que dependerá del valor asignado a la misma. Sin embargo, es posible saber ese tipo mediante la función ***gettype*** que devuelve una cadena que lo identifica.

```
$var=12;
echo gettype($var); //mostraría integer
echo gettype("adios");  //mostraría string
```

Los valores posibles son:

- Integer: entero
- Double: real
- Boolean: booleano
- String: cadena de caracteres
- Array: Vector o array de elementos.
- Object: Objeto
- Resource: Recurso creado (Por ejemplo, descriptor de Base de Datos MySql)
- NULL: variable sin valor

Existen también toda una serie de funciones que nos permiten comprobar si una expresión es de un tipo concreto, las funciones ***is_XXX***, donde XXX es el tipo que queremos comprobar (is_integer, is_bool, is_float, is_object, ….)

ENTEROS

A las variables se les puede asignar valores enteros. Los números enteros se usan tal cual. Pueden ser positivos o negativos, e indicarle valores de diversas bases:

```
$num = 12;     //entero positivo
$num = -100;   //entero negativo
$numBinario = 0b10011; //numero binario
$numOctal = 02361; //numero octal
$numOctal = 0o2361; //numero octal
$numHexa = 0x12A;  //numero hexadecimal
```

Como se ve, se puede asignar a una variable un numero binario (empieza por 0b), octal (empieza por 0) o hexadecimal (empieza por 0x).

REALES

Los números decimales en PHP son de tipo coma flotante. Este es un formato decimal para máquinas digitales que se manejan muy rápido por parte de un ordenador y ocupan poco en memoria.

Para asignar decimales hay que tener en cuenta en PHP que el formato es el inglés, por lo que las cifras decimales se separan usando un punto como separador decimal. Además, es posible usar notación científica. Ejemplos:

```
$real = 123.45; //numero real
$real1 = 23e2;  //numero real en notacion cientifica
```

Un problema de los números reales es la inexactitud, Para representar un número real se tiene un número fijo de dígitos significativos. Nunca se ha de confiar en resultados de números flotantes hasta el último dígito, y no comparar la igualdad de números de punto flotante directamente. Por ejemplo, floor((0.1+0.7)*10) usualmente devolverá 7 en lugar del 8 previsto, ya que la representación interna será algo así como 7.9999999999999991118. Para comparar, se puede usar un valor que represente el error aceptable. Así para comparar dos números con un error inferior a 0.00001 se podría tener:

```
<?php
$a = 1.23456789;
$b = 1.23456780;
$epsilon=0.00001;

if (abs($a-$b)<$epsilon){
      echo "iguales";
}
?>
```

Algunas operaciones numéricas pueden resultar en un valor representado por la constante **NAN**. Este resultado representa un valor no definido o no representable mediante cálculos de punto flotante. Cualquier comparación, ya sea estricta o no, de este valor con cualquier otro valor, incluido él mismo, tendrá un resultado de `FALSE`.

Ya que **NAN** representa cualquier número de diferentes valores, `NAN` no debería

compararse con otros valores, incluido él mismo; en su lugar debería comprobarse usando la función ***is_nan***().

CADENAS

Se denomina así a los textos, que en programación se les denomina cadenas de caracteres o **Strings**. Se asignan a las variables entrecomillando (en simples o dobles) el texto a asignar.

Si el propio texto lleva comillas, se puede utilizar combinaciones de las comillas para evitar el problema.

```
$cad="Esto es una cadena";   //definicion de cadena con "
$cad='Y esto otra';     //definicion de cadena con '
$cad="Juan 'El Magnifico' está aquí"; //inclusión de ' en una cadena
$cad='Pedro "El Secundario" también '; //inclusion de " en una cadena
```

En las cadenas con comillas “ se pueden realizar unas operaciones adicionales:

- Usar caracteres especiales.

secuencia de escape	Significado
\t	Tabulador
\n	Nueva línea
\f	Alimentación de página
\r	Retorno de carro
\”	Dobles comillas
\’	Comillas simples
\\	Barra inclinada (***backslash***)
\$	Símbolo dólar

- Incluir el valor de una variable. La variable puede indicarse directamente o usando {} para delimitar donde empieza y acaba.

```
$cad="Juan \"El Valiente\" se asustó\n"; //uso de caracteres especiales

$nombre="Rosa";
$cad1="Le he dado dinero a $nombre\n";   //inclusion de una variable
$cad2="Le he dado \$ a {$nombre}\n"; //delimito la variable con {}
```

BOOLEANOS

Sólo pueden tomar como valores **TRUE** (verdadero) o **FALSE** (falso);

```
$verdadero= true;
echo $verdadero; //escribe 1
```

Sorprende que echo devuelva el valor uno; la explicación es que True está asociado a valores positivos, mientras que False se asocia al cero.

NULL

El valor especial **null** representa una variable sin valor. **null** es el único valor posible del tipo null.

Una variable se considera null si:

- Se le ha asignado la constante null
- No se le ha asignado unvalor todavía.
- Se ha destruido con unset().

En PHP se realizan **conversiones de tipos** cuando se usa una variable en una expresión que involucra otros tipos. Las conversiones pueden ser implícitas o explícitas.

Conversiones Implícitas Se realizan de forma automática cuando el tipo de la variable no corresponde con el de la "expresión".

Conversiones Explícitas El usuario indica el tipo al que quiere convertir una expresión. Se puede hacer precediendo la expresión por **(tipo)** (forzado)**,** mediante la función **settype**() o usando funciones específicas como **intval, floatval, boolval o stringval**.

```
$num = 12; //num es entero
$num = (float)$num;  //num lo he convertido en real
$num = intval(16.34); //num pasa a ser el entero 16
```

Si usamos la expresión **(tipo)** es posible:

- (int), (integer) - forzado a integer
- (bool), (boolean) - forzado a boolean
- (float), (double), (real) - forzado a float
- (string) - forzado a string
- (array) - forzado a array
- (object) - forzado a object

Las conversiones implícitas más usuales entre tipos son:

- *A boolean*. Valdrá false
    - el boolean **`FALSE`** mismo
    - el integer 0 (cero)
    - el float 0.0 (cero)
    - el valor string vacío, y el string "0"
    - un array con cero elementos
    - el tipo especial NULL (incluidas variables no establecidas)
    - objetos SimpleXML creados desde etiquetas vacías

- A integer (int). Se hace automáticamente cuando en la expresión se requiere un argumento entero
    - Desde booleanos: 0 (false), 1 (true)
    - Desde float: será redondeado el número hacia arriba. Si el número real está fuera de los límites del integer, el valor será indefinido.
    - Desde strings: Se analiza la cadena y si se puede extraer un entero al principio, se devuelve el número. P.ej.: `12+"3sdre" valdría 15`
    - Desde otros tipos: Es indefinido

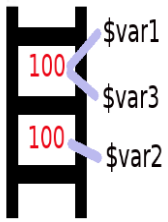
- A float (double). La conversión es semejante a la de cualquier tipo a integer.

- A string.
    - Desde booleanos: "1" (true) y "" (false).
    - Desde integer o float: String que representa textualmente el número.
    - Desde array: string 'array'.
    - Desde object: string 'object'.
    - Desde resource (recursos): String "Resource id #XXX".

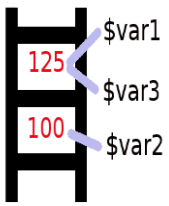
3.4.- Referencias (&).

Realmente, una variable no es más que un identificador que apunta a una zona de memoria donde se almacena el valor. Como en otros lenguajes, podemos hacer que más de una variable nos apunte a la misma zona de memoria. Esto se consigue con el operador de referencia (&).

Por defecto, la asignación entre variables siempre se hace por valor (se crea una nueva copia de la zona de memoria). Con el operador & hacemos que la asignación se haga por referencia (comparten la misma zona de memoria).




```
$var1=100;      //creo la zona de memoria
$var2=$var1;    //hago asignacion por valor, creo una
                //nueva zona de memoria para $var2
$var3=&$var1;   //asigno por referencia, $var1 y $var3
                //comparten la misma zona de memoria
```




```
$var1=125;      //se modifica el valor de la zona de
                //memoria apuntada por $var1 y por
                //tanto $var3 cambia su valor
```

Siempre se realiza la asignación por valor salvo para los objetos en los que se realiza la asignación por referencia.

3.5.- Constantes

Las constantes almacenan valores que no cambian en el tiempo. La forma de definir constantes es gracias a la función **define**. Que funciona indicando el nombre que tendrá la constante, entre comillas, y el valor que se le otorga.

```
define("PI",3.141592);
```

Las constantes no utilizan el signo **$** de las variables, simplemente utilizan el nombre. Es decir escribir el valor de la constante PI tras haberla declarado con la instrucción anterior, sería:

```
echo PI.PHP_EOL;
```

Desde la versión 5.3 de PHP es posible definir una constante de una forma más estructurada. Se trata de utilizar la palabra clave **const** habitual en la mayoría de lenguajes de programación.

```
const PI=3.141592;
```

Aunque no es obligatorio, es conveniente la regla de nombrado **UPPER_SNAKE_CASE** para constantes. Esto es, empleamos mayúsculas y guiones bajos para separar las palabras. ya que se considera una norma de buen estilo de escritura en cualquier lenguaje de programación.

3.6.- Operadores

Lo habitual al programar en PHP es utilizar expresiones que permiten realizar comprobaciones o cálculos. Las expresiones dan un resultado que puede ser de cualquiera de los tipos de datos comentados anteriormente (enteros, decimales, booleanos, strings,…)

Aritméticos

Son:

operador	significado
+	Suma
-	Resta
*	Producto
/	División
%	Módulo (resto)

Ejemplos:

```
$x=15.5;
$y=2;

echo $x+$y,"<br />"; //escribe 17.5
echo $x-$y,"<br />"; //escribe 13.5
```

```
echo $x*$y,"<br />"; //escribe 31
echo $x/$y,"<br />"; //escribe 7.75
echo $x%$y,"<br />"; //escribe 1, sólo coge la parte entera
```

El operador / realiza la división real. Si queremos realizar la división entera tenemos la función

```
$num = 10/3;          //3.333
$num = intdiv(10,3);  //3
```

Operadores condicionales

Sirven para comparar valores. Siempre devuelven valores booleanos. Son:

operador	significado
<	Menor
>	Mayor
>=	Mayor o igual
<=	Menor o igual
==	Igual, devuelve verdadero si las dos expresiones que compara son iguales
===	Equivalente, devuelve verdadero si las dos expresiones que compara son iguales y además del mismo tipo
!=	Distinto
!	No lógico (NOT)
!==	Distinto, en valor o en tipo
<=>	(nave espacial) Compara dos valores y devuelve un entero.
&&	"Y" lógico
AND	"Y" lógico
||	"O" lógico
OR	"O" lógico
XOR	"OR" exclusivo
??	Devuelve el primer operador de izquierda a derecha que no sea null

Los operadores lógicos (**AND**, **OR** y **NOT**), sirven para evaluar condiciones complejas.**NOT** sirve para negar una condición. Ejemplo:

```
$edad = 21;
$mayorDeEdad = $edad >= 18; //mayorDeEdad será true
$menorDeEdad = !$mayorDeEdad; //menorDeEdad será false
echo $mayorDeEdad."\t".$menorDeEdad;
```

El operador **&&** (**AND**) sirve para evaluar dos expresiones de modo que si ambas son ciertas, el resultado será **true**. En cualquier otro caso el resultado será **false**. Ejemplo:

```
$carnetConducir = TRUE;
$edad = 20;
$puedeConducir= ($edad>=18) && $carnetConducir;
```

```
//puedeConducir es TRUE, puesto que edades mayor de 18
//y carnet es TRUE
```

El operador || (**OR**) sirve también para evaluar dos expresiones. El resultado será **true** si al menos uno de las expresiones es **true**. Ejemplo:

```
$nieva = TRUE;
$llueve = FALSE;
$graniza = FALSE;
$malTiempo = $nieva || $llueve || $graniza;
//malTiempo es TRUE porque nieva
```

La diferencia entre la igualdad y la equivalencia se puede explicar con este ejemplo:

```
$resultado = (18 == 18.0); //resultado es verdadero
$resultado = (18 === 18.0); //resultado es falso
```

En el ejemplo ***18===18.0*** devuelve falso porque aunque es el mismo valor, no es del mismo tipo.

El operador <=> devuelve -1, 0 o 1 según el primer operando sea menor, igual o mayor al segundo

```
// Números enteros
echo 1<=>1; // 0
echo 1<=>2; // -1
echo 2<=>1; // 1

// Numeros decimales
echo 1.5<=>1.5; // 0
echo 1.5<=>2.5; // -1
echo 2.5<=>1.5; // 1

// Cadenas de caracteres
echo "a"<=>"a"; // 0
echo "a"<=>"b"; // -1
echo "b"<=>"a"; // 1
```

El operador ?? devuelve el primer operando distinto de null.

```
$num = null;
$cadena = null;
//asigna el 13 ya que es primer operando distinto de null
$resultado = $num ??$cadena ??13;
```

Operadores de asignación

Ya se ha comentado el operador de asignación que sirve para dar valor a una variable (=).

```
$x=$y+9.8;
```

Sin embargo, existen operadores que permiten mezclar cálculos con la asignación, ejemplo:

```
$x+=3;
```

En el ejemplo anterior lo que se hace es sumar ***3*** a la ***x*** (es lo mismo ***$x+=3*** que ***$x=$x+3***). Eso se puede hacer también con todos estos operadores:

+=	-=	*=	/=
&=	|=	^=	%=
>>=	<<=	.=	

Otros operadores de asignación son "++" (incremento) y "--" (decremento). Ejemplo:

```
$x++; //esto es $x=$x+1;
$x--; //esto es $x=$x-1;
```

Existen dos formas de utilizar el incremento y el decremento. Se puede usar por ejemplo ***x***++ o ++***x***. La diferencia estriba en el modo en el que se comporta la asignación. Ejemplo:

```
$x=5;
$y=5;
$z=$x++; //z vale 5, x vale 6
$z=++$y; //z vale 6, y vale 6
```

Operadores de BIT

Manejan números, los cuales se pasan a formato binario y se opera con ellos BIT aBIT. Son:

operador	significado
&	AND
|	OR
~	NOT
^	XOR
>>	Desplazamiento a la derecha
<<	Desplazamiento a la izquierda

Concatenación

El punto (.) es un operador que permite unir textos. Su uso es muy sencillo. Existe también el operador de asignación y concatenación (.=)Ejemplo:

```
$x="Hola";
$x=$x." amigo"; //$x valdría Hola amigo

$x="Hola";
$x.=" amigo";  //$x valdría Hola amigo
```

4.- ESTRUCTURAS DE CONTROL

Hasta ahora las instrucciones que hemos visto, son instrucciones que se ejecutan secuencialmente; es decir, podemos saber lo que hace el programa leyendo las líneas de izquierda a derecha y de arriba abajo.

Las instrucciones de control de flujo permiten alterar esta forma de ejecución. A partir de ahora habrá líneas en el código que se ejecutarán o no dependiendo de una **condición**.

Esa condición se construye utilizando lo que se conoce como **expresión lógica**. Una expresión lógica es cualquier tipo de expresión que dé como resultado verdadero o falso.

Las expresiones lógicas se construyen a través de los operadores relacionales **(==,** >, <,…) y lógicos **(&&,**||, **!,…**) vistos anteriormente.

4.1.- Sentencia de bloque.

Aunque no es en sí una estructura de control se usa mucho dentro de ellas. Se indica mediante las llaves (**{}**). Es una sentencia que sirve para agrupar un conjunto de sentencias que se ejecutarán de forma secuencial y que se consideran como si fueran una sola.

```
{
  $x=10;
  $x=$x+12;
  echo $x;
}  //las tres sentencias se ejecutan secuencialmente
//y son consideradas como "una sola sentencia"
```

4.2.- Sentencia IF

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta otra sentencia en caso de que la expresión lógica sea verdadera. Si la expresión lógica se evalúa a false se puede indicar una sentencia alternativa.

Su sintaxis sería

```
if (expresión lógica)
            <sentencia si>
      [else
            <sentencia no>
      ]
```

Como se ve, la parte del ***else*** puede no indicarse. Para indicar más de una sentencia en cada caso se debe indicar un bloque ({}).

```
$edad=12;

//condicional con solo parte si
if ($edad>=18)
      echo "mayor de edad";

//condicional con solo parte si
```

```php
if ($edad>=18)
    {  //hay mas de una sentencia en caso de si
      echo "mayor de edad";
      $dinero=1000;
    }

//condicional con parte si y no
if ($edad>=18)
    {
      echo "mayor de edad";
      $dinero=1000;
    }
   else
    {
      echo "menor de edad";
      $dinero=0;
    }
```

Además, es posible anidar sentencias IF. Cada sentencia elseif permite añadir una condición que se examina si la anterior no es verdadera. En cuanto se cumpla una de las expresiones, se ejecuta el código del if o elseif correspondiente. El bloque else se ejecuta si no se cumple ninguna de las condiciones anteriores.

```php
if ($edad<14)
      echo "infantil";
  elseif ($edad<18)
      echo "juvenil";
     else
      echo "mayor de edad";
```

4.3.- Sentencia Switch

Esta instrucción se usa cuando tenemos sentencias que se ejecutan de forma diferente según evaluemos el conjunto de valores posible de una expresión. Cada **case** contiene un valor de la expresión; si efectivamente la expresión equivale a ese valor, se ejecutan las sentencias de ese **case** y de los siguientes.

```
switch (expresión) {
      case valor1:
            sentencias del valor 1
            [break]

      [case valor2:
            sentencias del valor 1
            [break]
      […]
      ]
      [default:
            sentencias que se ejecutan si la expresión no toma
            ninguno de los valores anteriores ..]
```

```
}
```

La instrucción **break** se utiliza para salir del **switch**. De tal modo que si queremos que para un determinado valor se ejecuten las instrucciones de un apartado **case** y sólo las de ese apartado, entonces habrá que finalizar ese **case** con un **break.**

Se tiene también una expresión similar a switch que es **match**. Tiene como sintaxis:

```
match (a_comprobar){
      valor1 => retorno1,
      valor2, valor3 => retorno2
};
```

La comprobación de igualdad se realiza con el operador ===, por lo que se valida también que el tipo sea el mismo. Esta expresión puede usarse para asignar un valor a una variable.

```
$valorBool = true;

$resultado=match ($valorBool){
  true=>"es verdad",
  false=>"es mentira"
};
```

4.4.- Bucles.

Un bucle es un conjunto de sentencias que se repiten mientras se cumpla una determinada condición. Los bucles agrupan instrucciones que se ejecutan continuamente hasta que una determinada condición se evalúa sea falsa.

4.4.1.- Bucle while

Su sintaxis:

```
While (expresión lógica)
      Sentencia
```

El programa se ejecuta siguiendo estos pasos:

1) Se evalúa la expresión lógica
2) Si la expresión es verdadera ejecuta las sentencias, sino el programa abandona la sentencia **while**
3) Tras ejecutar las sentencias, volvemos al paso 1)

En el siguiente ejemplo se muestran los números que van del 1 al 100.

```
$cont=1;
```

```php
while ($cont<=100)
{
      echo "$cont<br>".PHP_EOL;
      $cont++;
}
```

Este tipo de bucles son los fundamentales. Todas las demás instrucciones de bucle se basan en el bucle **while**.

En los programas que se realizan es muy normal usar bucles con un funcionamiento específico. Esto nos da lugar a los dos principales tipos de bucle: de centinela y de contador.

Bucles contador

Se llaman así a los bucles que se repiten una serie determinada de veces. Dichos bucles están controlados por un contador (o incluso más de uno). El contador es una variable que va variando su valor (de uno en uno, de dos en dos,… o como queramos) en cada vuelta del bucle. Cuando el contador alcanza un límite determinado, entonces el bucle termina.

En todos los bucles de contador necesitamos saber:

1) Lo que vale la variable contadora al principio. Antes de entrar en el bucle
2) Lo que varía (lo que se incrementa o decrementa) el contador en cada vuelta del bucle.
3) Las acciones a realizar en cada vuelta del bucle
4) El valor final del contador. En cuanto se rebase el bucle termina. Dicho valor se pone como condición del bucle, pero a la inversa; es decir, la condición mide el valor que tiene que tener el contador para que el bucle se repita y no para que termine.

```php
$cont=1; /* valor inicial del contador */
while ($cont<=100) /* condicion de bucle*/
{
    /* sentencias a realizar en la iteracion;*/
    echo "$cont<br>".PHP_EOL;

    $cont++; /* variación del contador*/
}
```

Bucles centinela

Es el segundo tipo de bucle básico. Una condición lógica llamada **centinela**, que puede ser desde una simple variable booleana hasta una expresión lógica más compleja, sirve para decidir si el bucle se repite o no. De modo que cuando la condición lógica se incumpla, el bucle termina.

Esa expresión lógica a cumplir es lo que se conoce como centinela y normalmente la suele realizar una variable booleana.

```php
$salir=false; /* variable usada como centinela */

while ($salir===false) /* mientras la centinela no valga true*/
```

```
{
    //sentencias a ejecutar
    $num=rand(1,500);
    echo($num);

    $salir=($num%7==0); /* evaluacion condición de centinela */
}
```

Comparando los bucles centinela con los de contador, podemos señalar estos puntos:

- Los bucles de contador se repiten un número concreto de veces, los bucles de centinela no
- Un bucle de contador podemos considerar que es seguro que finalice, el de centinela puede no finalizar si el centinela jamás varía su valor (aunque, si está bien programado, alguna vez lo alcanzará)
- Un bucle de contador está relacionado con la programación de algoritmos basados en series.

4.4.2.- Bucle Do…While

La única diferencia respecto al anterior está en que la expresión lógica se evalúa después de haber ejecutado las sentencias. Es decir, el bucle al menos se ejecuta una vez. Los pasos son:

1) Ejecutar sentencias
2) Evaluar expresión lógica
3) Si la expresión es verdadera volver al paso 1, sino continuar fuera del while

Su sintaxis es:

```
do{
    sentencia;
} while (expresion logica)
```

Ejemplo: bucle que muestra los números del 1 al 100

```
$cont=0;
do{
    $cont++;
    echo "$cont<br>".PHP_EOL;
} while ($cont<100);
```

Se utiliza cuando sabemos al menos que las sentencias del bucle se van a repetir una vez (en un bucle **while** puede que incluso no se ejecuten las sentencias que hay dentro del bucle si la condición fuera falsa, ya desde un inicio).

4.4.3.- Bucle for

Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador. Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Sintaxis:

Su sintaxis sería:

```
for(inicializacion;condicion;incremento)
      sentencia;
```

Las sentencias se ejecutan mientras la condición sea verdadera. Además, antes de entrar en el bucle se ejecuta la instrucción de inicialización y en cada vuelta se ejecuta el incremento. Es decir, el funcionamiento es:

1) Se ejecuta la instrucción de inicialización
2) Se comprueba la condición
3) Si la condición es cierta, entonces se ejecutan las sentencias. Si la condición es falsa, abandonamos el bloque ***for***
4) Tras ejecutar las sentencias, se ejecuta la instrucción de incremento y se vuelve al paso 2

Volviendo al ejemplo de mostrar los números del 1 al 100, con un bucle for quedaría

```
for($cont=1;$cont<=100;$cont++)
      echo"$cont<br>".PHP_EOL;
```

5.- FUNCIONES

Uno de los problemas habituales del programador ocurre cuando los programas alcanzan un tamaño considerable en cuanto a líneas de código. El problema se puede volver tan complejo que acaba siendo inabordable.

Para mitigar este problema apareció la **programación modular.** En ella el programa se divide en módulos de tamaño manejable. Cada módulo realiza una función muy concreta y así el programador se concentra en cada módulo y evita la complejidad de manejar el problema completo. Los módulos se pueden programar de forma independiente. Se basa en concentrar los esfuerzos en resolver problemas sencillos y una vez resueltos, el conjunto de las soluciones a esos problemas soluciona el problema original.

En definitiva, la programación modular implementa el paradigma ***divide y vencerás***, tan importante en la programación. El programa se descompone en módulos. Los módulos se pueden entender que son pequeños programas que reciben datos y a partir de ellos realizan un cálculo o una determinada tarea. Una vez el módulo es probado y validado se puede utilizar las veces que haga falta en el programa sin necesidad de tener que volver a programarlo; incluso se puede utilizar en diferentes programas ya que se pueden almacenar funciones en un mismo archivo formando lo que se conoce como **librería**.

En PHP la programación modular se implementa mediante funciones. Las funciones trabajan de esta manera:

- Las funciones poseen un nombre, un identificador que cumple las reglas indicadas para los demás identificadores que conocemos (como los de las variables). Pero, a diferencia de las variables, no utilizan el signo $. Se aconseja que el nombre de las funciones se escriba en minúsculas.
- A las funciones se les indica, aunque no a todas, unos valores (parámetros) que la función necesita para hacer su labor
- Las funciones pueden devolver un valor, resultado del trabajo de la misma.
- Las funciones contienen el código que permite realizar la tarea para la que se creó la función.
- Lo más importante es que las funciones son fundamentales en la programación de aplicaciones PHP porque permiten centrarse en un problema concreto y pequeño.

De hecho, PHP incorpora muchas funciones ya creadas para trabajar (como la propia **print** o la muy utilizada **echo**). Por ejemplo, este código:

```
echo rand(1,10);
```

Escribe un número aleatorio entre uno y diez. Los parámetros son el número uno y el diez, el valor que se devuelve es el número aleatorio entre esos dos números, el identificador es ***rand*** y al

ser una función estándar de PHP, no podemos ver su código.

Es importante documentar las funciones. Si utilizamos documentación tipo javadoc, los entornos de programación pueden mostrarnos ayuda cuando nos situemos la función.

Esta documentación se define escribiendo /** */ y utilizando palabras clave especiales.

Como ejemplo, podemos tener:

```
/**
 * Esta función se encarga de devolver la suma de lo parámetros
 *
 * @param integer $a operando primero
 * @param integer $b operando segundo
 * @return integer devuelve la suma de los operandos
 */
function suma(int$a, int$b):int
{
     //devuelve la suma
     return $a+$b;
}
```

5.1.- Declaración y uso de funciones.

Se pueden crear funciones mediante la siguiente declaración:

```
function funcion(parametros)
{
      sentencias;
      [return valor_retorno;]
}
```

Como se ve, se definen parámetros que son variables que se pasaran como argumentos y es posible indicar en el cuerpo de la función la sentencia return que nos permite devolver un valor.

Por ejemplo, podemos declarar la función doble. Esta función tiene un parámetro **$valor** y devuelve el doble de ese valor

```
function doble($valor)
{
      return 2*$valor;
}
```

Este código se puede declarar en cualquier parte de una página PHP, pero lo aconsejable es declarar las funciones en la cabecera.

Para utilizar una función simplemente hay que invocarla pasando los parámetros que requiere, por ejemplo:

```
$numero=12;
$valor=doble($numero);
echo "el doble de $numero vale $valor<br>".PHP_EOL;
```

Como se ha indicado, si no se quiere que la función devuelva ningún valor, puede omitirse la sentencia **return**.

5.2.- Alcance de las variables.

Las variables definidas en una función existen mientras existe la función, es decir, al finalizar la función se eliminan. Por lo tanto, su ámbito es local a la función. Ejemplo:

```
function f1()
{
      $var=10;
}

echo $var; //Error: $var no se ha declarado
```

La instrucción ***echo*** del ejemplo anterior, provoca un fallo de variable no definida, porque PHP no reconoce a la variable ***$var***, la única $var del código se crea en la función y sólo se puede usar en la función; hacer referencia a $var fuera dela función no tiene sentido; ya que tras el cierre de la llave en la que se definió, la variable muere. La variable es, en definitiva, **local** a la función.

Esto se puede observar también en las siguientes sentencias:

```
$var=5;
function f1()
{
      $var=10;
}

echo $var; //Muestra 5
```

Aunque declaro variables con el mismo nombre $var, son distintas la variable de dentro de la función (local) y la que se declara fuera de la función (global).

5.3.- Declaraciones de tipo.

Aunque PHP es un lenguaje de programación en el que no es necesario declarar el tipo de una variable, en las últimas versiones de PHP se ha incluido la declaración de tipos en tres ámbitos: argumentos de funciones, valores de retorno y propiedades de clase (se verá en el apartado de clases).

Estas declaraciones aseguran que el valor es del tipo especificado en el momento de la llamada, lanzando una excepción (de tipo TypeError) en caso de que los valores de llamada no sean del tipo correcto.

Al declarar un tipo se puede usar:

TIPO	DESCRIPCIÓN
Nombre de Clase/Interfaz	El valor debe ser una instancia de la clase o interfaz

Self	El valor debe ser una instanceof de la misma clase que aquella en la que se utiliza la declaración de tipo. Solamente se pueden usar en clases
Parent	El valor debe ser una instanceof del padre de la clase en la que se usa la declaración de tipo. Solo se puede usar en clases.
Array	El valor deber ser un array
Callable	El valor debe ser un callable válido. No puede ser usado como una declaración de tipo de propiedad de clase
Bool	El valor debe ser un booleano
Float	El valor debe ser un número de coma flotante
Int	El valor debe ser un número entero
String	El valor debe ser un string
Iterable	El valor debe ser un array o una instanceof de la clase Traversable
Object	El valor debe ser un objeto
Mixed	El valor puede ser cualquiera

No se permiten usar alias de los tipos anteriores (pej, boolean en vez de bool)

```
function prueba(boolean $parametro)
{
};

prueba(true);
```

Al ejecutarlo nos genera un warning y un error :

```
Warning: "boolean" will be interpreted as a class name. Did you mean
"bool"?
Fatal error: Uncaught TypeError: prueba(): Argument #1 ($parametro) must
be of type boolean, bool given, called in
```

Podemos usar la declaración de tipos en funciones (parámetros y valores de retorno)

```
//declaración de función con dos parámetros enteros
//y resultado (valor de retorno) entero
function suma(int $a, int $b): int
{
     return$a+$b;
}

echo "resultado de sumar ".suma(3,5);
```

O usando clases…

```
classA{};
classB extends A{};
classC{};

//funcion a la que se pasa un objeto de clase A
//o descendientes
function hacer(A $obj)
{
```

```
    echo get_class($obj);
}

//correcto
hacer(new A);

//correcto
hacer(new B);

//error C no hereda de A
hacer(new C);
```

Es posible marcar las declaraciones como **nullable**, es decir, que se puede asignar un valor null. Esto se consigue antecediento el tipo con el símbolo **?**.

```
function llamadaConNulos(?int $x)
{
   if (isset($x))
........echo"el valor es $x";
......else
........echo"el valor es nulo";
}

//llamada a la función con un valor
llamadaConNulos(25);

//llamada a la función con nulo
llamadaConNulos(null);
```

De igual forma si definimos un parametro con valor por defecto null, se considera que es nullable.

```
function funcionConParametros(int $par1, string $par2 = null, float
$par3 = null)
{
  echo $par1;
  if (isset($par2))
     echo $par2;
  if (isset($par3))
     echo $par3;
}
```

El pseudotipo callable (callback/llamada de retorno) se utiliza para definir parámetros que se indican en una función que garantizan que la variable puede ser llamada como una función.

```
//creo funciones
function dameSaludo():string
{
    return"hola";
}
```

```php
//defino la función con parametro callable
function funcionConLLamada(string$cadena, callable$funcion)
{
    echo $cadena." ".$funcion()."<br>";
}

//llamo a la funcion previamente
funcionConLLamada("El saludo es","dameSaludo");

//llamo a la función usando como parámetro una función anónima
funcionConLLamada("hoy es",function ():string
{
    return date('d/m/Y');
});
```

El pseudotipo Iterable se usa para indicar un Array u objeto que implemente la interfaz Traversable. Estos dos tipos se recorren mediente un foreach y se pueden emplear con yield from. Cuando se usa en la definición de un parámetro, indica que se requiere un conjunto de valores pero no importa la forma de dicho conjunto ya que se utilizará foreach para recorrerlo.

```php
function recorrer(iterable $x):int
{
    $y=0;
    foreach($x as $valor)
    {
       if (is_int($valor))
          $y+=$valor;
    }
    return $y;
}

$lista=[1,4,23];

$resultado=recorrer($lista);
```

También es posible combinar tipos simples en tipos compuestos de dos modos:

- Unión de tipos simples.
- Intersección de tipos de clases (interfaces y nombre de clase)

En el primer caso (**unión**), usamos el símbolo | para unir los tipos. Además de poder usar los tipos básicos podemos usar el tipo null (se considera que ?T es lo mismo que T | null)

```php
function concatenacion(string $cad, int|float $num)
{
    echo $cad." ".$num.PHP_EOL;
}

//correcto
```

```
concatenacion("el numero es", 25);

//incorrecto
concatenacion("el valor es ", "una cadena");
```

Un tipo especial de unión es **mixed** que se define como object | resource | array | string | int | float | bool | null.

Una declaración de tipo **Intersección** acepta valores que incluyen diferentes tipos de clase. Utiliza el símbolo **&** para indicarlo (se verá en las clases)

Además de los tipos ya indicados es posible utilizar el "pseudotipo" false en el retorno de una función. Por razones históricas, se devolvía false en las funciones cuando se encontraba algún error. False no puede usarse en ningún otro sitio, sólo en la declaración del tipo de retorno.

```
function doble(int $num): int|false
{
   //$num es menor de 10, consideramos que hay un error
   if ($num<10)
      return false;

   return2*$num;
}

$x=15;
if (($y=doble($x))===false)
    echo"hay error";
   else
    echo"el doble de $x es $y ";
```

Para indicar que la función no devuelve nada (no llama a return o si usa return no devuelve nada) se usa el tipo de retorno **void**.

```
function sinRetorno(): void
{
   //hago algo en la función

   //no retorno nada
   return ; //o simplemente sin return
}

sinRetorno();
```

Por último, en las clases, se puede usar el tipo de retorno **static**, que indica que el valor devuelto debe ser una instancia de la misma clase en la que se llama al método.

5.4.- Paso de parámetros por referencia.

Por defecto las funciones reciben los parámetros por valor, esto significa que se recibe una copia del valor en la función. Por lo que examinando este código:

```
function f1($var1)
{
      $var1=10;
}

$var=5;
f1($var);
echo $var; //Muestra 5
```

En este caso, se pasa a la función f1, la variable $var. Dentro de la función se modifica su valor por 10 (internamente se usa el parámetro $var1). Al finalizar la función, $var mantiene su valor al haber sido pasada a la función por valor.

Sin embargo, a veces sí se desea que las funciones cambien el valor de las variables que se pasan como parámetro. El ejemplo más claro es el de la función ***swap***. La función swap sirve para intercambiar los valores de dos variables.

Se desea que está instrucción: ***swap($x,$y),*** sirva para intercambiar los valores de *x* e *y* por ello se programa de esta forma:

```
function swap(int $x,int $y){
     $aux=$x;
     $x=$y;
     $y=$aux;
}

$valor1=12;
$valor2=17;
swap($valor1,$valor2);

echo$valor1." ".$valor2;
```

En apariencia swap funciona, pero realmente no modifica los valores de las variables *$valor1* y *$valor2* puesto que los parámetros (*$x* y *$y*) reciben una copia de losvalores. El problema es que esta función recibe los parámetros por valor, luego no modifica las variables originales. Por ello, *echo* escribiría *12 17*.

Cuando los parámetros se usan por referencia, entonces sí se cambian las variables.Para ello basta con preceder el signo **&** a los parámetros que se usarán por referencia.

Así *swap*se programaría así:

```
function swap(int &$x,int &$y){
     $aux=$x;
     $x=$y;
     $y=$aux;
}
```

```
$valor1=12;
$valor2=17;
swap ($valor1,$valor2);

echo $valor1." ".$valor2;
```

Escribe ***17 12***, las variables originales han sido modificadas.

Cuando una función utiliza paso por referencia, entonces al invocarla debe recibir nombres de variables y no expresiones o valores literales. Es decir, este código:

```
swap(12,17);
```

Produce un error, porque la función espera variables y no expresiones.

5.5.- Parámetros predefinidos.

Se puede indicar un valor por defecto a los parámetros a fin de permitir que dicho parámetro sea opcional; de modo que, si se pasa el parámetro se toma el valor que se pasa y si no, se toma el valor por defecto. Ejemplo:

```
function potencia(int $base,int $exponente=2):int
{
     $valor=1;
     for ($cont=1;$cont<=$exponente;$cont++)
          $valor*=$base;

     return $valor;
}

$res=potencia(5,3); //$res=5^3
$res=potencia(5);//$res=5^2
```

En este caso se tiene la llamada a la función por posición. Una vez indicado un parámetro por defecto, todos los siguientes que se definan deben ser establecidos como por defecto.

Puede llamarse también a las funciones pasando los **parámetros por nombre**. En este caso, se puede indicar cualquier orden en la llamada e incluso llamar a un parámetro predefinido cualquiera de los existentes. Para llamar a un parametro por nombre se usa la sintaxis parametro:valor

```
function funcionConParametros(int $par1, string $par2=null, float
                              $par3=null)
{
   echo $par1;
```

```
   if (isset($par2))
      echo $par2;

   if (isset($par3))
      echo$par3;
}

//llamamos a la función pasando los parámetros por
//posicion
funcionConParametros(12,"correcto");

//llamamos a la función usando la llamada por nombre
//para poder indicar el parámetro por defecto que me interese
funcionConParametros(12,par3:2.7);
```

5.6.- Variables globales.

Cuando definimos variables en un script de PHP, se consideran como globales es decir accesibles en todo el código del mismo salvo en las funciones. Por lo tanto una variable en una función se considera local y sería distinta de una global que se llame igual.

Si queremos referirnos a una variable global dentro de una función se utiliza la palabra ***global.***

El usar GLOBAL es una **práctica mala y no se aconseja**. La programación modular busca crear bloques lo más independientes posibles que chocan con el uso de esta funcionalidad.

```
$exponente=5;

function potencia(int $base):int
{
     global $exponente; //variable global, no aconsejado

     $valor=1;
     for ($cont=1;$cont<=$exponente;$cont++)
          $valor*=$base;

     return $valor;
}

$res=potencia(5);   //$res=5^5
$res=potencia(7);   //$res=7^5
```

Esto es una mala solución ya que no podemos llevar la función potencia a otra aplicación al obligar a tener definida la variable $exponente .

5.7.- Variables estáticas.

Se trata de variables locales a la función (sólo se podrán utilizar dentro de la función), pero son variables que recuerdan su valor entre llamadas. Por ejemplo:

```
function estatica():int{
     static $cuenta=0;

     $cuenta++;

     return $cuenta;
}

for($i=1;$i<=10;$i++){
     echo "Esta es la llamada número ".estatica()." <br />".PHP_EOL;
}
```

La variable $cuenta se inicializa a 0 la primera vez que se llama a la función estatica (en llamadas sucesivas no se inicializa). En llamadas posteriores recuerda su valor. El resultado sería:

```
Esta es la llamada número 1
Esta es la llamada número 2
Esta es la llamada número 3
Esta es la llamada número 4
Esta es la llamada número 5
Esta es la llamada número 6
Esta es la llamada número 7
Esta es la llamada número 8
Esta es la llamada número 9
Esta es la llamada número 10
```

5.8.- Listas de argumentos variables.

Una característica de PHP es la posibilidad de definir en una función un número variable de parámetros en la llamada, es decir, que al llamar a la función se le puedan indicar 2, 5, 10, etc parámetros.

Existen dos mecanismos para gestionar la lista de argumentos variables.

Token …

Está disponible desde PHP 5.6. Las funciones usan el ***token …*** para indicar que la función acepta un número variable de argumentos. Los argumentos serán pasados a la variable dada como un array, por ejemplo:

```
function suma(float $a, float $b, float ...$resto):float
{
     $res=$a+$b;

     foreach($resto as $num)
          $res+=$num;

     return$res;
}
```

```
echo suma(5,6);  //resultado 11
echo suma(5,6,2,4,9,1,3);   //resultado 30
```

En este caso hemos definido la función suma con dos argumentos obligatorios ($a y $b). Además, se le puede indicar en la llamada cualquier número de argumentos adicional que se almacenaran como array en el argumento $resto. Así la hemos podido llamar con 2 o con 7 argumentos.

Versiones antiguas

En versiones anteriores a PHP 5.6 se utilizan las funciones ***func_num_args(), func_get_arg() y func_get_args()*** para el acceso a la lista de argumentos variables. Devuelven respectivamente el número de argumentos pasados a la función, el valor para el argumento x y todos los argumentos en forma de array. Así el ejemplo anterior quedaría como

```
function suma(float $a,float $b):float
{
     $res=0;

     foreach(func_get_args() as $num)
          $res+=$num;

     return $res;
}


echo suma(5,6);//resultado 11
echo suma(5,6,2,4,9,1,3);   //resultado 30
```

Usamos la función ***func_get_args*** para devolver todos los argumentos pasados a la función. Al definir dos argumentos en la función, obligo a que se llame con al menos dos argumentos.

5.9.- Funciones variables.

PHP admite el concepto de funciones variables. Esto significa que si un nombre de variable tiene paréntesis anexos a él, PHP buscará una función con el mismo nombre que lo evaluado por la variable, e intentará ejecutarla. Entre otras cosas, esto se puede usar para implementar llamadas de retorno, tablas de funciones, y así sucesivamente.

```
function escribe(string $texto)
{
     return $texto;
}

function error(string $err)
```

```
{
     return "el error producido es $err";
}

$var="escribe";
echo $var("correcto"); //hace la llamada escribe("correcto")

$var="error";
echo $var("no se puede"); //hace la llamada error("no se puede")
```

5.10.- Funciones anónimas y funciones de flecha

Las funciones anónimas, también conocidas como *cierres* (closures), permiten la creación de funciones que no tienen un nombre especificado. Son más útiles como valor de los parámetros de llamadas de retorno, pero tienen muchos otros usos.

Las funciones anónimas están implementadas utilizando la clase Closure.

Por ejemplo, podríamos tener:

```
//definición de una funcion anonima
$operacion=function (int $a):int{
     return $a^3;
};

//llamada a la funcion anonima
echo $operacion(5); //muestra 5^3
```

En este caso he definido una función anónima (sin nombre) asignándola a una variable. Posteriormente he usado el mecanismo de función variable para ejecutarla.

```
//creacion de una función con un argumento callable
functionhacer(int$a, callable $oper):int
{
     return $oper($a);
}

//llamada a la función hacer pasando como argumento
//a $oper una función anónima
echo hacer(2, function(int$x):int{
                    return$x+20;
               });//escribiría 22 => 2 + 20
```

La función hacer tiene dos argumentos: $a y $oper. $oper es un argumento callable por lo que al llamarlo se le debe indicar el nombre de una función como cadena o pasarle una función anónima. Posteriormente he llamado a la función hacer con una función anónima como argumento que es lo que se espera. Esta función anónima se llama internamente en hacer pasándole el argumento $a.

Los cierres pueden heredar variables del ámbito padre. Esto se indica con la palabra **use**. Las variables tendrán el valor en el momento en el que se declaró la función.

```php
$numero=15;

//defino la función de flecha
$funcion= function(int $otro) use ($numero){
    return $otro+$numero;
};

//$var vale 10+15
$var=$funcion(10);

$numero=1;

//$var sigue valiendo 10+15
$var=$funcion(10);
```

Para simplificar todavía más las funciones anónimas, han aparecido las **funciones de flecha,** que representan una función en la que se devuelve el resultado de una expresión.

La sintaxis es

```
fn(lista de argumentos) => expresión
```

Las funciones flecha soportan las mismas características que las funciones anónimas, excepto que el uso de variables del ámbito padre siempre es automático. Cuando una variable utilizada se define en el ámbito padre será implícitamente capturada por valor.

```php
$numero=15;

//defino la función de flecha
$funcion= fn(int $otro):int => $otro+$numero;

//llamo a la función igual que a las funciones anónimas
$var=$funcion(10);

//igual que en las funciones anónimas
//$numero tiene el valor en la función
//del momento de la declaración
$numero=1;
$var=$funcion(10);
```

Esto también funciona si las funciones flechas están anidadas

```php
$numero=15;

//defino la función de flecha
$funcion= fn($otro) =>fn($yOtra) => $otro+$numero+$yOtra;

//$otro = 10, $yOtro = 2
$var=$funcion(10)(2);
```

5.11.- Ámbito de las funciones.

Las funciones en PHP siempre son globales. Es decir, una función se puede utilizar en cualquier parte del código PHP. Por supuesto esto prohíbe poner el mismo nombre a dos funciones.

5.12.- Inclusión de funciones.

Dentro del código PHP se puede hacer uso de las instrucciones ***include*** y/o ***require*** para incluir el código de otro archivo. El archivo puede ser de cualquier tipo: html, php u otras extensiones. Por lo que permite la creación de archivos que contengan librerías de funciones o código reutilizable en múltiples páginas.

Tanto include como require lo que hacen es simplemente copiar y pegar el código del archivo tal cual. La diferencia es que si el archivo no existe (o no se encuentra), include seguirá ejecutando el código (aunque normalmente mostrará una advertencia del error), mientras que require generará un error grave y parará la ejecución del código.

Si creamos una librería con funciones y la incluimos más de una vez, PHP nos generará un error al definir la función varias veces. Para solucionarlo existe include_once y require_once que garantizan que solo se haga la inclusión una vez.

En la inclusión se entiende que el código que se incluye es código HTML, por lo que se entenderá que lo que no esté encerrado en etiquetas **<?php** y **?>** es código HTML.

Ejemplo de uso:

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

5.13.- Espacios de nombres

En algunas ocasiones, nos puede interesar tener diversas implementaciones de funciones que queremos nombrar igual, de forma que llamemos posteriormente a la implementación que nos interese. En PHP, no se permite incluir dos librerías que contuvieran funciones con el mismo nombre. Para solucionar esto, PHP incluye el concepto de los ***Espacios de nombres*** queno son más que formas de encapsular elementos.

Los espacios de nombres siguen el mismo principio de los directorios de los sistemas operativos. Los directorios sirven para agrupar ficheros relacionados, actuando como espacios de nombres para los ficheros que contienen. Como ejemplo, el fichero *foo.txt* puede existir en los directorios */home/uno* y */home/otro*, pero no pueden coexistir dos copias de *foo.txt* en el mismo directorio. Además, para acceder al fichero *foo.txt* fuera del directorio */home/uno*, se debe anteponer el nombre del directorio al nombre del fichero, empleando el separador de directorios para así obtener */home/uno/foo.txt*.

Los espacios de nombres están diseñados para solucionar dos problemas con los que se encuentran los autores de bibliotecas y de aplicaciones al crear elementos de código reusable, tales como clases o funciones:

1. El conflicto de nombres entre el código que se crea y las clases/funciones/constantes internas de PHP o las clases/funciones/constantes de terceros.
2. La capacidad de apodar (o abreviar) Nombres_Extra_Largos diseñada para aliviar el primer problema, mejorando la legibilidad del código fuente.

Los espacios de nombres de PHP proporcionan una manera para agrupar clases, interfaces, funciones y constantes relacionadas.

Aunque cualquier código de PHP válido puede estar contenido dentro de un espacio de nombres, solamente se ven afectados por espacios de nombres los siguientes tipos de código: clases, interfaces, funciones y constantes.

Los espacios de nombres se declaran utilizando la palabra reservada *namespace*. El nombre debe comenzar por una letra. Un fichero que contenga un espacio de nombres debe declararlo al inicio del mismo, antes que cualquier otro código. (no puede haber antes ningún código html ni siquiera espacios en blanco)

```
<?php
namespace nombre;
```

Podríamos tener el siguiente código:

Fichero librería.php

```
<?php
namespace MiLibreria;

function f1(int $x,int $y):int
{
   return $x*$y;
}

const PI=3.14;
```

Fichero index.php

```
<?php
include "libreria.php";

echo \MiLibreria\f1(12,14).PHP_EOL;

echo "el valor de PI es ".\MiLibreria\PI.PHP_EOL;
```

Como se ve, definimos una librería con espacio de nombres MiLibrería dentro de la cual definimos la función f1 y la constante PI.

Desde el fichero index.php llamamos a la función y a la constante mediante la cualificación absoluta \MiLibreria\f1 y \MiLibreria\PI.

De esta forma cuando nombramos un elemento podemos:

- No indicar nada: En este caso el elemento debe estar en el espacio de nombres actual o ser global (no definido dentro de ningún espacio de nombres)

```
f1(12,24);
```

f1 es una función definida en el espacio de nombres actual o es una función global.

- Indicar una ruta relativa: La ruta indicada debe ser la de un subespacio de nombres dentro del espacio de nombres actual.

```
subespacio\f1(12,24);
```

f1 está definida en el subespacio de nombres SUBESPACIO que está dentro del espacio de nombres actual

- Indicar una ruta absoluta: Se indica una ruta totalmente cualificada para el elemento

```
\espacio\f1(12,24)
```

f1 está definida dentro del espacio de nombres ESPACIO

Hay que indicar que para acceder a cualquier clase, función o constante global se puede usar el nombre cualificado. Por ejemplo, para acceder a *strlen()* se puede indicar *\strlen()*.

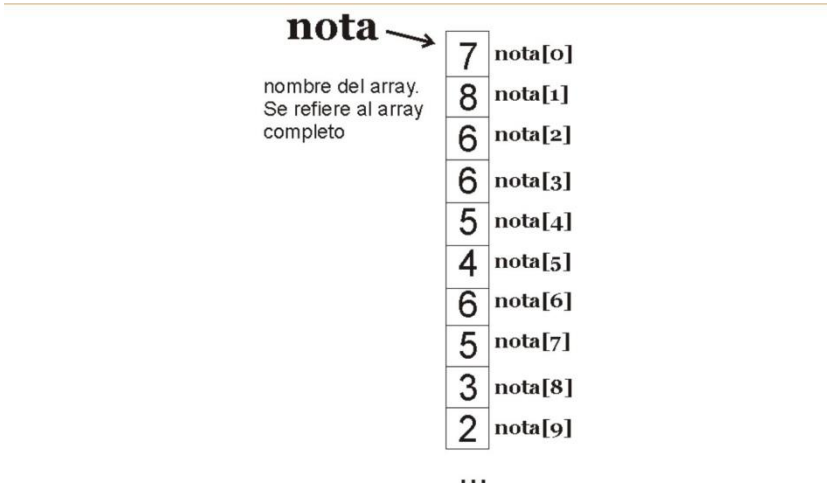
6.- ARRAYS

Los tipos de datos que conocemos hasta ahora no permiten solucionar problemas que requieren gestionar muchos datos a la vez. Por ejemplo, imaginemos que deseamos almacenar las notas de una clase de 25 alumnos, no habrá más remedio que declarar 25 variables.

Eso es tremendamente pesado de programar. Manejar esos datos significaría estar continuamente manejando 25 variables. Por supuesto si necesitamos 2000 notas, el problema se hace inmanejable.

Por ello en casi todos los lenguajes se pueden agrupar una serie de variables del mismo tipo en una misma estructura que comúnmente se conoce como array. Esa estructura permite referirnos a todos los elementos, pero también nos permite acceder individualmente a cada elemento.

Los arrays son una colección de datos del mismo tipo al que se le pone un nombre (por ejemplo ***nota***). Para acceder a un dato individual de la colección hay que utilizar su posición. La posición es un número entero que se llama **índice**, así para acceder a la quinta nota se pondría ***nota[4]***. Hay que tener en cuenta que en los arrays el primer elemento tiene como índice el número cero. El segundo el uno y así sucesivamente.



Esta definición de arrays es la común en todos los lenguajes clásicos. En el caso de PHP, los arrays son elementos más complejos, puesto que admiten datos de distinto tipo. La realidad es que los arrays de PHP son en realidad elementos que asocian un valor y una clave (arrays asociativos) como se explica más adelante.

6.1.- Arrays escalares.

Los arrays escalares cumplen la definición indicada anteriormente de array: es un conjunto de valores a los que se accede a través de un índice que es un número entero. Así, por ejemplo, las notas se podrían almacenar de esta forma:

```
$nota[0]=5;
$nota[1]=9;
$nota[2]=8;
$nota[3]=5;
$nota[4]=6;
$nota[5]=7;
```

Acceder a una nota cualquiera, requiere conocer su índice.

No hace falta declarar el array (siguiendo el estilo habitual de PHP), simplemente se utiliza. Es posible incluso este código:

```
$valor[0] =18;
$valor[1]="Hola";
$valor[2]=true;
$valor[3]=3.4;
```

Como se ve, los valores de un array pueden ser heterogéneos.

Al no declarar el array, la posición pasa a existir en el momento en el que se hace la asignación. Por ejemplo la posición 3 no existía hasta que no se realizó la asignación $valor[3].

Para la asignación de nuevas posiciones se puede usar directamente ***[]***, que rellena la posición escalar siguiente a la última existente.

```
$valor[0]=25;       //posicion 0
$valor[5]='hola';   //posicion 5
$valor[]=true;      //posicion 6, ultima posicion+1
```

La gracia de los arrays en todos los lenguajes es la facilidad de rellenar o utilizar su contenido mediante bucles **for**. Imaginemos que necesitamos almacenar 1000 números aleatorios del 1 al 10; evidentemente sería terrible tener que escribir 1000 líneas de código para realizar esta labor:

```
$valor[0]=rand(1,10);
$valor[1]=rand(1,10);
$valor[2]=rand(1,10);
//¡¡¡¡mucho trabajo!!!!
```

Lo lógico es usar este código:

```
for($i=0;$i<1000;$i++)
    $valor[$i]=rand(1,10);
```

La variable $i sirve para recorrer cada elemento del array. Va cambiando de 0 a 999 y así en dos líneas se rellena el array.

Puesto que en principio no sabemos cuántos elementos puede tener un array, existe la función ***count***, que nos resuelve este problema.

Por ejemplo, si deseamos recorrer un array que ya tiene valores para, por ejemplo, mostrar cada elemento en pantalla, el código sería:

```
for($i=0;$i<count($valor);$i++){
      echo $valor[$i]."<br />";
}
```

Un problema que pueden tener los arrays en PHP es la existencia de posiciones vacías.

```
$nota[0]=5;
$nota[1]=9;
```

```
$nota[3]=5;
$nota[4]=6;
$nota[5]=7;
```

En el código anterior la posición $nota[2] no está definida. Por lo tanto la instrucción echo $nota[2], nos mostraría un mensaje como "*Notice: Undefined offset: 2*" ya que no existe la posición 2.

Este problema se da también cuando usamos la función count. Esta función devuelve el tamaño del array sin tener en cuenta los elementos vacíos; es decir, sólo cuenta los elementos definidos. Por todo ello el código anterior:

```
for($i=0;$i<count($valor);$i++){
      echo $valor[$i]."<br />";
}
```

Puede fallar si el array $valor tiene elementos vacíos. Además, los últimos elementos no saldrían por que el contador no llega a ellos al contar elementos y no los índices más altos.

Para evitar eso podemos utilizar la función **isset** que devuelve verdadero cuando a la variable que se le pasa como parámetro se le ha asignado ya un valor. El bucle quedaría entonces:

```
$tope=count($nota);
for($i=0;$i<$tope;$i++){
      if(isset($nota[$i]))
          echo $nota[$i]."<br />";
         else
          $tope++;
}
```

El código es más enrevesado. Ahora sólo se escribe cada elemento del array si la función isset nos asegura que dicho elemento tiene valor; la variable $tope va incrementando su valor a fin de alcanzar al último elemento real del array.

6.2.- Arrays asociativos.

En este caso el índice es un texto que se comporta como una especie de clave que permite acceder al dato. Ejemplo:

```
$notas["antonio"]=7;
$notas["ana"]=9;
$notas["pedro"]=5;
```

La ventaja de estos arrays es que son más legibles, la desventaja es que no permiten su manejo mediante los bucles clásicos. Permiten de forma muy fácil la implementación de estructuras de datos más complejas como los conjuntos, hash, etc.

En un mismo array pueden coexistir tanto posiciones indexadas como asociativas.

```
$notas["antonio"]=7;
$notas["ana"]=9;
$notas["pedro"]=5;
$notas[]=3; //notas[0]
$notas[3]=6.5;
```

6.3.- Inicialización de arrays.

Para inicializar arrays se puede usar la función ***array()*** o ***[]***. Esto nos permite crear un array de una forma más compacta y fácil.

```
//definición del array $valor
$valor[0]=100;
$valor[1]=25;
$valor[2]='clase';
$valor[3]=false;

//definición del array $valor usando array()
$valor=array(100,25,'clase',false);

//definición del array $valor usando []
$valor=[100,25,'clase',false];
```

Como se ve tanto array() como [] nos va rellenando las posiciones empezando por 0.

Es posible indicar la posición a rellenar usando =>. Los índices que no se indican siguen el orden natural

```
//definición del array $valor
$valor[0]=100;
$valor[2]=25;
$valor[]='clase'; //siguiente posicion $valor[3]
$valor[7]=false;

//definición del array $valor usando array()
$valor=array(100,2=>25,'clase',7=>false);

//definición del array $valor usando []
$valor=[100,2=>25,'clase',7=>false];
```

Como se ha dicho en un array pueden coexistir tanto posiciones indexadas como asociativas. Tanto array() como [] permiten definiciones de este tipo:

```
//definición del array $valor
$valor[0]=100;
$valor[2]=25;
$valor["final"]=1284;   //posicion asociativa final
$valor[]='clase'; //siguiente posicion $valor[3]
```

```php
$valor[7]=false;

//definición del array $valor usando array()
$valor=array(100,2=>25,"final"=>1284, 'clase',7=>false);

//definición del array $valor usando []
$valor=[100,2=>25, "final"=>1284, 'clase',7=>false];
```

En PHP es muy fácil la creación de arrays multidimensionales.

Por ejemplo, quiero crear la matriz $\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{vmatrix}$. En PHP haría:

```php
$matriz[0][0]=1;
$matriz[0][1]=2;
$matriz[0][3]=3;
$matriz[1][0]=4;
$matriz[1][1]=5;
$matriz[1][2]=6;

$matriz=array(array(1,2,3),
              array(4,5,6)
         );

$matriz=[[1,2,3],
         [4,5,6]
        ];
```

Posteriormente podría acceder al elemento 1,2 como `$matriz[1][2]`.

Podríamos hacer un array multidimensional asociativo. Por ejemplo

```php
$poblacion["Alemania"]["Berlin"]=4000000;
$poblacion["Alemania"]["Hanover"]=1200000;
$poblacion["Francia"]["Paris"]=7400000;
$poblacion["Francia"]["Lyon"]=2300000;
$poblacion["España"]["Palencia"]=80000;

$poblacion=array("Alemania"=>array("Berlin"=>4000000,
                                   "Hanover"=>1200000),
                 "Francia"=>array("Paris"=>7400000,
                                   "Lyon"=>2300000),
                 "España"=>array("Palencia"=>80000)
);

$poblacion=["Alemania"=>["Berlin"=>4000000,
                         "Hanover"=>1200000],
            "Francia"=>["Paris"=>7400000,
                         "Lyon"=>2300000],
            "España"=>["Palencia"=>80000]
];
```

Al poder ser el contenido de cada elemento distinto, es posible crear matrices no regulares (no se puede hacer en otros lenguajes de programación).

```
$matrizIrregular=[[1],
                  [2,2],
                  [3,3,3],
                  [4,4,4,4]
                 ];
```

Este array tiene 4 filas, teniendo cada fila distinto número de columnas.

6.4.- Bucle foreach.

Recorrer arrays asociativos o incluso escalares con elementos sin inicializar resulta muy complejo con los bucles while o for. Por ello PHP dispone de un bucle muy potente que permite recorrer todos los elementos de un array sin importar si hay saltos en la numeración o si se trata de un array asociativo. Se trata del bucle foreach que tiene esta sintaxis:

```
foreach($array as $indice=>$valor)
{
      //sentencias
}
```

$array es el nombre del array que se va a recorrer; $índice es la variable que recogerá el índice de cada elemento a medida que se recorra el array y $valor es la variable que irá recogiendo el valor de cada elemento del array. Puede omitirse $indice =>, por lo que sólo se tendría el valor en cada iteración.

Ejemplo:

```
$notas=array(3,2=>7,"ana"=>9, 4,7=>10);

foreach($notas as $indice=>$valor)
{
      echo "Notas de $indice= $valor<br>";
}
```

Daría como salida:

```
Notas de 0= 3
Notas de 2= 7
Notas de ana= 9
Notas de 3= 4
Notas de 7= 10
```

6.5.- Recorrido de arrays mediante funciones.

Se trata de funciones basadas en el manejo de punteros de los lenguajes tradicionales. Utilizan un puntero virtual, que sería un objeto que señala a uno de los elementos del array y que al moverlo permite acceder al resto de elementos.

Esta técnica se basa en el manejo de las siguientes funciones:

función	uso
current(array)	Devuelve el valor del elemento al que actualmente señala el puntero. Si no hay ningún elemento, devuelve **false**
key(array)	Devuelve la clave a la que señala el puntero
next(array)	Mueve el puntero al siguiente elemento del array y devuelve su valor. Si el elemento actual es el último, **next** devuelve **false**.
prev(array)	Mueve el puntero al elemento anterior del array y devuelve su valor. Si el elemento actual es el primero, **prev** devuelve **false**.
reset(array)	Coloca el puntero en el primer elemento del array ydevuelve su valor. Si no hay ningún elemento en el array, devuelve **false**.
end(array)	Coloca el puntero en el último elemento y devuelve su valor.

Por ejemplo, podríamos tener:

```
$capital=array("Castilla y León"=>"Valladolid",
                  "Asturias"=>"Oviedo",
                  "Aragón"=>"Zaragoza");

while(key($capital)!= NULL){
      echo current($capital)."<br />";
      next($capital);
}
```

O recorriéndolo al revés:

```
$capital=array("Castilla y León"=>"Valladolid",
                  "Asturias"=>"Oviedo",
                  "Aragón"=>"Zaragoza");

end($capital);
while(key($capital)!= NULL){
      echo current($capital)."<br />";
      prev($capital);
}
```

6.6.- Funciones y arrays

Las funciones pueden recibir y devolver arrays al igual que cualquier otro tipo de variables.

A diferencia de la mayoría de lenguajes estructurados, en PHP los arrays se pasan por valor a las funciones. Es decir, cuando se pasa un array a una función, ésta recibe una copia del mismo. Así las modificaciones al array que se hagan dentro de la función, no afectan al array original.

Ejemplo:

```
function prueba($a){
      $a[0]=18;
```

```
}

$array=array(1,2,3,4,5,6,7);
prueba($array);
echo $array[0];
```

La función ***prueba*** modifica el elemento con índice cero en el array para darle el valor 18

Sin embargo al ejecutar el código PHP anterior, veremos el número *1* en pantalla. La modificación de la función prueba se ha hecho con el array *$a* que es una copia en realidad del original *$array*.

Si deseáramos que realmente la función modifique el array, necesitamos pasar el array por referencia, como ya se ha visto, basta con indicar el símbolo & delante del parámetro (o parámetros) que deseamos pasar por referencia:

```
function prueba(&$a){
      $a[0]=18;
}

$array=array(1,2,3,4,5,6,7);
prueba($array);
echo $array[0];
```

En este caso, si se escribe 18.

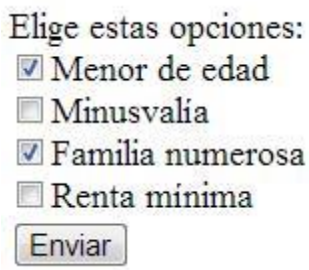
6.7.- Uso de arrays en formularios

Una de las virtudes de un array, es su capacidad de recoger en una sola variable, valores procedentes de diferentes controles en un formulario. La forma es muy sencilla, basta con indicar un nombre de array como atributo name en los elementos del formulario; para indicar que ese nombre es de un array, se añaden la apertura y cierre de corchetes **[** y **]**. Ejemplo:

```
<!DOCTYPE html>
<html lang="es-ES">
<head>
<meta charset="UTF-8">
      <title></title>
</head>
<body>
<form action="formArrayget.php" method="GET">
      Elige estas opciones:<br />
      <input type="checkbox" name="opciones[]" value="menor" />
Menor de edad <br />
      <input type="checkbox" name="opciones[]" value="minusvalia" />
        Minusvalía<br />
      <input type="checkbox" name="opciones[]" value="numerosa" />
      Familia numerosa <br />
      <input type="checkbox" name="opciones[]" value="minima" />
Renta mínima<br />
```

```
        <input type="submit" value="Enviar" />
</form>
</body>
</html>
```

La página resultante es (suponiendo que hemos marcado después algunas opciones):



En esta página, todos los checkbox están asociados a un array llamado ***opciones***. Dicho array contendrá un valor por cada elemento al que le hayamos hecho clic. Los índices se indican de forma escalar, es decir el primer valor (en el orden de escritura de la página web) será el cero, luego el uno,…

De modo que si el código de la página que recoge el array (***formArrayget.php***) es:

```
<!DOCTYPE html>
<html lang="es-ES">
<head>
<meta charset="UTF-8">
        <title></title>
</head>
<body>
<?php
        print_r($_GET["opciones"]);
?>
</body>
</html>
```

La salida de dicha página tras recoger los valores del formulario anteriormente comentados es:

```
Array ( [0] => menor [1] => numerosa )
```

Indicando que el array ***opciones*** (presente en el array de recogida del formulario **$_GET**) tiene dos elementos (con índices 0 y 1) con las valores ***menor*** y ***numerosa*** (correspondientes a los valores de los controles checkbox del formulario).

6.8.- Funciones estándar de uso en arrays.

Funciones básicas

función	significado
count($array)	Número de elementos del array
print_r($array)	Escribe el contenido del array (tanto valores como índices)

is_array(array)	Devuelve verdadero si el parámetro que recibe es un array

Ordenación de arrays

función	significado
sort(array,**[**flags**]**)	Ordena el array indicado. Si no se indica el segundo parámetro, ordena de forma normal sin convertir los tipos de datos. El segundo parámetro permite establecer la forma de ordenación pudiendo indicar estas posibilidades: • **SORT_REGULAR.** Ordenación normal (utilizada por defecto), se adapta según el valor sea numérico o String y no usa ordenación basado en alfabetos distintos del inglés. • **SORT_NUMERIC**. Ordena entendiendo que el array contiene valores numéricos • **SORT_STRING**. Ordena entendiendo que el array contiene valores String (texto) • **SORT_LOCALE_STRING**. Ordena basándose en la configuración regional establecida con la función **setlocale**. De modo que setlocale(LC_ALL,"es_ES") establecería como configuración regional, español de España. • **SORT_NATURAL**. Ordena el array considerando sus valores como lo haría un ser humano (es decir el texto ***"texto2"*** iría delante de ***"texto10"*** porque entendería el diez como un número superior al dos, y no ordenaría como si fueran textos alfabéticos. El funcionamiento es el mismo que la función **natsort** •**SORT_FLAG_CASE**Permite combinar (usando el operador **OR** (|) a nivel de bits) con **SORT_STRING** o **SORT_NATURAL** para ordenar cadenas de forma insensible a mayúsculas/minúsculas. Por ejemplo: sort($array,SORT_STRING | SORT_NATURAL) Al ordenar de esta forma, los índices desaparecen, el array final es un array escalar (el primer índice será el cero, luego el uno, etc.)
rsort(array,**[**flags**]**)	Igual que la anterior, pero ordena en orden descendente.
asort(array,**[**flags**]**)	Idéntica a **sort**, pero los índices no se desprecian y se respetan. Es decir se ordenan los valores y los índices se recolocan junto a la posición de su valor correspondiente
arsort(array,**[**flags**]**)	Como la anterior pero el orden es descendente
ksort(array,**[**flags**]**)	Idéntica a **sort**, pero se ordenan las claves. La relación de la clave junto con su valor correspondiente se respeta.
krsort(array,**[**flags**]**)	Como la anterior pero el orden es descendente
usort(array, funcionUsuario)	Ordena el array utilizando una función de usuario. Dicha función se debe crear de forma que reciba dos parámetros y devuelva un número mayor de cero cuando el primerparámetro sea mayor que cero, cero cuando sean iguales y un número menor que cero cuando el segundo parámetro sea mayor que el primero. **usort**recibe el nombre de la función entre comillas. Ejemplo: usort($array,"funcion1") Se supone que la ***funcion1*** ha sido creada previamente y cumple los requisitos indicados anteriormente.Esto permite crear ordenaciones personales con los arrays. Las claves se pierden (al igual que ocurre con la función **sort**) y sólo se ordenan los valores, resultando un array escalar.

uasort(array, funcionUsuario)	Igual que la anterior, pero se respeta la relación de las claves con sus valores.
uksort(array, funcionUsuario)	Igual que la anterior, pero lo que ordena son las claves en lugar de los valores.
array_multisort(array1 , arg1 ó array2,…)	Permite ordenar varios arrays al mismo tiempo, indicando cada array y (opcionalmente) la forma de ordenar. La forma de ordenar se indica mediante las palabras **SORT_ASC** (orden ascendente), **SORT_DESC** (orden descendente), **SORT_REGULAR** (orden normal), **SORT_NUMERIC** (ordenación numérica), **SORT_STRING** (ordenación para texto). Ejemplo: array_multisort($a,SORT_DES, SORT_NUMERIC,$b,SORT_STRING) En el ejemplo, el primer array (***$a***) se ordena en descendente y de forma numérica, mientras que el segundo (***$b***) se ordena de forma textual.
natsort(array)	Ordena el array de modo que se ordena en la forma en la que lo haría un ser humando. Es decir el texto ***imagen2*** iría antes que ***imagen10***. Se respeta la relación entre clave y valor, por lo que no se pierden las claves.
natcasesort(array)	Igual que la anterior, pero no distingue entre mayúsculas y minúsculas.

Búsquedas y filtros en arrays

función	significado
array_search (valorBusq,array [,estricto])	Busca el valor indicado en el array y devuelve la clave del valor en el array o **false** si no lo encuentra. El tercer parámetro (***stricto***) es opcional y en caso de valer verdadero (por defecto vale falso), sólo encuentra el valor si tiene el mismo valor en el array y además es del mismo tipo.
in_array(valorBusq,array [,estricto])	Busca el valor indicado en el array, de forma estricta o no (ver función anterior) y devuelve **true** si lo encuentra o **false** en caso contrario.
array_key_exists(array , clave)	Devuelve verdadero si en el array existe la clave indicada.
preg_prep(expresionRegular,array[,flag])	Devuelve un array que contiene los valores del array que cumplen la expresión regular. Si se usa el tercer parámetro con la constante **PREG_GREP_INVERT**, el array resultado contiene los elementos que no cumplen la expresión regular.
array_filter(array,función)	El segundo parámetro es el nombre de una función existente indicada entre comillas. Dicha función se creará de modo que tenga un solo parámetro y devuelva verdadero o falso en base a una condición que se valorará en el parámetro. **array_filter** aplica dicha función a cada elemento del array y devuelve un nuevo array que contiene los elementos del primer array a los que, pasándolos como parámetro a la función, ésta devuelva falso. Es decir elimina todos los elementos que no cumplan la condición establecida por la función.

Funciones para manejo aleatorio

función	significado

shuffle(array)	Reordena de forma aleatoria el array
array_rand(**array**)	Devuelve un índice aleatorio del array

Funciones de manipulación del contenido del array

función	significado
array_keys($array)	Devuelve un array escalar que contiene todas las claves del array
array_values($array)	Devuelve un array escalar en el que están los valores del array (los índices se retiran y se cambian por escalares)
array_flip($array)	Intercambia las claves por los valores en el array. Es decir: las claves pasan a ser valores y los valores las claves de dichos valores.

Búsquedas y filtros en los arrays

función	significado
array_combine(array1, array2)	Toma los valores de ambos arrays y devuelve un nuevo array donde las claves son los valores del primer array y los valores son los del segundo array. Ambos arrays deben de tener el mismo número de valores (de otro modo ocurrirá un error). Las claves de ambos arrays se ignoran en el resultado final.
array_fill(inicio,n,valor)	Devuelve un array que contiene ***n*** elementos, todos ellos con el valor indicado. El parámetro ***inicio*** indica cuál será el primer índice del array (es numérico); los siguientes índices serán los números consecutivos al inicial
array_fill_keys(claves,valor)	Devuelve un array a partir de otro (parámetro ***claves***) array cuyos valores se considerarán las claves del nuevo. Todos los elementos del nuevo array valdrán el valor indicado.
array_unique(array,**[**flags**]**)	Elimina los valores duplicados en el array. El segundo parámetro sirve para indicar la forma de comparar los valores y utiliza las mismas posibilidades que las indicadas en la función **sort**.
array_pad(array,n,valor)	Devuelve un nuevo array copia del primero, pero en el que se rellena del valor indicado hasta alcanzar el tamaño indicado por ***n***. Si ***n*** es un número positivo el relleno se hace hacia la derecha y si no, hacia la izquierda. Ejemplo: $a=array("a","b","c"); $b=**array_pad**($a, 5, "x") El array b será: ***a, b, c, x, x***
array_slice(array,posInicio[,posFin])	Devuelve una porción del array que se indica de modo que se toman los elementos desde la posición inicial indicada, hasta la posición final indicada. Si el parámetro ***posFin*** se omite, entonces se toma desde la posición inicial hasta el último elemento del array. Las posiciones indicadas se toman de forma escalar, es decir el primer elemento tiene la posición cero, el segundo uno,… El array original no cambia, pero en el resultante los índices nuevos será escalares; es decir, comienzan a numerar los índices desde el cero.
array_splice(array,posInicio[,tamaño[,arraySubst]])	Elimina los elementos del array usado como primer parámetrode la función desde la posición marcada mediante elparámetro ***posInicio***hasta el final. Este parámetro puede sernegativo y entonces la posición (***posInicio***) desde la que eliminar se cuenta desde el final. A la hora de indicar la ***posInicio***, se maneja el array como si fuera escalar. Es decir el primer elemento es el cero, el segundo es el

	uno,… Devuelve un array con los elementos eliminados. Si se usa el parámetro ***tamaño*** entonces se recorta el número indicado por ese parámetro (en lugar de llegar hasta el final). El parámetro ***arraySubst*** sirve para indicar un array que contiene los elementos con los que se sustituirán los eliminados.
array_count_values(array)	Genera un nuevo array en el que los índices son los valores y el valor es el número de veces que cada valor antiguo aparecía en el array original
array_sum(array)	Suma todos los valores del array y retorna el resultado
array_product(array)	Multiplica todos los valores del array y retorna el resultado
array_shift(array)	Retira el primer elemento del array, desplazando a todos los demás elementos en el mismo. Es decir, no deja el hueco del eliminado. Los índices se ponen de forma escalar, es decir comienzan a numerarse desde el número cero.
array_unshift(array,valor1,valor2,…)	Coloca al principio del array los valores indicados
array_pop(array)	Retira del array a su elemento y lo devuelve como resultado
array_push(array,valor1, valor2,…)	Coloca los valores indicados (al menos uno) al final del array original. Combinado con **array_pop** permite simular pilas
array_map(función, array)	Indica una función existente (pero entrecomillada) y devuelve un array resultado de aplicar la función indicada a cada elemento del array. La función se debe definir usando un solo parámetro, ese parámetro representa a cada elemento del array.

array_walk(array, función)	Indica una función existente (pero entrecomillada) y la aplica a cada elemento del array. La función al definirla usará un solo parámetro que representa a cada elemento del array. Dicho elemento se define por referencia al crear la función (es decir utiliza el operador &) de ese modo la función realmente podrá modificar el valor de cada elemento..
	Ejemplo:
	Function doble(&$x){
	$x*=2;
	}
	$array=**array**(1,2,3,4,5);
	array_walk($array,"doble");
	print_r($array);
	Escribiría: ***2,4,6,8,10***

array_walk_recursive(array, función)	Indica una función existente (pero entrecomillada) y la aplica a cada elemento del array. La función al definirla usará un solo parámetro que representa a cada elemento del array. En el caso de que los valores del array sean otros arrays, se ejecutará la función para dichos arrays.
array_chunk(array,tamaño [,respetarClaves])	Devuelve un array multidimensional, resultado de dividir el array que se pasa como parámetro en trozos del tamaño indicado. El parámetro opcional **respetarClaves**, si vale **true** (por defecto es **false**) respeta las claves originales; de otro modo las claves se pierden y cada array se renumera de formaescalar.

Combinación de arrays

función	significado
array_merge($array1,$array2,…)	Une los dos arrays que se le pasan los valores del segundo van a continuación de los del primero. Si el array es asociativo, en caso de repetir claves, sólo se queda con las últimas.

	$a=array("uno"=>"Pedro","dos"=>"Antonio","tres"=>"Santiago"); $b=array("dos"=>"Sara","tres"=>"Antonio","cuatro"=>"Santiago"); $c=array_merge($a,$b); print_r($c); */* obtiene:* *Array(* *[uno] => Pedro* *[dos] => Sara* *[tres] => Antonio* *[cuatro] => Santiago* *)* Si el array es escalar se renumeran de nuevo todas las claves y sí aparecen las claves con índices repetidos.
array_merge_recursive($a1,$a2,...)	Igual que el anterior, sólo que ahora si hay índices repetidos, en el array resultante se convertirán en un array que combina todos los valores de ese índice.
array_intersect($array1,$array2,…)	Genera un nuevo array, intersección de los indicados. En el array resultante sólo aparecen los valores duplicados en todos los arrays. Se mantienen los índices, pero toma los primeros índices que aparezcan en la lista de arrays.
array_intersect_key($array1,$array2,…)	Genera un nuevo array en el que aparecen las claves presentes en todos los arrays indicados como parámetros.
array_diff($array1,$array2,…)	Genera un nuevo array en el que aparecen los valores del primer array que no están en el segundo (array de diferencia). La relación entre clave y valor se mantiene.
array_diff_key($array1,$array2,…)	Genera un nuevo array en el que aparecen las claves del primer array que no están en el segundo. La relación entre clave y valor se mantiene.

Conversión de matrices a partir de variables comunes y viceversa

función	significado
compact(lista de variables)	Se le pasa una lista de variables. La lista es en realidad una lista de textos (strings) que contienen el nombre de las variables. La función devuelve un array con esos valores
list(listaDeVariables)=array	list no es una función, realmente permite asociar valores de un array a una lista de variables. Ejemplo: $a=array(979797979,"Manuel",1250.45); list($tlfno,$nombre,$salario)=$a; Las variables ***$tlfno***, ***$nombre*** y ***$salario*** tomarán cada una el valor que les corresponda del array.

7.- STRINGS

La palabra String en inglés significa ***cadena*** y por eso es muy habitual en muchos libros de informática referirse así a este tipo de dato. Sin embargo, en realidad por String se entiende a una serie de caracteres, lo que normalmente llamamos simplemente texto.

¿Por qué no llamarlas simplemente textos? Porque no todos los strings son textos. Es decir, hay series de caracteres que no representan necesariamente textos. Esto por ejemplo: ***!"·$%&/()=?*** es un String en cuanto a qué es una serie de caracteres, pero no diríamos que es un texto al no ser entendible.

En toda aplicación informática los strings son el tipo de dato fundamental. Tanto es así que en realidad ya hemos usado strings en los apartados anteriores. Sin embargo, en este punto vamos a desglosar todo el manejo de strings que permite PHP.

Se asignan a las variables entrecomillando (en simples o dobles) el texto a asignar. Para asignar, como ya se dijo se usa =.

Si el propio texto lleva comillas, se puede utilizar combinaciones de las comillas para evitar el problema.

```
$cad="Esto es una cadena";   //definicion de cadena con "
$cad='Y esto otra';     //definicion de cadena con '
$cad="Juan 'El Magnifico' está aquí"; //inclusión de ' en una cadena
$cad='Pedro "El Secundario" también '; //inclusion de " en una cadena
```

En las cadenas con comillas “ se pueden realizar unas operaciones adicionales:

- ***Usar caracteres especiales***.

secuencia de escape	Significado
\t	Tabulador
\n	Nueva línea
\f	Alimentación de página
\r	Retorno de carro
\”	Dobles comillas
\’	Comillas simples
\\	Barra inclinada (***backslash***)
\$	Símbolo dólar

- ***Incluir el valor de una variable***. La variable puede indicarse directamente o usando {} para delimitar donde empieza y acaba.

```
$cad="Juan \"El Valiente\" se asustó\n"; //uso de caracteres especiales

$nombre="Rosa";
$cad1="Le he dado dinero a $nombre\n";   //inclusion de una variable
$cad2="Le he dado \$ a {$nombre}\n"; //delimito la variable con {}
```

Otra operación básica es la concatenación. Como se ha comentado en apartados anteriores, el operador de concatenación de textos es el símbolo del punto (.).

Ejemplos:

```
$texto1="Hola";
$texto2="a todos y todas";
$texto3=$texto1." ".$texto2;
echo $texto3;
```

Por pantalla saldría ***Hola a todos y todas***. Como se observa en el ejemplo se pueden concatenar tanto variables de tipo strings como textos literales (se encadena el contenido de la variable ***$texto*** se añade un espacio en blanco y se concatena con el contenido de la variable ***texto2)***.

Podemos usar la concatenación y asignación:

```
$texto1="Hola ";
$texto1.="a todos y todas";
echo $texto1;
```

El resultado es como en el texto anterior ya que el símbolo **.=** permite añadir al final de la variable el String que se indique.

7.1.- Definición de cadenas.

Además de poder definir las cadenas con "" o '', es posible declarar cadenas **HEREDOC** y **NOWDOC**

Cadena HEREDOC

Son cadenas equivalentes a las definidas con "" pero que se pueden definir en varias líneas

```
$nombre="Jorge";
$texto=<<<fin
Mi querida amiga <br />
escribo estas líneas esperando que me leas. <br />
Firmado: $nombre<br />
fin;

echo$texto;
```

En el ejemplo, de color azul vemos el texto que se almacena en la variable ***$texto***. El texto a asignar es el que sigue al símbolo de inserción de documento (<<<) y al **marcador de texto**, que es un grupo de caracteres concreto (en este caso se ha usado la palabra ***fin*** como marcador de texto). El marcador se indica inmediatamente después del símbolo <<< y vuelve a aparecer en la primera columna tras el último carácter que se almacenará en la variable (al inicio de línea). Es decir, escribe:

```
Mi querida amiga
escribo estas líneas esperando que me leas.
Firmado: Jorge
```

Cadenas NOWDOC

Son equivalentes a cadenas definidas con ‘’(comillas simples), es decir, no se interpretan los caracteres de escape ni se evalúan las variables.

Se definen de forma similar a las cadenas HEREDOC, Para diferenciar las dos notaciones, en ésta el marcador de línea va entre comillas simples (en la declaración, no en el cierre):

```
$nombre="Jorge";
$texto=<<<'fin'
Mi querida amiga <br />
escribo estas líneas esperando que me leas. <br />
Firmado: $nombre <br />
fin;

echo $texto;
```

En este caso la salida por pantalla sería:

```
Mi querida amiga
escribo estas líneas esperando que me leas.
Firmado: $nombre
```

No se ha interpretado la variable ***$nombre***, se ha entendido que es un texto normal.

7.2.- Manejo de strings como array de caracteres.

Se puede trabajar con las cadenas como si fueran un array donde cada elemento es un carácter.

```
$string1="Este es el texto de prueba";
$string1[6]="X";
echo $string1;//Sale: Este eX el texto de prueba
```

Aunque realmente no se considera un array; de hecho no se admite el uso de foreach en strings.

Es posible usar índices negativos. En este caso, un índice negativo se interpreta como o un índice desde el final de la cadena. Los índices negativos se pueden usar en cualquier función de cadenas que soporte índices.

```
$cadena="La casa es grande";

//usando notación array accedo al penúltimo caracter
echo "El penúltimo caracter es {$cadena[-2]}";

//me devuelve los 2 carateres ultimos
$parte=mb_substr($cadena, -2);
```

7.3.- Cadenas Multibyte

Aunque existen muchos idiomas en los cuales cada carácter puede ser representado por una referencia uno a uno a un valor de 8 bits, existen también bastantes idiomas que requieren tantos caracteres para la comunicación escrita que no pueden ser representados dentro del rango que un mero byte puede codificar (con un byte solo se pueden representar 256 valores únicos (2 a la 8ª potencia). Los esquemas de codificación multibyte fueron desarrollados precisamente para expresar más de 256 caracteres en el sistema de codificación regular a nivel de bits.

De esta forma nos encontramos con tablas de caracteres que usan un solo byte para codificar cada carácter como por ejemplo latin1 o ISO8859-1 y con tablas de caracteres que permiten definir caracteres usando 2, 3 o 4 bytes como por ejemplo UTF8 o UNICODE. Esto se puede apreciar cuando trabajamos con un string como array. En este caso, cada posición del array es uno de los bytes usados para la definición del carácter.

```
$cadena="Hola niño";
```

$cadena	"Hola niño"
length	10
[0]	48
[1]	6f
[2]	6c
[3]	61
[4]	20
[5]	6e
[6]	69
[7]	c3
[8]	b1
[9]	6f

La cadena está codificada con UTF8 (es la página de códigos usada al codificar el documento). UTF8 usa dos bytes para representar la ñ. Por lo que la cadena tiene 9 caracteres mientras que el array tiene 10 posiciones (las posiciones 7 y 8 representan la ñ)

Cuando se manipulan cadenas de caracteres (trim, split, splice, etc.) en una codificación multibyte, es necesario utilizar funciones especiales, ya que dos o más bytes consecutivos pueden representar un único carácter en tal esquema de codificación. Si, de lo contrario, se usa una función que no considera caracteres multibyte con la cadena de caracteres, es probable que falle al detectar el comienzo o el final del carácter multibyte, y que se termine con una cadena de caracteres corrupta que probablemente pierda su significado original.

MBSTRING proporciona funciones específicas para cadenas de texto multibyte que ayudan a tratar codificaciones multibyte en PHP. Además, *mbstring* controla la conversión de la codificación de caracteres entre los posibles esquemas de codificación. *mbstring* está diseñada para manejar codificaciones basadas en Unicode, tales como UTF-8 y UCS-2.

Actualmente, la mayor parte de las aplicaciones de PHP están escritas con las funciones estándar de cadenas de caracteres, como por ejemplo substr(), la cual se sabe que no maneja correctamente los strings codificados de forma multibyte. Para solventar esto, mbstring admite la 'sobrecarga de funciones', característica que permite que tales aplicaciones consideren los multibytes sin necesidad de modificar el código, sobrecargando las funciones homólogas multibytes

sobre las estándar de cadenas de caracteres. Por ejemplo, se invocaría a mb_substr() en lugar de a substr() si se habilitara la sobrecarga de funciones. En muchos casos, esta funcionalidad simplifica la portabilidad de las aplicaciones que tan solo admiten codificaciones de un único byte a entornos multibyte.

Para utilizar la sobrecarga de funciones, se ha de establecer *mbstring.func_overload* en *php.ini* a un valor positivo que represente una combinación de máscaras de bits que especifiquen las categorías de las funciones que se sobrecargarán. Para sobrecargar la función mail() debe establecerse a 1. Para funciones de string, a 2. Para funciones de expresiones regulares, a 4. Por ejemplo, si se estableciera a 7, se sobrecargaría las funciones de mail, de string, y de expresiones regulares. A continuación, se muestra la lista de funciones sobrecargadas.

valor de mbstring.func_overload	función original	función sobrecargada
1	mail()	mb_send_mail()
2	strlen()	mb_strlen()
2	strpos()	mb_strpos()
2	strrpos()	mb_strrpos()
2	substr()	mb_substr()
2	strtolower()	mb_strtolower()
2	strtoupper()	mb_strtoupper()
2	stripos()	mb_stripos()
2	strripos()	mb_strripos()
2	strstr()	mb_strstr()
2	stristr()	mb_stristr()
2	strrchr()	mb_strrchr()
2	substr_count()	mb_substr_count()
4	ereg()	mb_ereg()
4	eregi()	mb_eregi()
4	ereg_replace()	mb_ereg_replace()
4	eregi_replace()	mb_eregi_replace()
4	split()	mb_split()

Como se ve, esta librería dispone de todo un conjunto de funciones equivalentes a las clasicas str.

7.4.- Expresiones regulares

Uno de los usos más habituales en la mayoría de lenguajes de programación, tiene que ver con expresiones regulares.

Se utilizan, sobre todo, para establecer un patrón que sirva para conseguir que ciertos textos

la cumplan. Ese patrón permite búsquedas avanzadas, criterios avanzados de verificación de claves o códigos (por ejemplo, números de serie de productos que cumplen unas reglas muy concretas), etc.

PHP permite el uso de dos tipos de expresiones regulares:

- **POSIX (Portable Operating System Interface) 1003.2** correspondientes a unestándar aceptado por el organismo IEEE (***The Institute of Electrical andElectronics Engineers***) muy influyente en normas electrónicas y que se basa enla sintaxis del sistema **Unix**. Las funciones compatibles con este formato comienzan por la palabra **ereg**.
- **PCRE (Perl Compatible Regular Expressions)**. Parte de PHP desde la versión4.2, procedente del lenguaje **Perl** (precisamente famoso por su uso de las expresiones regulares). Es el formato que más se usa, de hecho, se recomienda no utilizar el anterior. Las funciones que utilizan este formato de expresiones regulares comienzan por la palabra **preg**. Por otro lado, este formato escompatiblecon Unicode, por lo que también es más recomendable para lenguas que usan símbolos fuera del ASCII original como el castellano.

formato de las expresiones regulares PCRE

Las expresiones regulares utilizan símbolos especiales (meta-caracteres) para indicar el patrón correspondiente. Las expresiones regulares de tipo Perl deben ir delimitadas por un carácter que debe aparecer al principio y al final. Normalmente este carácter es **/,** aunque se suele usar también #, % y los delimitadores estilo corchete (), [], {}, <>. Si el delimitador se usa en la expresión regular debe escaparse (/a\/.*/)

Los símbolos y patrones que se pueden utilizar son:

función	significado
c	Si ***c*** es un carácter cualquiera (por ejemplo ***a***, ***H***, ***ñ***, ***etc.***) indica, donde aparezca dentro de la expresión, que en esa posición debe aparecer dicho carácter para que la expresión sea válida. ***cde***Siendo ***c***, ***d***, y ***e*** caracteres, indica que esos caracteres deben aparecer de esa manera en la expresión.
(*x*)	Permite indicar un subpatrón dentro del paréntesis. Ayuda a formar expresiones regulares complejas.
.	Cualquier carácter. El punto indica que en esa posición puede ir cualquier carácter excepto el de nueva línea.
^*x*	Comenzar por. Indica el String debe empezar por la expresión ***x***
***x*$**	Finalizar por. Indica que el String/linea debe terminar con la expresión x.
***x*+**	La expresión a la izquierda de este símbolo se puede repetir una o más veces
x*	la expresión a la izquierda de este símbolo se puede repetir cero o más veces
***x*?**	El carácter a la izquierda de este símbolo se puede repetir cero o una veces

***x*{*n*}**	Cuantificador min-max. Significa que la expresión ***x*** aparecerá ***n*** veces, siendo ***n*** un número entero positivo.
***x*{*n*,}**	Cuantificador min-max. Significa que la expresión ***x*** aparecerá ***n*** o más veces
***x*{*m*,*n*}**	Cuantificador min-max. Significa que la expresión ***x*** aparecerá de ***m*** a ***n*** veces.
X*|*y	La barra indica que las expresiones ***x*** e ***y*** son opcionales, se puede cumplir una u otra. Por emplo, el patrón /camina(nte|ado|)/ coincide con las palabras caminante, caminado y camina
[*cde*]	Se enlaza a uno de los caracteres indicados. Por ejemplo, ***[acf]*** se enlazaría al carácter ***a***, ***c*** ó ***f***
[*c*-*e*]	Clase caráter. Cumplen esa expresión los caracteres que, en orden ASCII, vayan del carácter ***c*** al carácter ***d***. Por ejemplo ***a-z*** representa todas las letras minúsculas del alfabeto inglés.
[^*x*]	No es válido ninguno de los caracteres que cumplan la expresión ***x***. Por ejemplo ***[^dft]*** indica que no son válidos las caracteres ***d***, ***f*** ó ***t***.
[*[: :]*]	Permite indicar una clase de caracteres por nombre. Por ejemplo, ***[[:alpha:]]*** representa cualquier carácter que sea una letra. ***[01[:alpha:]%]*** se enlaza con los caracteres 0, 1, letra y %. Nos podemos encontrar con las siguientes clases: Alnum, letras y dígitos Alpha, letras Ascii, códigos de caracteres de 0-127 Blank, carácter de espacio o tabulador Cntrl, caracteres de control Digit, dígitos decimales (igual que \d) Graph, caracteres imprimibles, excepto el espacio en blanco Lower, Letras en minúsculas Print, caracteres imprimibles incluyendo el espacio en blanco Punct, caracteres imprimibles excluyendo letras y dígitos Space, espacio en blanco (no es igual a \s) Upper, letras en mayúsculas Word, caracteres “word” (igual a \w) Xdigit, dígitos hexadecimales
****	Carácter de escape general
\d	Dígito, vale cualquier dígito numérico
\D	Todo menos dígito
\s	Espacio en blanco
\S	Cualquier carácter salvo el espacio en blanco
\h	Cualquier carácter espacio en blanco horizontal
\H	Cualquier carácter que no es espacio en blanco horizontal
\v	Cualquier carácter espacio en blanco vertical
\V	Cualquier carácter que no es espacio en blanco vertical
\w	***Word***, carácter válido dentro de los que PHP considera para identificar variables. Es decir, letras, números o el guion bajo.
\W	Todo lo que no sea un carácter de tipo ***Word***.
\n	Nueva línea
\t	Tabulador
c	Permite representar el carácter ***c*** cuando este sea un carácter que de otra manera no sea representable (como ***[***, ***]***, ***/***, \,…). Por ejemplo \\ es la forma de representar la propia barra invertida.
\a	Alarma, carácter BEL (07 Hex)

\f	Salto de página (0C hex)
\e	Escape (1B hex)
\n	Nueva línea (0A hex)
\r	Retorno de carro (0D hex)
\R	Salto de línea. Coincide con \n, \r y \r\n
\c*x*	Control-x, donde x es cualquier carácter
\x*ff*	Permite indicar un carácter **Unicode** mediante su código hexadecimal
\p{*xx*}	Indica un carácter que cumpla la propiedad Unicode indicada con los símbolos *xx* que pueden ser: **Cc** Control **Cf** Formato **Cn** Sin asignar **Co** Uso privado **Cs** Sustituto **L** Letra **Ll** Letra minúscula **Lm** Letra modificadora **Lo** Otra letra **Lt** Letra de título **Lu** Letra mayúscula **M** Marca **Mc** Marca de espacio **Me** Marca de cierre **Mn** Marca de no-espacio **N** Número **Nd** Número decimal **Nl** Número letra **No** Otro número **P** Puntuación **Pc** Puntuación de conexión **Pd** Puntuación guión **Pe** Puntuación de cierre **Pf** Puntuación final **Pi** Puntuación inicial **Po** Otra puntuación **Ps** Puntuación de apertura **S** Símbolo **Sc** Símbolo de moneda **Sk** Símbolo modificador **Sm** Símbolo matemático **So** Otro símbolo **Z** Separador **Zl** Separador de línea **Zp** Separador de párrafo **Zs** Separador de espacio

Lo indicado anteriormente nos permite establecer expresiones regulares básicas. Existen unas características avanzadas que nos permiten indicar un comportamiento adicional. Estas características avanzadas son:

Modificadores

Además de los símbolos y patrones indicados anteriormente, es posible establecer unos modificadores que cambian el comportamiento de la expresión regular. Los modificadores se indican tras la expresión regular. Por ejemplo, si queremos indicar el modificador i (ignorar

mayúsculas y minúsculas) en la expresión regular **/[abx]\d{2,3}/**, la expresión regular sería**/[abx]\d{2,3}/i**.

Los modificadores principales son:

- i (PCRE_CASELESS): Insensible a mayúsculas/minúsculas, es decir, las letras del patrón coincidirán tanto con letras mayúsculas como minúsculas.
- m (PCRE_MULTILINE): Por defecto, PCRE trata la cadena como si fuera una única línea de caracteres, aunque incluya saltos de línea. Por lo tanto, ^ coincide sólo con el inicio de la cadena, mientras que $ coincide sólo con el final. Cuando se aplique este modificador, los símbolos ^ y $, coinciden con inmediatamente después e inmediatamente antes de cualquier nueva línea, así como al inicio y final absolutos.
- s (PCRE_DOTALL): Si se aplica este modificador, el meta-carácter . coincide con todos los caracteres incluyendo nueva línea. Sin él, las nuevas líneas son excluidas.

Los modificadores anteriores se aplican a toda la expresión regular. También es posible aplicar los modificadores a una parte concreta del patrón. Para ello se utiliza **(?** y **)**. Por ejemplo, **(?im)** establece la coincidencia insensible a mayúsculas-minúsculas y multilínea. También es posible eliminar estas opciones precediendo la letra con un guión. Por ejemplo, **(?im-s)** establece PCRE_CASELESS y PCRE_MULTILINE y elimina PCRE_DOTALL.

Lo indicado anteriormente se puede aplicar a:

- Patrón. Se indica directamente en el patrón. Los modificadores se aplicarían en el resto del patrón a partir de donde se indiquen.
  Por ejemplo, el patrón /ab(?i)c/ coincide con abc y abC.
- Subpatrón. Un cambio de opción afecta sólo a aquella parte del subpatrón que le sigue.
  Por ejemplo, /a(b(?i)c)d/ coincide con abcd y abCd y ninguna otra cadena.
- Ramas de opciones. Cualquier cambio hecho en una alternativa continúa en ramas subsiguientes del patrón/subpatrón.
  Por ejemplo, /a(b(?i)c|d)e/, coincide con abce, abCe, ade y aDe

Subpatrones

Cuando se indican paréntesis - ( ) -, se está estableciendo un subpatrón de captura, de forma que cuando encontramos en la cadena una porción que coincide con el subpatrón, esta parte es devuelta.

A cada porción capturada se le asigna un número en secuencia. Por ejemplo, si tenemos la cadena "el rojo amanecer" y el patrón ((rojo|oscuro) (amanecer|atardecer)) las cadenas capturadas son "rojo amanecer", "rojo" y "amanecer" numeradas respectivamente como 1, 2 y 3.

Si se indica **(?:**, el subpatrón no se captura y no es numerado. Por ejemplo, si tenemos la cadena "el oscuro atardecer" y el patrón ((?:rojo|oscuro) (amanecer|atardecer)) las cadenas capturadas son "oscuro atardecer" y "atardecer" numeradas respectivamente como 1 y 2.

Repeticiones.

Por defecto, los cuantificadores son codiciosos, es decir, comparan todo lo posible hasta el número máximo de veces. Este funcionamiento puede que no lo queramos en algún caso. Por ejemplo, cuando buscamos comentarios. Si tenemos la cadena "/* primer comentario */ no comentado /* segundo comentario */"y el patrón /*.**/, nos obtendría como coincidencia toda la cadena, incluyendo el texto no comentado. Podemos cambiar este funcionamiento si seguimos el cuantificador de **?**. Es este caso se vuelve perezoso, coincidiendo con el número mínimo de veces. Para la cadena anterior si indicamos como patrón /*.*?*/, nos desvolvería dos cadenas con los comentarios, que es lo que deseamos.

Ojo, ? es también un cuantificador por lo que es posible que nos encontremos un patrón como \d??, donde la primera ? es el cuantificador y la segunda ? convierte el cuantificador en perezoso.

Retroreferencias

Una barra invertida seguida por un digito mayor de 0 es una retroreferencia a un subpatrón de captura anterior, siempre que haya habido tantas capturas previas a la izquierda.

Por ejemplo, si tenemos el patrón "(abraz|apreci)o de un \1ador" coincide con la cadena "abrazo de un abrazador" y con "aprecio de un apreciador" pero no con "abrazo de un apreciador".

Es posible usar \g para referenciar una retroreferencia. Por ejemplo, \1, \g1 \g{1} son sinónimas.

De igual forma, es posible indicar un valor negativo en \g lo que indica una referencia relativa. Ej \g{-2}

Declaraciones

Una declaración es una comprobación de los caracteres siguientes o anteriores al punto de coincidencia actual que en realidad no consumen carácter alguno.

Hay dos tipos: aquellas que buscan hacia delante desde el punto actual de la cadena y aquellas que buscan hacia atrás desde él.

Las declaraciones de búsqueda hacia delante comienzan con **(?=** para declaraciones positivas y **(?!** para declaraciones negativas.

Por ejemplo, \w+(?=;) coincide con una palabra seguida de un punto y coma, pero no incluye el punto y coma. El patrón foo(?!bar) coincide con la cadena foo si no va seguida de bar.

Las declaraciones de búsqueda hacia atrás comienzan con **(?<=** para las declaraciones positivas y **(?<!** para las negativas.

Por ejemplo, el patrón (?<!foo)bar encuentra la cadena bar si no está precedida por foo.

El contenido de una declaración de búsqueda hacia atrás está restringido de tal manera que todas las cadenas que se comparen con ella deben tener una longitud fija.

Varias declaraciones (de cualquier tipo) pueden producirse en sucesión. Por ejemplo, (?<=\d{3})(?<!999)foo coincide con "foo" precedido de tres dígitos que no sean "999". Nótese que cada una de las declaraciones es aplicada en el mismo punto de la cadena objetivo. Primero, se verifica que los tres caracteres previos son todos dígitos, después se verifica que esos mismos tres caracteres no sean "999". Este patrón no coincide con "foo" precedido de seis caracteres, los

primeros de los cuales son dígitos y los tres últimos de éllos no son "999". Por ejemplo, no coincide con "123abcfoo".

Para el caso anterior podríamos tener el patrón (?<=\d{3}...)(?<!999)foo Esta vez, la primera declaración examina los seis caracteres precedentes, verificando que los tres primeros son dígitos, y después, la segunda declaración verifica que los tres caracteres anteriores no son "999".

Un caso típico, es cuando queremos comprobar que una contraseña cumple unos requisitos: al menos una letra minúscula, al menos una letra mayúscula, al menos un dígito y al menos un carácter especial con una longitud de entre 8 y 15 caracteres. El patrón sería:

```
/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[$@!%*?&\-_+])[A-Za-z\d$@!%*?&\-_+]{8,15}$/
```

funciones de expresiones regulares

función	significado
preg_match(patrón, string[, arrayResult] [, flag] [,despl])	Indica una expresión regular (apartado ***patrón***) y la comprueba en el String que se pasa a la función. Devuelve verdadero en el caso de que el texto cumpla la expresión regular. **arrayResult**es un array que se puede indicar para almacenar en él el texto dentro del String que cumple el patrón. De modo que el elemento cero del array será el texto completo que cumpla todo el patrón, el elemento 1 será el texto que cumpla el primer subpatrón que se haya definido (los subpatrones se indican entre paréntesis), etc. El parámetro **flags**permite utilizar la constante **PREG_OFFSET_CAPTURE** para que el array anterior almacene además de cada texto que cumpla la condición, la posición en la que se estaba ese texto dentro del array original (lo coloca en el array detrás de cada texto que cumpla la condición) **despl**permite indicar el índice dentro del String por el que se empezará a buscar el patrón (si no se indica este parámetro comenzamos a buscar por el principio).
preg_match_all(patrón, string[, arrayResult] [, flag] [,despl])	Igual que la anterior, sólo que cuando encuentra el patrón, sigue buscando en el resto del texto para buscar la siguiente aparición. Esto la hace más útil para usar arrays de resultados.
preg_replace(patrón,reemplazo,string[, límite] [, cuenta])	Busca el patrón en el string indicado y cuando le encuentra, devuelve un nuevo string resultado de sustituir el patrón encontrado por el texto de reemplazo indicado. Ejemplo: $s="Este es un documento Java, y no de la isla de Java"; **echo** preg_replace("/java/i", "PHP",$s); Escribe ***Este es un documento PHP, y no de la isla de PHP*** El patrón puede incluso ser un array con varios patrones: si es así, se sustituye cada patrón del array en el string por el texto de reemplazo indicado. Es posible que incluso el reemplazo sea un array de textos de reemplazo. Si es así: cada elemento del array de patrones se reemplaza por el elemento del array de reemplazo correspondiente. El parámetro opcional ***límite*** pone un tope al número de reemplazos efectuados. Por ejemplo si vale **1**, sólo se reemplaza la primera aparición del patrón. Por defecto vale -1 (es decir, no hay límite). La variable ***cuenta***, si se utiliza, sirve para almacenar el número de

	reemplazos efectuados.
preg_split(patrón, string[, límite[, flags]])	Divide el string indicado según el patrón de expresión regular indicado y devuelve un array que contiene cada trozo obtenido del string. Ejemplo: $s="texto a dividir, ole y ole"; print_r (preg_split("/[\s,]+/", $s)); Sale: Array([0] => texto [1] => a [2] => dividir [3] => ole [4] => y [5] => ole) El parámetro opcional ***límite*** indica el máximo de trozos a obtener.

7.5.- Funciones estándar de uso con strings

funciones básicas

función	significado
strlen(string)/ **mb_strlen**(string)	Devuelve el tamaño del String

mayúsculas y minúsculas

función	significado
strtolower(texto)/ **mb_strtolower**(texto)	Convierte el texto a minúsculas
strtoupper(texto)/ **mb_strroupper**(texto)	Convierte el texto a mayúsculas
lcfirst(texto)	Retorna un string resultado de poner en minúsculas el primer carácter del texto (suponiendo que sea una letra).
ucfirst(texto)	Retorna un string resultado de poner en mayúsculas el primer carácter del texto (suponiendo que sea una letra).

funciones de comparación

función	significado
strcmp(texto1, texto2)	Compara los dos textos (strings) y devuelve cero si son iguales, uno si el primero es mayor en orden alfabético (usando el código ASCII) y -1 si es mayor el segundo string.
strccasemp(texto1, texto2)	Igual que la anterior pero no tiene en cuenta las mayúsculas (sólo en inglés)
strnatcmp(texto1,texto 2)	Igual que las anteriores, pero la comparación que hace tiene en cuenta la forma natural humana de ordenar. Así por ejemplo el texto ***imagen2*** sería considerado menor que ***imagen11*** (en orden alfabético es mayor).

strcasenatcmp(texto1,texto2)	Igual que la anterior pero sin considerar mayúsculas ni minúsculas.
levenshtein(texto1, texto2)	Devuelve un número entero conocido como distancia **Levenshtein** que simboliza el número de modificaciones al primer texto que nos permitirían conseguir el segundo texto. De modo que si esa distancia es pequeña, los dos textos son parecidos.
metaphone(texto [,fonemas])	Devuelve un código que indica la forma de pronunciar una palabra (usando el inglés), de modo que dos palabras con pronunciación similar tendrían el mismo código. El parámetro ***fonemas*** indica el número máximo de fonemas a tener en cuenta (cuantosmenos fonemas, más palabras parecidas habrá)
similar_text(texto1,texto2[,porcentaje])	Devuelve el número de caracteres parecidos entre el ***texto1*** y el ***texto2***. En realidad lo interesante es usar el tercer parámetro ***porcentaje***. Este parámetro se pasa por referencia por lo que tiene que ser una variable. Dicha variable recibe un porcentaje de similitud entre ambas cadenas de texto.

Funciones de búsqueda y reemplazo

función	significado
strpos(texto, textoBusq)/ **mb_strpos**(texto, textoBusq)	Devuelve la posición del segundo texto dentro del primero (empieza a contar la posición desde el número cero). Si el texto buscado no se encuentra, devuelve ***false***. Ejemplo: $a="Esta es la comunidad de Castilla y León"; **echo strpos**($a," Castilla"); Escribiría ***24***, ya que ***Castilla*** empieza a aparecer en la posición 24 del string ***$a***.
stripos(texto, textoBusq)/ **mb_stripos**(texto, textoBusq)	Igual que la anterior pero no tiene en cuenta mayúsculas ni minúsculas. Sólo es válida con textos que no usen caracteres fuera del alfabeto inglés.
strpbrk(texto, listaCars)	***listaCars***es un string que contiene todos los caracteres que deseamos buscar en el texto de modo que busca cualquiera de esos caracteres dentro del texto. La función si encuentra cualquiera de esos caracteres, devuelve el resto de caracteres desde la posición del primer carácter que encuentre en el texto. Ejemplo: **echo strpbrk**("este año no voy a la montaña","ñyn"); Escribe (puesto que encuentra primero a la eñe): ño no voy a la montaña
str_replace(textoBuscado,textoReemplazo,texto[,veces])	Localiza todas las apariciones del ***textoBuscado*** en el texto que se pasa como tercer parámetro y las cambia por el texto indicado en ***textoReemplazo***. El cuarto parámetro (opcional), ***veces***, se envía por referencia y almacenará el número de reemplazos realizados
str_ireplace(textoBuscado,textoReemplazo,texto[,veces])	Idéntica a la anterior, pero no distingue entre mayúsculas y minúsculas.
substr_replace(texto,textoReemplazo,posInicial [,posFinal])	Coloca en el texto original, el texto indicado como ***textoReemplazo***, de modo que sustituya a todos los caracteres desde la posición indicada como ***posInicial***, hasta el final o hasta el número indicado con el parámetro***posFinal***.

strtr(texto,arrayTraducción**)**	Hace reemplazos múltiples de caracteres en el texto y devuelve el texto resultante de realizar esos reemplazos. El array que se usa como segundo parámetro, contiene datos de forma que cada índice se buscará en el texto y se reemplazará por su valor. Ejemplo: $array=**array**("a"=>"á","e"=>"é","i"=>"í","o"=>"ó"); **echo strtr**("este texto tiene", $array) Escribe: ésté téxtó tíéné
stripslash(texto**)**	Elimina en el texto todos los caracteres ***backslash***(\). Es muy útil para texto procedente de lenguajes de programación.
strip_tags(texto[,noRetirables]**)**	Retira del texto todas las etiquetas de tipo HTML o PHP que haya contenidas, excepto las que se indiquen en el parámetro ***noRetirables***, que es un string que contendrá las etiquetas que no queremos retirar seguidas. Ejemplo: **strip_tags**(texto,"<p><strong") Elimina del texto todas las etiquetas que haya excepto ***p*** y ***strong***.

Funciones de extracción de texto

función	significado
strstr(texto, textoBusq**)/** **mb_strstr(**texto, textoBusq**)**	Busca un string dentro de otro y devuelve los caracteres desde su primera aparición hasta el final. En la respuesta incluye el texto buscado. Ejemplo: $a="Esta es la comunidad de Castilla y León"; **echo strstr**($a," Castilla"); Escribiría ***Castilla y León***
stristr(texto, textoBusq**)**	Igual que la anterior pero no tiene en cuenta mayúsculas ni minúsculas. Sólo es válida con textos que no usen caracteres fuera del alfabeto inglés.

strtok(string [,token]**)**	Permite extraer un texto en base a una serie de caracteres de modo que primero coge el primer texto dentro del string original (primer token) y en las siguientes llamadas irá devolviendo el resto de tokens (en las siguientes llamadas no se usa el primer parámetro) hasta que tras devolver el último retorna el valor falso.
	Ejemplo:
	$s="este es el texto, lo vamos a partir";
	$tok=strtok($s,",");
	while($tok!=false){
	echo $tok."--|--";
	$tok=strtok(",");
	}
	Escribe:
	este es el texto--|--lo vamos apartir

explode(delimitador,texto[,límite]**)**	Devuelve un array donde cada elemento del mismo es una subcadena que procede de separar el texto indicado en base a su delimitador. El parámetro límite indica el máximo número de cadenas a extraer. Ejemplo: $s="soy una frase, con unas, cuantas, comas"; $a=explode(",",$s); print_r($a); Escribirá: Array()

	([0]=>soy una frase [1]=>con unas [2]=>cuantas [3]=>comas)
implode([pegamento,]array)	Une todos los elementos del array para formar un texto que contiene el valor de cada elemento. Usa pegamento como cadena para unir las partes. Si no se indica pegamento se usa la cadena vacía.
str_repeat(texto, veces)	Devuelve un string que contiene el texto indicado repetido las veces que se indiquen en el segundo parámetro.
str_shuffle(texto)	Devuelve una copia del texto donde cada carácter se ha movido aleatoriamente. Es decir, forma un anagrama.
strrev(texto)	Devuelve un string que contiene el texto original al revés.
str_split(texto [,tamaño])	Devuelve un array en el que en cada elemento hay un trozo del texto original. El texto se trocea por tamaño de caracteres; si no se indica ***tamaño*** se divide carácter a carácter; si, por ejemplo, tamaño vale 3, se divide de tres en tres caracteres.
str_word_count(texto[, formato[,listaCaracteres]])	Sin indicar segundo ni tercer parámetro, devuelve el número de palabras encontradas en el texto. El parámetro ***formato*** puede tomar los valores: 1 La función devuelve el número de palabras encontradas 2 Devuelve un array escalar con todas las palabras encontradas 3 Devuelve un array asociativo donde cada valor es cada palabra encontrada y su índice la posición en el texto original. ***listaCaracteres*** es un string que contiene caracteres que en esta función se deben de considerar como parte de la palabra y no como separadores de caracteres. Es muy útil para texto escrito en cualquier lengua distinta del inglés y así considerar a la eñe como parte normal del texto.

Funciones para extraer subcadenas de texto

función	significado
substr(texto, posInicial[,tamaño])/ **mb_substr**(texto, posInicial[,tamaño])	Extrae del string que se pasa como primer parámetro, el texto que va desde la posición indicada (empezando a contar por cero) hasta el final. O bien, extrae desde dicha posición el número de caracteres indicados por el parámetro ***tamaño***.

Funciones de limpieza de texto

función	significado
trim(texto, [caracteres])	Sin usar el segundo parámetro, elimina del texto los espacios en blanco del principio y el final. Elimina también saltos de línea, retornos de carro, tabuladores, nulos y caracteres de salto vertical. El segundo parámetro permite indicar una lista de caracteres entrecomillada que serán los que se eliminen si se encuentran al final y al principio del texto (en lugar de eliminar los espacios).
ltrim(texto, [caracteres])	Igual que el anterior, pero sólo elimina los caracteres al inicio del texto.
rtrim(texto, [caracteres])	Igual que el anterior, pero sólo elimina los caracteres al final del texto.

Obtener códigos ASCII

función	significado
chr(códigoASCII)/ **mb_chr**(códigoUNICODE)	Devuelve el carácter correspondiente al código ASCII indicado.
ord(carácter)/ **mb_ord**(carácter)	Inversa a la anterior. Devuelve el código ASCII o UNICODE del carácter indicado.

Funciones de formato de datos

función	significado
number_format(número[,decimales[,caracterDecimal[,caracterDeMiles]]])	Devuelve un string que contiene al número que se indica formateado de modo que aparece el separador de miles. El segundo parámetro (opcional) permite indicar cuántos decimales vamos a utilizar para mostrar el número. Los otros dos parámetros se utilizan para indicar cuál es el carácter de separador de decimales y el de miles. Ejemplo: **echo number_format**(1234567.892,2,",","."); Muestra: 1.234.567,89
money_format(formato,número)	Devuelve un string resultante de aplicar al número que se recibe como segundo parámetro el formato de moneda indicado mediante el primer parámetro. Sólo funciona en entornos compatibles con la función **strfmon** de lenguaje C (Windows no lo es). El parámetro formato es, a su vez, un string que contiene símbolos especiales para dar formato. Esa expresión está precedida por el símbolo % al que le pueden seguir uno o más de estos caracteres **i**Formatea el número usando los símbolos y separadores pertinentes según lo especificado por la función **setlocale** basándose en el sistema monetario internacional. Por ejemplo, si antes hemos usado ***setlocale("LC_ALL","es-ES")***, el formato del número será el correspondiente al formato monetario español. **N** Idéntica a la anterior pero usa el sistema nacional de moneda, según la configuración regional especificada por **setlocale**. **=*f*** Mediante el símbolo =, se especifica un carácter de relleno para el número (por defecto se usa el espacio en blanco). **^** Deshabilita las opciones de agrupamiento (el separador de miles) **+** Indica signo para el número **(** Muestra los números negativos entre paréntesis **!** Elimina el símbolo de moneda **#*n*** Caracteres de anchura que se usarán para la parte entera del número. **.*p*** Número de decimales que se utilizarán **-**Alineación izquierda del número (normalmente la alineación es derecha)

Cifrado

función	significado
md5(texto [,raw_input])	Devuelve hash correspondiente al texto indicado usando el algoritmo MD5. Si el parámetro ***raw_input*** se indica con valor **true** (por defecto vale **false**), codifica en binario crudo con tamaño 16.
sha1(texto [,raw_input])	Igual que el anterior pero utilizando el algoritmo SHA1
hash(algoritmo,	Genera el hash correspondiente a aplicar sobre el texto el algoritmo

texto[,raw_input])	que se indique en el parámetro ***algoritmo*** (puede ser "sha256" , "md5". La lista completa de posibles algoritmos se puede consultar con la función ***hash_algos*** que devuelve un array con todos los nombres posibles a utilizar.
crc32(texto)	Genera el polinomio CRC32 de comprobación sobre el string indicado. Mediante ese polinomio podemos comprobar la validez de los datos.
crypt(texto, salt)	Función completa para cifrar el texto indicado. Sufuncionamiento es complejo.
str_rot13(texto)	Codifica el texto utilizando la rotación **ROT13** de cifrado.

8.- FECHAS/HORAS

Es muy normal trabajar en nuestra aplicación con fechas/horas. En PHP la información de la fecha y de la hora se almacena internamente como un número de 64 bits, por lo que se admiten todas las fechas útiles posibles (incluyendo años negativos). El rango va aproximadamente de 292 mil millones de años en el pasado hasta lo mismo en el futuro.

Para trabajar con fechas PHP aporta dos aproximaciones:

- Librería de funciones: es la forma básica y que encontramos en PHP desde sus inicios
- Clases específicas. Disponible desde PHP 5.2 y aporta un enfoque basada en clases para gestionar cada aspecto de una fecha.

8.1.- Funciones de fecha

Como ya se ha dicho, PHP almacena las fechas como un número. Realmente utiliza el formato de fecha y hora de Unix, por lo que muchas funciones requieren pasar las fechas en este formato. Por ello hay otras muchas funciones que nos ayudan a crear fechas en ese formato a partir de un texto que represente a una fecha.

función	significado
time()	Devuelve la fecha y hora actual en el formato nativo (Unix) de PHP.
strftime(formato[,fecha])	Devuelve un texto que representa la fecha actual (o la que se indique como segundo parámetro) en el formato regional establecido con **setlocale**. El formato puede incluir estos símbolos: %a Día de la semana (tres letras) %A Día de la semana (completo) %d El día del mes con dos dígitos (del 01 al 31) %e El día del mes (de 1 a 31) %j Día del año, 3 dígitos (del 001 al 366) %u Día de la semana (del 1 al 7) %w Día de la semana (del 0 al 6) %W Número de semana del año, comenzando con el primer Domingo como la primera semana %b Nombre del mes con tres letras %B Nombre del mes completo %m Número del mes en dos cifras (del 01 al 12) %C Número de siglo (por ejemplo 21) %y Año en dos cifras %Y Año en cuatro cifras %H Hora en formato 24 horas (del 00 al 23) %k Hora en formato 24 horas (del 0 al 23) %I Hora en formato de 12 horas (01 al 12) %l La hora en formato de 12 horas (del 1 al 12) %M Minutos en dos cifras (del 01 al 59) %p 'AM' o 'PM' en MAYÚSCULAS basados en la hora dada %P 'am' o 'pm' en minúsculas basados en la hora dada %S Segundos en dos dígitos (del 00 al 59) %X Representación preferida de la hora basada en la configuración regional, sin la fecha
date(formato [,fecha]**)**	Da formato a la fecha. Si no se indica segundo parámetro, se toma la fecha actual. El formato puede llevar estos símbolos (se indican los más importantes): d Día del mes (del 01 al 31) D Día de la semana con trEs letras (en inglés) j Día del mes (del 1 al 31)

	l Día de la semana (en inglés) w Día de la semana (del 1 al 7) z Día del año (del 1 al 365 ó 366) F Nombre del més(en inglés) m Mes (del 01 al 12) M Mes con tres letras (en inglés) n Mes (del 1 al 12) t Número de días del mes dado (28,29,30 ó 31) L Verdadero si el año es bisiesto y Año con dos cifras Y Año con cuatro cifras a Símbolo AM en minúsculas A AM en mayúsculas h Hora en formato 12 horas (del 01 al 12) H Hora en formato 24 horas (del 00 al 23) g Hora en formato 12 horas (del 1 al 12) G Hora en formato 24 horas (del 0 al 23) i Minutos (del 00 al 59) s Segundos (del 00 al 59) u Microsegundos
mktime([hora[,minuto[,segundo[,mes[,día[,año[época]]]]]]]**)**	Crea una fecha y hora a partir de los parámetros indicados.Los que no se indiquen se toman de la fecha y hora actuales. El último parámetro indica con un 1 que estamos en horario de verano y con un -1 en el de invierno.
strptime(fecha, formato**)**	Devuelve un array asociativo que contiene como valores los datos de la fecha indicada. El formato cumple los posibles valores del parámetro ***formato*** de la función ***strftime*** explicada anteriormente.
local_time([fecha[,asociativo]] **)**	Devuelve la fecha actual (o la que se indique como primer parámetro) en forma de array en el que cada elemento contiene cada parte de la fecha y hora. Se puede indicar una fecha concreta y un valor verdadero como segundo parámetro para indicar que deseamos un array asociativo en lugar de escalar.
strtotime(texto**)**	Analiza el texto y devuelve la fecha correspondiente.Ejemplos de uso: <?php **echo strtotime**("now"), "\n"; **echo strtotime**("10 September 2000"), "\n"; **echo strtotime**("+1 day"), "\n"; **echo strtotime**("+1 week"), "\n"; **echo strtotime**("+1 week 2 days 4 hours 2 seconds"), "\n"; **echo strtotime**("next Thursday"), "\n"; **echo strtotime**("last Monday"), "\n"; ?>
checkdate(mes, dia, año**)**	Devuelve verdadero si la fecha indicada de esta forma es válida.
date_default_timezone_get()	Devuelve la zona horaria en uso (por ejemplo ***Europe/Berlin***)
date_default_timezone_set(zona**)**	Establece la nueva zona horaria. EL texto debe ser estándar, por ejemplo (***Europe/Berlin***).

Por ejemplo, podríamos tener el siguiente código:

```
//fecha/hora actual
echo "fecha1: ".date("d/m/Y")."<br>".PHP_EOL;

$dia=time(); //dia/hora actual UNIX
```

```php
$dia+=60*60*24*5; //sumo a la fecha 5 dias
echo "fecha+5dias: ".date("d/m/Y",$dia)."<br>".PHP_EOL;

//fecha-hora 17/7/2018 14:27:35
$dia=mktime(14,27,35,7,17,2018);
echo "fecha3: ".date("d/m/Y H:i:s",$dia)."<br>".PHP_EOL;

//formateo de fecha segun configuracion local
setlocale(LC_TIME, "esp_esp","es_ES"); //configuracion windows/linux
$dia=strtotime('03/24/2019'); //24/03/2019
echo "fecha4: ".strftime("%A, %d de %B de %Y",$dia)."<br>".PHP_EOL;
```

Que nos daría como salida

```
Fecha1: 22/02/2019
fecha+5dias: 27/02/2019
fecha3: 17/07/2018 14:27:35
fecha4: domingo, 24 de marzo de 2019
```

8.2.- Clases para fecha/hora

Se tiene la clase DateTime para representar una fecha/hora concreta. Esta clase nos aporta métodos para crearla, formatearla y modificarla (sumarle o restarle un periodo). En el caso de agregar un número de días, meses, etc se utiliza la clase DateInterval que permite establecer el valor.

Por ejemplo:

```php
//creo una fecha a partir de una cadena
$dia=new DateTime("18-01-2019");

//convierto la fecha a cadena
echo "fecha1: ".$dia->format("d/m/Y")."<br>".PHP_EOL;

//crear fecha usando método estatico
$dia=DateTime::createFromFormat("d/m/Y", "01/01/2019");
echo "fecha2: ".$dia->format("d/m/Y")."<br>".PHP_EOL;

//intervalo a sumar a la fecha
$intervalo=new DateInterval("P2M3DT5H");
//muestro años, meses, dias, horas, minutos en intervalo
echo "intervalo: ".$intervalo->format("a: %Y, m: %M, d: %D, h: %H, min:
%I, s: %S")."<br>".PHP_EOL;

//sumo el intervalo
$dia->add($intervalo);
echo "fecha3: ".$dia->format("d/m/Y")."<br>".PHP_EOL;
```

Daría como resultado

```
fecha1: 18/01/2019
fecha2: 01/01/2019
intervalo: a: 00, m: 02, d: 03, h: 05, min: 00, s: 00
fecha3: 04/03/2019
```

DateTime

Objeto que representa una fecha/hora concreta.

Además de acceder mediante el estilo orientado a objetos, se tiene un estilo basado en procedimientos, de forma que por cada método existe un procedimiento equivalente.

Por ejemplo, el método modify de DateTime

```
Estilo orientado a objetos
public DateTime::modify ( string$modify ) : DateTime

Estilo por procedimientos
date_modify( DateTime$object , string$modify ) : DateTime
```

método	Significado
public__construct ([string $time = "now" [, DateTimeZone $timezone = **NULL**]])	Devuelve un nuevo objeto DateTime. $Time es una cadena que representa una fecha/hora válida según un formato válido (mm/dd/YYYY, YYYY/mm/dd, YYYY-mm-dd, dd.mm.YYYY, dd-mm-YYYY, HH:MM:ss) o una hora UNIX válida "@94553555889"
Public **DateTime::add** (DateInterval $interval) : DateTime	Añade al objeto una cantidad de días, meses, años, horas, minutos y segundos indicados por el intervalo
public static **DateTime::createFromFormat** (string $format , string $time [, DateTimeZone $time-zone]) : DateTime	Este método estático devuelve l fecha indicada en $time y codificada según el formato $format. El formato es idéntico al de la función date()
public **DateTime::modify** (string $modify) : DateTime	Altera la marca temporal de un objeto DateTime aumentando o disminuyendo en un formato aceptado por strtotime().
public **DateTime::setDate** (int $year , int $month , int $day) : DateTime	Asigna al objeto como fecha la indicada por los parámetros indicados
public **DateTime::setTime** (int $hour , int $mi-nute [, int $second = 0]) : DateTime	Asigna al objeto como hora la indicada por los parámetros.
public **DateTime::sub** (DateInterval $interval) : DateTime	Resta al objeto una cantidad de días, meses, años, horas, minutos y segundos indicados por el intervalo
public **DateTime::diff** (DateTimeInterface $datetime2 [, bool $absolute = false]) : DateInterval	Devuelve un objeto intervalo que representa la diferencia entre el objeto DateTime y uno indicado como parámetro. Si absolute vale true, se fuerza el resultado para que sea positivo
public **DateTime::format** (string $format) : string	Devuelve una cadena que representa la fecha/hora según un formato. El formato es idéntico al usado con la función date()

DateInterval

Representa un intervalo de fechas.

Método/propiedad	Significado
$y, $m, $d, $h, $i, $s	Propiedades públicas que almacenan respectivamente el número de años, mese, día, horas, minutos y segundos del intervalo
public **DateInter-val::__construct** (string	Se crea un objeto DateInterval a partir de una cadena que representa el intervalo. El formato empieza con la letra *P*, de periodo. Cada periodo de duración está

`$interval_spec`)	representado por un valor de tipo integer seguido de un indicador de periodo. Si contiene elementos de hora, esa parte de la especificación estará precedida por una letra *T*. Los indicadores de periodo son: Y años M meses D días W semanas H horas M minutos S segundos. Por ejemplo "P2Y3MT50M" representa un periodo de 2 años, 3 meses y 50 minutos. "PT12S" representa un periodo de 12 segundos.
public static **DateInterval::createFromDateString** (string`$time`) : DateInterval	Permite crear un intervalo a partir de una cadena que representa un valor relativo. El valor aceptado es el mismo del usado en la función strtotime. Ej "1 day", "2 weeks", "1 year + 1 day".

public **DateInterval::format** (string `$format`) : string	Devuelve una cadena con el intervalo formateado. Se indica unos caracteres que representan la parte a formatear precedidos de %.
	% literal %
	Y Años, numérico, al menos 2 dígitos empezando con 0
	y Años, numérico
	M Meses, numérico, al menos 2 dígitos empezando con 0
	m Meses, numérico
	D Días, numérico, al menos 2 dígitos empezando con 0
	d Días, numérico
	a Número total de días como resultado de una operación con DateTime::diff(), o de lo contrario *(unknown)*
	H Horas, numérico, al menos 2 dígitos empezando con 0
	h Horas, numérico
	I Minutos, numérico, al menos 2 dígitos empezando con 0
	i Minutos, numérico
	S Segundos, numérico, al menos 2 dígitos empezando con 0
	s Segundos, numérico
	R Signo "-" cuando es negativo, "+" cuando es positivo
	r Signo "-" cuando es negativo, vacío cuando es positivo
	Por ejemplo:
	`$intervalo = new DateInterval('P2Y4DT6H8M');`
	`echo $intervalo->format('%d días');`
	`El resultado sería 4 días`

9.-CLASES E INSTANCIAS

A partir de PHP 5, el modelo de objetos ha sido reescrito para permitir un mejor rendimiento y con más características. Este fue un cambio importante a partir de PHP 4. PHP 5 tiene un modelo de objetos completo.

Entre las características de PHP5 están la inclusión de la visibilidad, las clases abstractas y clases y métodos finales, métodos mágicos adicionales, interfaces, clonación y tipos sugeridos.

PHP trata ***los objetos como referencias o manejadores***, lo que significa que cada variable contiene una referencia de objeto en lugar de una copia de todo el objeto.

La definición básica de clases comienza con la palabra clave ***class***, seguido por un nombre de clase, continuado por un par de llaves que encierran las definiciones de las propiedades y métodos pertenecientes a la clase.

Una clase puede tener sus propias constantes, variables (llamadas "propiedades"), y funciones (llamadas "métodos").

```
/**
 * Clase prueba usada para mostrar el funcionamiento de los
 * distintos elementos
 */
Class ClasePrueba{
     //definición de constantes
     public const PI=2.14;

     //definición de propiedad
     public int $propPublica=1;
     private string $_propPrivada="hola";
     protected float $_propProtegida=1.234;

     //definición de propiedad estática
     public static $propEstatica=array(1,2,3);

     //definición de método
     public function metodo(){
          //acceso a las propiedades desde el propio objeto
          $this->propPublica=13;
          $this->_propPrivada=100;
          $this->_propProtegida="adios";

          //acceso a la propiedad estática
          self::$propEstatica[0]="otra";

          return true;
     }
}

//se crea una instancia de la clase prueba
$instancia=new ClasePrueba();

//accede a la propiedad pública de la instancia
$instancia->propPublica=50;

//se accede a la propiedad de clase (estática)
```

```
ClasePrueba::$propEstatica[1]=10;
```

Con la sintaxis anterior se define la clase. Por otro lado, en un momento determinado, se trabajará con una variable de esa clase, lo que se llama una instancia.

Para crear una instancia se usa la palabra clave *new*. Un objeto siempre se creará a menos que el objeto tenga un constructor que arroje una excepción en caso de error.

```
//se crea una instancia de la clase prueba
$instancia=new ClasePrueba();
```

Cuando se define la clase se pueden hacer referencia a las propiedades y métodos de la instancia mediante la pseudo-variable *$this*. Está disponible cuando un método es invocado dentro del contexto de un objeto. *$this* es una referencia del objeto que invoca.

Por ejemplo, en la definición anterior de la clase, $this->propPublica hace referencia a la propiedad “propPublica” de la instancia.

Igual que con las funciones podemos definirle documentación tipo javadoc usando /** */

9.1.- Propiedades y constantes

Las variables pertenecientes a clases se llaman "**propiedades**". También se les puede llamar usando otros términos como "atributos" o "campos”. Éstas se definen usando una de las palabras clave *public*, *protected*, o *private*, seguido de una declaración normal de variable. Esta declaración puede incluir una inicialización. La inicialización debe ser mediante un valor constante, es decir, debe poder ser evaluada en tiempo de compilación y no debe depender de información en tiempo de ejecución para ser evaluada.

```
public int $propPublica=1;
private string $_propPrivada="hola";
protected float $_propProtegida=1.234;
```

El acceso a las propiedades se realiza dentro de la clase mediante la pseudo-variable *$this*

```
$this->propPublica=13;
$this->_propPrivada=100;
$this->_propProtegida="adios";
```

Fuera de la clase se accedería a las propiedades usando -> y referenciando a la instancia.

```
//accede a la propiedad pública de la instancia
$instancia->propPublica=50;
```

Se usa una convención en cuanto al nombrado de las propiedades. Todas las propiedades privadas y protegidas llevan _ al principio. Además, cuando en la propiedad se hace referencia a varias palabras, la primera letra de cada palabra se pone en mayúscula.

Es posible aplicar el tipado en la definición de propiedades.

En el caso anterior hemos definido las propiedades con el tipo

```
public int $propPublica=1;
private string $_propPrivada="hola";
protected float $_propProtegida=1.234;
```

Y tenemos un objeto ($obj) de la clase, nos daría un error esta sentencia

```
//propiedad $propPublica es de tipo entero,
//por lo que falla si queremos asignar un valor de otro tipo
$obj->propPublica="esto esta mal";
```

Todo lo indicado anteriormente es relativo a las propiedades normales (propiedades de instancia, propiedades no estáticas). Además de ellas se pueden definir propiedades estáticas.

Las propiedades estáticas son propiedades sobre la clase, es decir, se acceden a ellas sin necesidad de instanciar una variable.

Una propiedad (o método) estática se crea anteponiendo a la definición de la misma la palabra *static*.

```
//definición de propiedad estática
public static $propEstatica=array(1,2,3);
```

Un método o una propiedad estática puede accederse desde dentro de la clase en la que se define o desde fuera de la misma. En el primer caso se usa la palabra clave *self*.

```
self::$propEstatica[0]="otra";
```

Si nos queremos referir a la propiedad estática desde fuera de la definición de la clase se usa directamente el nombre de la clase. Es posible además usar una variable para referirse a la clase o incluso usar una instancia.

```
ClasePrueba::$propEstatica[1]=10;

$instancia::$propEstatica[1]=10;

$clase="ClasePrueba";
$clase::$propEstatica[1]=10;
```

Como se observa en los ejemplos anteriores se usan los ::. Este operador es conocido como Operador de Resolución de Ámbito. Se usa para acceder a los elementos estáticos, constantes y para sobreescribir propiedades o métodos de una clase.

En el interior de una clase se usa este operador junto con self (referencia a la propia clase) o parent (referencia a la clase padre). También se podría usar junto con el nombre de una clase si está es un antecesor.

Es posible definir valores **constantes** en función de cada clase manteniéndola invariable. Las constantes se diferencian de variables comunes en que no utilizan el símbolo *$* al declararlas o usarlas.

El valor debe ser una expresión constante, no (por ejemplo) una variable, una propiedad, un resultado de una operación matemática o una llamada a una función.

Igual para que con las propiedades, se les puede indicar una visibilidad a las constantes (public, private o protected).

La definición en la clase será:

```
//definición de constantes
public const PI=2.14;
```

Para usarlo podríamos acceder igual que con las propiedades estáticas.

```
echo $instancia::PI;
echo ClasePrueba::PI;
```

9.2.- Constructores/destructores

Un constructor es un método que se llama cuando se crea una instancia de una clase. A través del constructor se puede inicializaciones de las propiedades, llamar a funciones externas, …

Un constructor es un método dentro de la definición de la clase de la forma __construct.

```
class Objeto
{
    private ?string $_nombre;
    protected ?int $_numero;

    //constructor
    public function __construct(string $nom)
    {
          //realizo operaciones al crear el objeto
          $this->_nombre=$nom;
          $this->_numero=0;
          echo "se entra al constructor<br>";
    }
```

```
    //destructor
    public function __destruct()
    {
          //hago operaciones finales como cerrar enlaces BD.
          unset($this->_nombre);
          echo "se entra al destructor<br>";
    }
}
```

Cuando tenemos herencia la clase hija puede:

- No definir su propio constructor. En este caso al instanciar una variable de la clase se llama al constructor de la clase base.
- Definir un constructor propio. En este caso al instanciar una variable se llamará al constructor de la clase y si queremos llamar al de la clase padre usaremos **parent::__construct(….)** dentro del constructor.

```
Class Mesa extends Objeto
{
    //define sus propias propiedades
   private $_numPatas;

   //crea su propio constructor
   public function __construct($nom)
   {  //se puede llamar al constructor de la clase padre
        parent::__construct($nom);

        //puede acceder a las propiedades protegidas
        $this->_numero=1;

        $this->_numPatas=4;

   }
}
```

Al definir el constructor es posible definir la **"promoción" de parámetros**. La promoción consiste en la definición de parámetros en el constructor de forma que en la clase se cree propiedades de forma automática que se llaman igual. Para ello basta con incluir en la definición del parámetro del constructor una indicación de la visibilidad (public, prívate o protected)

```
class ClaseConPromocion
{
   private string $_propPrivada;

   public function __construct(string $privada, protected int
                  $_propProtegida)
   {
        $this->_propPrivada=$privada;
    }
}
```

```php
//$obj sera una instancia de la clase ClaseConPromocion
//con dos propiedades $_propPrivada y $_propProtegida
$obj=new ClaseConPromocion("datos",25);
```

A diferencia de otros lenguajes, PHP sólo permite un constructor. Si queremos mas, es común la definición de métodos estáticos de creación. Se ha incluido un nuevo pseudotipo de retorno static, que indica que se devuelve una instancia de la propia clase.

Por ejemplo, podríamos tener:

```php
class Prueba {

   private ?int $_codigo;
   private ?string $_nombre;

   //el constructor es privado
   private function __construct(?int $codigo = null, ?string $nombre =
               null) {
     $this->_codigo = $codigo;
     $this->_nombre = $nombre;
   }

   //constructor estatico publico
   public static function desdeDatos(int $codigo, string $nombre): static
   {
     //creo el objeto usando static
     $obj = new static($codigo, $nombre);
     return $obj;
   }

   //constructor estatico publico
   public static function desdeJson(string $json): static
   {
     $datos = json_decode($json);
     return new static($datos->codigo, $datos->nombre);
   }

}

$datos=["codigo"=>12,"nombre"=>"prueba2"];
$datosJSon=json_encode($datos);

//no hay constructor público, por lo que tengo que
//llamar a las funciones de creación estaticas
$p1 = Prueba::desdeDatos(5, 'Objeto');
$p2 = Prueba::desdeJson($datosJSon);
```

PHP 5 introduce un concepto de destructor similar al de otros lenguajes orientados a objetos, tal como C++. El método destructor será llamado tan pronto como no haya otras referencias a un objeto determinado, o en cualquier otra circunstancia de finalización.

Como los constructores, los destructores padre no serán llamados implícitamente por el motor. Para ejecutar un destructor padre, se deberá llamar explícitamente a **parent::__destruct()** en

el interior del destructor. También como los constructores, una clase child puede heredar el destructor de los padres si no implementa uno propio.

El destructor será invocado aún si la ejecución del script es detenida usando exit(). Llamar a exit() en un destructor evitará que se ejecuten las rutinas restantes de finalización.

El proceso de creación/destrucción se puede ver en el siguiente código

```
//creo el objeto.
//en este momento se llama al constructor.
$obj=new Objeto("nombre");


//creo otra referencia al objeto
echo "creo otra referencia<br>";
$obj1=$obj;

//borro una referencia
echo "borro obj<br>";
unset($obj);

//al borrar todas las referencias se llamaría al destructor
echo "borro obj1<br>";
unset($obj1);
```

Que generaría como salida:

```
se entra al constructor
creo otra referencia
borro obj
borro obj1
se entra al destructor
```

9.3.- Herencia y visibilidad

La **herencia** es un principio de programación bien establecido y PHP hace uso de él en su modelado de objetos. Este principio afectará la manera en que muchas clases y objetos se relacionan unas con otras.

En una clase se pueden definir tres niveles de visibilidad: publico, protegido y privado.

- Un método/propiedad público (public) será accesible desde cualquier lugar.
- Un elemento definido como protegido (protected) será accesible solamente desde las clases descendientes y la propia clase.
- Un elemento definido como privado (prívate) será solamente accesible desde la propia clase.

Así, cuando se extiende una clase, la subclase hereda todos los métodos públicos y protegidos de la clase padre. A menos que una clase sobrescriba esos métodos, mantendrán su funcionalidad original.

Esto es útil para la definición y abstracción de la funcionalidad y permite la implementación de funcionalidad adicional en objetos similares sin la necesidad de reimplementar toda la funcionalidad compartida.

Para indicar que una clase hereda de otra se usa la palabra extends.

Si en una clase hija se redefine un método éste sobreescribirá el definido en la clase padre. Para poder acceder al de la clase padre se puede usar la palabra clave parent.

```
class Objeto
{
   private string $_nombre;
   protected int $_numero;

   //constructor
   public function __construct(string $nom)
    {
      //realizo operaciones al crear el objeto
      $this->_nombre=$nom;
      $this->_numero=0;
      echo "se entra al constructor<br>";
    }

    //destructor
    public function __destruct()
    {
      //hago operaciones finales como cerrar enlaces BD.
      unset($this->_nombre);
      echo "se entra al destructor<br>";
    }

    //método público
    Public function escribeTexto()
    {
      echo "es un objeto de nombre {$this->_nombre}";
    }
}

class Mesa extends Objeto
{
    //define sus propias propiedades
    private$_numPatas;

    //crea su propio constructor
    public function __construct(string $nom)
    {  //se puede llamar al constructor de la clase padre
      parent::__construct($nom);

      //puede acceder a las propiedades protegidas
      $this->_numero=1;

      $this->_numPatas=4;

    }
```

```
    //sobreescribe un método de la clase base
    public function escribeTexto()
    {
      //a través de parent puede acceder al método de la
      //clase base
      parent::escribeTexto();

      echo "mesa de {$this->_numPatas} patas";
    }
}
```

En PHP es posible definir **clases y métodos abstractos**. Las clases definidas como abstract no se pueden instanciar y cualquier clase que contiene al menos un método abstracto debe ser definida como abstract. Los métodos definidos como abstractos simplemente declaran la estructura del método, pero no pueden definir la implementación.

Cuando se hereda de una clase abstracta, todos los métodos definidos como abstract en la definición de la clase padre deben ser redefinidos en la clase hija; además, estos métodos deben ser definidos con la misma visibilidad (o con una menos restrictiva). Por ejemplo, si el método abstracto está definido como protected, la implementación de la función puede ser redefinida como protected o public, pero nunca como private. Por otra parte, las estructuras de los métodos tienen que coincidir; es decir, la implicación de tipos y el número de argumentos requeridos deben ser los mismos. Por ejemplo, si la clase derivada define un parámetro opcional, mientras que el método abstracto no, no habría conflicto con la estructura del método.

```
//clase base abstracta
abstract class Objeto
{
    private string $_nombre;
    protected int $_numero;

    //constructor
    public function __construct(string$nom)
    {
        //realizo operaciones al crear el objeto
        $this->_nombre=$nom;
        $this->_numero=0;
    }

    //método abstracto
    abstract public function escribeTexto();

}

class Mesa extends Objeto
{
    //define sus propias propiedades
    prívate int $_numPatas=5;

    //al ser la clase base abstracta, y el método abstracto
    //tengo que definirlo en la clase hija
    public function escribeTexto()
```

```
    {
          echo "mesa de {$this->_numPatas} patas";
    }
}
```

En PHP se tiene la palabra clave **final.** Impide que las clases hijas sobrescriban un método, antecediendo su definición con *final*. Si la propia clase se define como final, entonces no se podrá heredar de ella.

Por ejemplo, podríamos haber definido

```
abstract class Objeto
{
    private string $_nombre;
    protected int $_numero;

    //constructor
    public function __construct(string $nom)
    {
          //realizo operaciones al crear el objeto
      $this->_nombre=$nom;
      $this->_numero=0;
    }

    //método abstracto
    abstract public function escribeTexto();

    //método final escribe numero
    public final function escribeNumero()
    {
        echo "el numero es {$this->_numero}";
    }
}
```

En este caso la clase Mesa no podría redefinir el método escribeNúmero.

La herencia en PHP, al igual que en muchos otros lenguajes, es simple. Esto es, una clase hija solo puede heredar (extends) de una clase padre. Sin embargo, en muchos casos debemos poder "simular" una herencia múltiple. Esto se realiza mediante **las interfaces**.

Las interfaces de objetos permiten crear código con el cual especificamos qué métodos deben ser implementados por una clase, sin tener que definir cómo estos métodos son manipulados.

Las interfaces son definidas utilizando la palabra clave interface, de la misma forma que con clases estándar, pero sin métodos que tengan su contenido definido.

Todos los métodos declarados en una interfaz deben ser públicos, ya que ésta es la naturaleza de una interfaz.

```
//creo la interfaz iFabricable
interface iFabricable
{
```

```
    //puede definir sus propias constantes
    Const PATAS_DEFECTO=4;

    //defino el método forma Fabricacion.
    //las clases que implementen esta intefaz
    //tendrán que redefinir el método
    public function formaFabricacion();
}

//la clase Mesa además deheredar de Objeto
//implementa la interface iFabricable
class Mesa extends Objeto implements iFabricable
{
    //definesuspropiaspropiedades
    private $_numPatas=5;

    //al ser la clase base abstracta,y el método abstracto
    //tengo que definirlo en la clase hija
    public function escribeTexto()
    {
        echo "mesa de {$this->_numPatas} patas";
    }

    //hay que definir el método de la interface
    public function formaFabricacion()
    {
       echo "primero se corta la madera<br>";
       echo "segundo se hacen las patas<br>";
    }

}
```

Para implementar una interfaz, se utiliza el operador implements. Todos los métodos en una interfaz deben ser implementados dentro de la clase; el no cumplir con esta regla resultará en un error fatal. Las clases pueden implementar más de una interfaz si se deseara, separándolas cada una por una coma.

Es posible tener constantes dentro de las interfaces. Las constantes de interfaces funcionan como las constantesdeclases excepto porque no pueden ser sobrescritas por una clase/interfaz que las herede.

9.4.- Sobrecarga, iteración y métodos mágicos

En PHP es posible tener propiedades y métodos creados “dinámicamente”, es decir, no estaban definidos al crear la clase. A esto se le llama **sobrecarga.** En PHP el concepto de sobrecarga es distinto del de otros lenguajes: la sobrecarga tradicionalmente ofrece la capacidad de tener múltiples métodos con el mismo nombre, pero con un tipo o un número distinto de parámetros.

Para la creación de elementos dinámicos (sobrecargados) se usan una serie de métodos mágicos. Estos métodos son invocados cuando se interactúa con propiedades o métodos que no se

han declarado o que no son visibles en el ámbito activo.

Para la **sobrecarga de propiedades** se tienen los métodos mágicos:

- *public function* ***__set****(string* `$nombre`*, mixed* `$valor`*):void* Se ejecuta al escribir datos sobre propiedades inaccesibles. Es llamado de forma automática cuando se actualiza el valor de una propiedad que no es visible.
- *Public function* ***__get****(string*`$nombre`*):mixed* Se utiliza para consultar datos a partir de propiedades inaccesibles. Se llama de forma automática cuando se lee el valor de una propiedad que no es visible.
- *Public function* ***__isset****(string* `$nombre`*):bool* Se lanza al llamar a isset() o a empty() sobre propiedades inaccesibles.
- *public function* ***__unset****(string* `$nombre`*):void* Se invoca cuando se usa unset() sobre propiedades inaccesibles.

$nombre representaría la variable sobrecargada.

La sobrecarga de propiedades se usa para dar acceso a propiedades que no son visibles (propiedades privadas o protegidas) o para crear nuevas propiedades según se necesite.

Una cosa a tener en cuenta es que la sobrecarga está habilitada por defecto. Es decir, automáticamente se crean propiedades públicas sobrecargadas cuando se hace la asignación de una nueva propiedad que no existe.

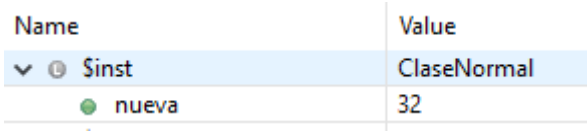
```php
class ClaseNormal{

}

$inst=new ClaseNormal();
$inst->nueva=32; //se sobrecarga la instancia
//se crea automaticamente la propiedad publica nueva

echo $inst->nueva;
```

Al crear la instancia $inst no existía la propiedad nueva. Cuando he hecho una asignación, por la sobrecarga, se ha creado la propiedad pública $nueva. Esto se puede observar cuando



paramos la ejecución y observamos la salida del depurador

Para evitar este comportamiento, se puede definir __set para que no permita la creación de propiedades sobrecargadas.

```php
/**
 * clase en la que se deshabilita la sobrecarga de propiedades
```

```
 */
class ClaseNormal{
     public function __set(string $nombre,mixed $valor){
         throw new Exception ('No existe la propiedad '.$nombre);
     }
     public function __get(string $nombre){
          throw new Exception ('No existe la propiedad '.$nombre);
     }
     public function __isset(string $nombre):bool{
          return false;
     }
 }

$inst=new ClaseNormal();
$inst->nueva=32; //daría una excepcion
```

Normalmente las variables sobrecargadas se gestionan mediante un atributo privado (suele ser un array) que nos sirve para controlar las que tenemos definidas.

```
Class ClaseDinamica
{
   //variable privada para mantener las propiedades
   //creadas de formadinámica
   private array $_datos=array();

   //variable pública, se accede directamente a ella
   public int $varPublica=10;


   //se llamará de forma automática cuando se lea
   //una propiedad y no sea visible.
   public function __get(string $nombre)
   {
     if(isset($this->_datos[$nombre]))
         return $this->_datos[$nombre];
       else
          //si no existe lanza una excepción
         throw new Exception('No existe la propiedad.');
    }

   //se llamará de forma automática cuando se escriba
   //una propiedad y no sea visible
   public function __set(string $nombre,mixed $valor)
   {
     //puede filtrar que propiedades están o no disponibles
     //no permito propiedades cuyo nombre tenga una
     //longitud de mas de 10 caracteres
     if (mb_strlen($nombre)>10)
        throw new Exception('No existe la propiedad.');

     //creo/actualizo la variable dinámica.
     $this->_datos[$nombre]=$valor;

   }
```

```
   //se llamará automáticamente al ejecutar isset o
   //empty
   public function __isset(string $nombre):bool
    {
       return(isset($this->_datos[$nombre]));
    }

//se llamará automáticamente al ejecutar
//unset
   public function __unset(string $nombre)
   {
      unset($this->_datos[$nombre]);
   }

}

$obj=new ClaseDinamica();

//se accede directamente a la propiedad al ser publica.
$obj->varPublica=12;
echo $obj->varPublica;

//la propiedad nueva no está definida en la clase.
//se llama automáticamente a __set
$obj->nueva=10;

//la propiedad nueva no está definida en la clase.
//se llama automáticamente a __get
echo $obj->nueva;


//se comprueba si existe la propiedad nueva.
//como no es visible se llama de forma automática
//a __isset
if (isset($obj->nueva))
    echo "la propiedad nueva existe";

//se elimina la propiedad nueva
unset($this->nueva);

//accede a una propiedad que no se ha inicializado antes
//se lanza una excepcion.
echo $obj->noexiste;
```

No es posible sobrecargar variables en el contexto estático solamente a nivel de objeto.

Una cosa también a tener en cuenta es que debido a la forma en que PHP procesa el operador de asignación, el valor que devuelve __set() se ignora. Del mismo modo, nunca se llama a __get() al encadenar asignaciones, por lo que esto no funcionaría

```
$a=$obj->b=8;
$obj->b+=5;
```

Para la **sobrecarga de métodos** se tienen:

- *public function* **__call(***string `$nombre`, array `$argumentos`***):mixed**es lanzado al invocar un método inaccesible en un contexto de objeto.
- *public static function* **__callStatic(***string `$nombre`, array `$argumentos`***):mixed**es lanzado al invocar un método inaccesible en un contexto estático.

```
class ClaseDinamica
{
   public function metodoExiste()
    {
       echo "este método existe";
    }
   //secargaríasinoexisteelmétodovisible
   public function __call(string $nombre,array $argumentos):mixed
    {
       $cadena=implode(',',$argumentos);

       echo"se ha llamado al metodo $nombre con argumentos $cadena";
    }

   //se cargaría si no existe el método estático
   //visible
   public static function __callStatic(string $nombre,array $argumen-
                   tos):mixed
   {
      $cadena=implode(',',$argumentos);

      echo"se ha llamado al metodo estático $nombre con argumentos $cade-
na";
    }
}

$obj=new ClaseDinamica();

//el metodo existe se llama directamente a el
$obj->metodoExiste();


//el método no existe y es a nivel de objeto
//se llama de forma automática a __call
$obj->metodoNoExiste();

//el método no existe y es a nivel de clase
//se llama de forma automática a __callStatic
ClaseDinamica::metodoNoExiste();
```

Con PHP se puede **iterar** a través de una lista de elementos de los objetos mediante una sentencia foreach. Por defecto, todas las propiedades visibles serán utilizadas para la iteración. Así si iteramos dentro de una clase se podría ver todas las propiedades mientras que si iteramos el objeto fuera del mismo sólo se podrían ver las propiedades públicas

```php
class ClaseAITerar
{
    public int $propPublica=1;
    private int $_propPrivada=2;
    protected int $_propProtegida=3;

    public function iteracionDentro()
    {
        //se tiene acceso a todas las propiedades
        foreach($this as $prop=>$valor)
        {
            echo"propiedad $prop valor $valor<br>";
        }
    }
}

$obj=new ClaseAIterar();

echo "Se itera desde la clase<br>";
$obj->iteracionDentro();

echo "Se itera desde fuera de la clase";
foreach($obj as $prop=>$valor)
{
    //se tiene acceso a todas las publicas
    echo"propiedad $prop valor $valor<br>";
}
```

Para dar un paso más, se puede implementar la **interfaz Iterator**. Esto permite al objeto decidir cómo será iterado y qué valores estarán disponibles en cada iteración.

Para implementar la interfaz Iterator se tienen que redefinir los métodos:

- *abstract public function* ***current****(void):mixed* Devuelve el elemento actual.
- *abstract public function* ***key****(void):mixed* Devuelve la clave del elemento actual
- *abstract public function* ***next****(void):void* Avanza al siguiente elemento
- *abstract public function* ***rewind****(void):void* De vuelve el puntero a la primera posición. Es llamado al iniciar el foreach.
- *abstract public function* ***valid****(void):bool*. Este método se llama para comprobar si la posición actual es válida. Se llama de forma automática tras ejecutar next o rewind.

Para el caso anterior podríamos hacer accesible las propiedades privadas y protegidas usando un foreach.

```php
classClaseAITerar implements Iterator
{
    public int $propPublica=1;
    private int $_propPrivada=2;
    protected int $_propProtegida=3;
```

```
    //servirá como puntero para indicar la posición
    //actual
    private$_posicion;

    public function iteracionDentro()
    {
        //se tiene acceso a todas las propiedades
        foreach($this as $prop=>$valor)
        {
            echo "propiedad $prop valor $valor<br>";
        }
    }

    //pone el puntero al valor inicial
    public function rewind()
    {
        $this->_posicion=1;
    }

    //devuelve el contenido correspondiente a la posición actual
    public function current()
    {
        switch($this->_posicion)
        {
            case1: return $this->propPublica;
                    break;
            case2: return $this->_propPrivada;
                    break;
            case3: return $this->_propProtegida;
                    break;

        }
    }

    //devuelve la clave correspondiente a la posicion actual
    public function key()
    {
        switch($this->_posicion)
        {
            case1: return"propPublica";
                    break;
            case2: return"propPrivada";
                    break;
            case3: return"propProtegida";
                    break;

        }
    }

    //mueve el puntero a la siguiente posicion
    public function next()
    {
        $this->_posicion++;
    }

    //comprueba si la posición actual es valida
    public function valid()
```

```
    {
        return($this->_posicion<=3);
    }
}

$obj=new ClaseAIterar();

echo "Se itera desde la clase<br>";
$obj->iteracionDentro();

echo "Se itera desde fuera";
foreach($obj as $prop=>$valor)
{
    //se tiene acceso a todas las propiedades
    echo "propiedad $prop valor $valor<br>";
}
```

También podemos usar los generadores para facilitar la iteración sobre valores definidos por el usuario. Básicamente un generador es un método fácil de implementar iteradores sin la sobrecarga o complejidad de crear una clase que implemente la interfaz Iterator. En su lugar, se puede escribir una función generadora, que es igual que una función normal, con la salvedad de que en vez de hacer un solo return, un generador puede invocar **yield** tantas veces como necesite para proporcionar valores por los que iterar.

```
function mi_rango(int $inicio, int $fin, int $paso = 1)
{
  if ($inicio < $fin)
    {
      if ($paso <= 0)
        {
          throw new LogicException('El paso debe ser positivo');
        }
      for ($i = $inicio; $i <= $fin; $i += $paso)
        {
          yield (1000+$i)=>($i*2);
        }
    }
  else
    {
      if ($paso >= 0)
        {
          throw new LogicException('El paso debe ser negativo');
        }
      for ($i = $inicio; $i >= $fin; $i += $paso)
        {
          yield (100+$i)=>($i*3);
        }
    }
}

//llamada al generador
foreach(mi_rango(1,10,2) as $clave=>$valor)
```

```
{
  echo "$clave => $valor<br>".PHP_EOL;
}

foreach(mi_rango(10,1,-2) as $clave=>$valor)
{
  echo "$clave => $valor<br>".PHP_EOL;
}
```

En el caso del generador mi_rango, se genera una clave-valor en cada iteración y un objeto de tipo Generator para mantener el estado.

Yield se puede usar para devolver el valor o el par clave=>valor.

```
yield $clave=>$valor;
yield $valor;
```

En PHP se definen algunos **métodos mágicos adicionales**. Todos comienzan con __ y son llamados de forma automática. Además de los indicados en los apartados anteriores nos podemos encontrar con:

- *public string* ***__toString****(void)* permite a una clase decidir cómo comportar se cuando se trata como una cadena. Por ejemplo, lo que *echo $obj;* mostraría. Este método debe devolver un string

```
class MiClase
{
    public string $nombre;

    public function __construct(string $nom)
    {
        $this->nombre=$nom;
    }

    public function __toString():string
    {
        return "Objeto de clase".__CLASS__." con nombre {$this->nombre}";
    }
}

$obj=new MiClase("profesor");

echo $obj;
```

- *void****__clone****(void)*. Permite crear una copia de un objeto. Al realizar la copia podemos modificar los valores del nuevo objeto.
  Su uso sería:

```
$copia=clone $objeto;
```

```
class MiClase
{
   public string $nombre;
   private int $numero=1;

   public function __construct($nom)
   {
        $this->nombre=$nom;
    }

   public function __toString()
   {
      return"Objeto de clase".__CLASS__.
                  " con nombre{$this->nombre}".
                  " con numero {$this->numero}";
    }

   //método mágico para el clonado
   publicfunction__clone()
    {
        //esto es sobre el objeto clonado
        //realizo todos los cambios que necesite
        $this->numero++;
    }
}

$obj=new MiClase();

//se clona el objeto
$obj1=clone $obj;

echo $obj1;
```

10.- ENUMERACIONES

10.1.- Introducción

Las **enumeraciones** o **enums** permiten a los usuarios definir tipos personalizados que están limitados a un número concreto de posibles valores.

Las enumeraciones aparecen en muchos lenguajes con diferencias en cuanto a sus características. En concreto, en PHP, los enums son un tipo especial de objeto. Un enum es una clase, donde sus posibles casos son todos instancias únicas de la clase, esto es, los casos de un enum son objetos válidos y pueden usarse en cualquier lugar en el que pueda usarse un objeto (incluyendo validación de tipos).

10.2.- Definición.

Las enumeraciones son similares a las clases y comparten los mismos espacios de nombres con clases, interfaces, etc.

Un enum define un nuevo tipo fijo y limitado a un número concreto de posibles valores. Se define con la siguiente sentencia

```
enum Nombre
{
    case Valor1;
    case Valor2;
    ....

}
```

Como ejemplo, tenemos el siguiente código

```
//defino el tipo enum Estado
enum Estado
{
    case Estudiante;
    case Trabajador;
    case Desempleado;
    case Jubilado;

    //es una clase y se pueden definir sus propios métodos
    function describete():string{
        return $this->name;
    }
}

//función en la que se tiene un parámetro de tipo Enun Estado
function dameTextoEstado(Estado $est):string
{
    //obtengo el nombre del caso
    return $est->name;
}
```

```
//asigno a una variable un valor de Estado
$valor=Estado::Estudiante;

//llamo a la función
echo dameTextoEstado($valor)."<br>".PHP_EOL;

//o al método
echo $valor->describete()."<br>".PHP_EOL;
```

El enum Estado es una clase con cuatro valores: Estado::Estudiante, Estado::Trabajador, Estado::Desempleado y Estado::Jubilado (cuatro instancias de la clase Estado). Estos valores pueden asignarse a variables directamente o usarse el tipo en una declaración (por ejemplo en la función).

Cada caso se define como un objeto único con la propiedad de solo lectura **name** que indica el nombre del caso. Al ser un objeto, no se asigna un equivalente escalar a cada caso como en otros lenguajes (Estado::Estudiante no es 0) **ni se pueden comparar** casos entre sí (Estado::Estudiante > Estado::Jubilado siempre devuelve false).

```
//asigno el mismo objeto a diferentes variables
$v1=Estado::Desempleado;
$v2=Estado::Desempleado;

//son el mismo objeto, resultado true
if ($v1===$v2)
    echo"son iguales";

//son instancias de la clase Estado
if ($v1 instanceof Estado)
    echo"Es una instancia de Estado";
```

En las enumeraciones se tiene el método estático cases() que devuelve un array con todos los casos posibles de una enumeración.

```
//devuelve un array con todos los casos
//posibles
$casosPosible=Estado::cases();
```

El enum Estado es un **Enum Puro**, es decir, una enumeración en la que no se definen datos relacionados.

10.3.- Backed Enums.

Como se ha dicho, un enumerado no se "iguala" a un escalar. Sin embargo, PHP permite definir los **backed enums** pudiendo asignar a cada caso un valor escalar que puede ser un entero o string. Las backed enum se definen indicando el tipo escalar tras el nombre

```
enum Nombre: int | string
{

}
```

Siguiendo con el ejemplo de Estado, podemos definir un backed enum:

```
//defino el tipo backed enum Estado
enum Estado:int
{
    case Estudiante = 1;
    case Trabajador = 2;
    case Desempleado = 3;
    case Jubilado = 4;

    //es una clase y se pueden definir sus propios métodos
    function describete():string{
        return $this->name;
    }
}

$valor=Estado::Trabajador;

//accedemos al valor con la propiedad value
$id=$valor->value;
```

Los Backed enum incorporan dos métodos estáticos adicionales:

- from(int | string): self
- tryFrom(int | string) ?self

que nos devuelven el objeto case correspondiente a valor escalar que le indicamos. En el primer caso, si el valor escalar no se encuentra se devuelve un error mientras que en el segundo se devuelve un null.

```
//asigno el Estado Desempleado a partir
//de su escalar
$v2=Estado::from(3);

echo $v2->describete();

//asigno un estado que no existe.
//en este caso asigno el valor null
$noExiste=Estado::tryFrom(100);
```

10.4.- Las enumeraciones y las clases.

Ya se ha dicho que las enumeraciones son implementadas como clases aunque no todo lo que se puede hacer con las clases es posible hacerlo con las enumeraciones:

- Las enumeraciones pueden contener métodos

- Las enumeraciones pueden implementar interfaces. De hecho un backed enum implementa la interface interna BackedEnum
- Se puede usar la variable $this y self dentro de una enumeración. ($this->name, self::cases())
- No se permite la herencia por lo que definir ambitos private o protected a un método. aunque posible, no sirve de nada.
- La mayoría de los métodos mágicos no pueden usarse (__constructor, __destructor, …)
- No se puede usar new para instanciar un objeto

11.- EXCEPCIONES

11.1.- Introducción

PHP tiene un modelo de excepciones similar al de otros lenguajes de programación. Las excepciones pueden ser lanzadas o capturadas.

Se utiliza un bloque **try** para indicar los puntos dentro del código susceptibles de producir excepciones y capturar/tratar esas excepciones y tenemos la sentencia **throw** que nos permite lanzar una excepción de cuando sea necesario. Si un error/excepción no es tratado se finaliza la ejecución del script.

En PHP 7.0 se ha creado una jerarquía de clases distinta a la existente hasta el momento. Se tiene la clase Throwable como "clase base" (realmente es una interfaz) de la que "heredan" la clase Error y la clase Exception. La mayoría de los errores generados por el sistema tienen como base Error quedando la clase Exception para el control de los errores generados por el usuario. Estas clases nos sirven a su vez como clases base sobre la que podemos definir nuestras propias excepciones. La idea es migrar todo a Error.

A continuación muestro la jerarquía existente:

```
Throwable
      Error
            ArithmeticError
                  DivisionByZeroError
            AssertionError
            CompileError
                  ParseError
            TypeError
                  ArgumentCountError
      Exception
            ...
```

La interfaz Throwable define una serie de métodos de los que destacaría getCode() (devuelve el código de error) y getMessage() (devuelve el mensaje)

11.2.- Lanzar una excepción de usuario.

El usuario puede lanzar sus propios errores mediante la sentencia **throw** usando para ello la clase Exception, ErrorException o cualquiera que cree el usuario (extendiendo alguna clase base).

```
throw new ErrorException("esto es un error");
```

11.3.- Capturar una excepción.

Para capturar una excepción se utiliza la sentencia **try**. Esta sentencia tiene la siguiente sintaxis:

```
try
    {
//sentencias
    }
catch ( ... )
    {
//sentencias para tratar la excepcion
    }
finally
    {
//sentencias a ejecutar siempre al final
    }
```

En el bloque de try se indican las sentencias susceptibles de lanzar una excepción.

Si se produce una excepción, se ejecutarán las sentencias del bloque catch correspondiente a ese error. Como objeto del catch se podrá utilizar Throwable, Exception o Error si no sabemos en concreto el error (o no queremos concretar) o una clase concreta cuando lo sabemos. Puede haber más de un bloque catch o incluso ninguno (debe indicarse finally).

Por ejemplo, para comprobar el funcionamiento para un error podríamos tener:

```
try {

    $num=(1>>-1);

}
catch (Throwable $e)
{
    echo "se ha producido el error ".$e->getCode().", ".
            $e->getMessage()." de clase ".get_class($e)."<br>".PHP_EOL;
}
finally
{
    echo "esto se escribe siempre<br>".PHP_EOL;
}
```

Daría como salida

```
se ha producido el error 0, Bit shift by negative number de clase Arith-
meticError
esto se escribe siempre
```

En su lugar en el bloque catch podría utilizarse la clase Error o incluso ArithmeticError.

Como se ha dicho antes, se pueden tener varios bloques catch. En tiempo de ejecución, para la gestión de ese error se usará el primer bloque con clase correcta. Para el caso anterior podría haber creado el siguiente bloque de control:

```
try {

   $num=(1>>-1);

}
catch (ArithmeticError $e)
{
   //controlar un error aritmetico
   echo "se ha producido el error ".$e->getCode().", ".
            $e->getMessage()." de clase ".get_class($e)."<br>".PHP_EOL;
}
catch (Error $e)
{
   //controlar un error generico
   echo "se ha producido el error ".$e->getCode().", ".
            $e->getMessage()." de clase ".get_class($e)."<br>".PHP_EOL;
}
catch (Exception $e)
{
   //controlar una excepcion
   echo "se ha producido el error ".$e->getCode().", ".
            $e->getMessage()." de clase ".get_class($e)."<br>".PHP_EOL;
}
catch (Throwable $e)
{
   // controlar un error/ excepcion cualquier
   // no entraría nunca ya que entraria por
   // los bloque anteriores.
   echo "se ha producido el error ".$e->getCode().", ".
            $e->getMessage()." de clase ".get_class($e)."<br>".PHP_EOL;
}
finally
{
   echo "esto se escribe siempre<br>".PHP_EOL;
}
```

También es posible indicar dentro de un bloque catch mas de un error, de forma que ese bloque sirva para tratar cualquier excepción de esos tipos

```
try {
// algo de código
} catch (FirstException | SecondException $e) {
// manejar la primera y segunda excepción
}
```

12.-FICHEROS

12.1.- Trabajo con ficheros

PHP tiene un conjunto de librerias relacionadas con la gestión de ficheros tanto locales como remotos (acceso ftp, web, etc). Estas librerias incluyen funciones de gestión de directorios y carpetas y para archivos.

En el caso de archivos un proceso típico es el leer/escribir datos de o a un archivo.

```php
/**
 * funcion para escribir en un fichero
 * @param string $nombre nombre del fichero
 * @param mixed $datos valores
 * @return bool devuelve si ejecuta correctamente o hay algún error
 */
function escribirAichero(string $nombre, array $datos):bool
{
    //ruta en la que se guardará el fichero
    $ruta=$_SERVER["DOCUMENT_ROOT"]."/temp";

    //si no existe la ruta se crea
    if(!file_exists($ruta))
        mkdir($ruta);


    $ruta.=$nombre;
    //se abre el fichero para escritura
    //si existe se borra el contenido
    $fic=fopen($ruta,"w");
    if(!$fic)
        return false;

    //se recorre el array con los datos
    foreach($datos as $fila)
    {
        $linea="";
        foreach($fila as $columna)
        {

            $linea.=$columna.",";
        }
        $linea.="\r\n";

        //se escribe en el fichero una linea
        fputs($fic,$linea);

    }

    //se cierra el fichero
    fclose($fic);
```

```
    return true;
}

/**
 * funcion para leer de un fichero
 * @paramstring $nombre nombre del fichero
 * @parammixed $datos valores
 * @returnbool devuelve si ejecuta correctamente o hay algún error
 */
function leerDeFichero(string $nombre,array &$datos):bool
{
    //ruta en la que se guardará el fichero
    $ruta=$_SERVER["DOCUMENT_ROOT"]."/temp";

    //si no existe la ruta se crea
    if(!file_exists($ruta))
        mkdir($ruta);


    $ruta.=$nombre;
    //se abre el fichero para lectura
    //debe existir
    $fic=fopen($ruta,"r");
    if(!$fic)
        return false;

    //borro el contenido del array
    foreach($datos as $pos=>$valor)
    {
        unset($datos[$pos]);
    }

    //leo el fichero linea a linea
    while($linea=fgets($fic))
    {
        $linea=str_replace("\r","",$linea);
        $linea=str_replace("\n","",$linea);

        if($linea!="")
        {
            $linea=explode(",",$linea);

            $datos[]=$linea;
        }
    }

    //se cierra el fichero
    fclose($fic);

    return true;
}
```

Como se ve en el ejemplo las funciones mas usuales son fopen (abrir un fichero, obtiene un recurso), fclose (cerrar el recurso correspondiente a un fichero), fgets/fread/fgetc/fscanf/… para leer de un fichero y fputs/fwrite para escribir en un fichero.

Cuando se abre un fichero se puede indicar una ruta local (d:\\datos\ o /usr/datos) o una url (ftp://ftp.datos.com, http://www.datos.com) y un modo de apertura: r (solo lectura, sin truncar, cursor al principio), w (solo escritura, truncando, cursor al principio), r+ (lectura/escritura sin truncar, cursor al principio), w+ (lectura/escritura, truncando, cursor al principio), a (solo escritura, sin truncar, puntero al final), a+ (lectura/escritura, sin truncar, puntero al final).

Existen además funciones que implementan acciones muy usuales:

- *bool* ***move_uploaded_file****(string $filename, string $destination)*. Mueve un archivo subido mediante http a otra posición.
- *array* ***file****(string $filename)*. Lee un fichero y lo devuelve como array.
- *int* ***file_put_contents****(string* `$filename`*, mixed* `$data`*)*.`Escribe el contenido del array en el fichero indicado. Equivalente a fopen-fwrite-fclose.`
- *string* ***file_get_contents****(string $filename).*Lee un fichero devolviéndo su contenido como una cadena.

12.2.- Subida de ficheros

La subida de archivos a un servidor se realiza fundamentalmente usando el métod POST.
En primer lugar es necesario indicar el código html para ello

```
<!-- El tipo de codificación de datos, enctype, se DEBE especificar como
a continuación-->
<form enctype="multipart/form-data"action="__URL__"method="POST">
  <!--MAX_FILE_SIZE debe preceder al campo de entrada de archivo-->
  <input type="hidden"name="MAX_FILE_SIZE"value="30000"/>

  <!-- El nombre del elemento de entrada determina el nombre en el array
       $_FILES -->
  Enviar este archivo: <inputname="archivo"type="file"/>
  <input type="submit"value="Enviar fichero"/>
</form>
```

Como se ve, se puede limitar el tamaño máximo del fichero a subir. “archivo” será la posición asociativa dentro de la variable $_FILES en donde encontraremos los datos del fichero subido.

Dentro del fichero php.ini se pueden configurar una serie de directivas para controlar la subida de archivos: file_uploads, upload_max_filesize, upload_tmp_dir, post_max_size y max_input_time.

Una vez subido el archivo se puede acceder a el a través de $_FILES. Para el caso anterior nos encontraríamos con:

- *$_FILES['archivo']['name']:*El nombre original del archivo en la máquina cliente.

- *$_FILES['archivo']['type']:* El tipo mime del archivo, si el navegador proporciona esta información. Un ejemplo podría ser *"image/gif"*. Este tipo mime, sin embargo no se verifica en el lado de PHP y por lo tanto no se garantiza su valor.
- *$_FILES['archivo']['size']:* El tamaño, en bytes, del archivo subido.
- *$_FILES['archivo]['tmp_name']:* El nombre temporal del archivo en el cual se almacena el archivo cargado en el servidor.
- *$_FILES['archivo]['error']:* El código de error asociado a esta carga de archivo.

El script PHP que recibe el archivo cargado, debe implementar cualquier lógica que sea necesaria para determinar qué se debe hacer con el archivo subido. Se puede, por ejemplo, utilizar la variable *$_FILES['archivo']['size']* para descartar cualquier archivo que sea demasiado pequeño o demasiado grande. Se podría utilizar la variable *$_FILES['userfile']['type']*para descartar cualquier archivo que no corresponda con un cierto criterio de tipo, pero usando esto sólo como la primera de una serie de verificaciones, debido a que este valor está completamente bajo el control del cliente y no se comprueba en el lado de PHP. A partir de PHP 4.2.0, se puede usar *$_FILES['userfile']['error']*y el planear la lógica de acuerdo con los códigos de error. Cualquiera que sea la lógica, se debe borrar el archivo del directorio temporal o moverlo a otra parte.

Si ningún archivo es seleccionado para realizar la carga en el formulario, PHP devolverá *$_FILES['userfile']['size']*como 0, y *$_FILES['userfile']['tmp_name']*como ninguno.

El archivo será borrado del directorio temporal al final de la solicitud si no se ha movido o renombrado.

```
//En versiones de PHP anteriores a 4.1.0, $HTTP_POST_FILES debe utilizar-
se en lugar
//de $_FILES.

$uploaddir='/var/www/uploads/';
$uploadfile=$uploaddir.basename($_FILES['archivo']['name']);

echo '<pre>';
if(move_uploaded_file($_FILES['archivo']['tmp_name'],$uploadfile))
   {
     echo "El archivo es válido y fue cargado exitosamente.\n";
   }
  else
   {
    echo "¡Posible ataque de carga de archivos!\n";
    }

echo 'Aquí hay más información de depurado:';
print_r($_FILES);

print "</pre>";
```

12.3.- Descarga de ficheros

Cuando se abre una página web se visualiza su contenido. Sin embargo, muchas veces nos interesa descargarla en vez de visualizarla. Por ejemplo, descargar una imagen, un documento pdf, etc. Es muy interesante cuando se genera el contenido de forma dinámica (al vuelo), es decir, el contenido depende de los parámetros de llamada.

La descarga se realiza indicando la cabecera Content-Disposition (revisar tema1)

```php
$nombreSalida="datos.txt";

header('Content-Type:'.'text/plain');
header('Content-Disposition:attachment;filename="'.$nombreSalida.'"');

//todo lo que se envíe de salida se descargará

//una cadena
echo "esto va en el fichero que se descarga";

//el contenido de un fichero
echo file_get_contents("fichero.php");
```