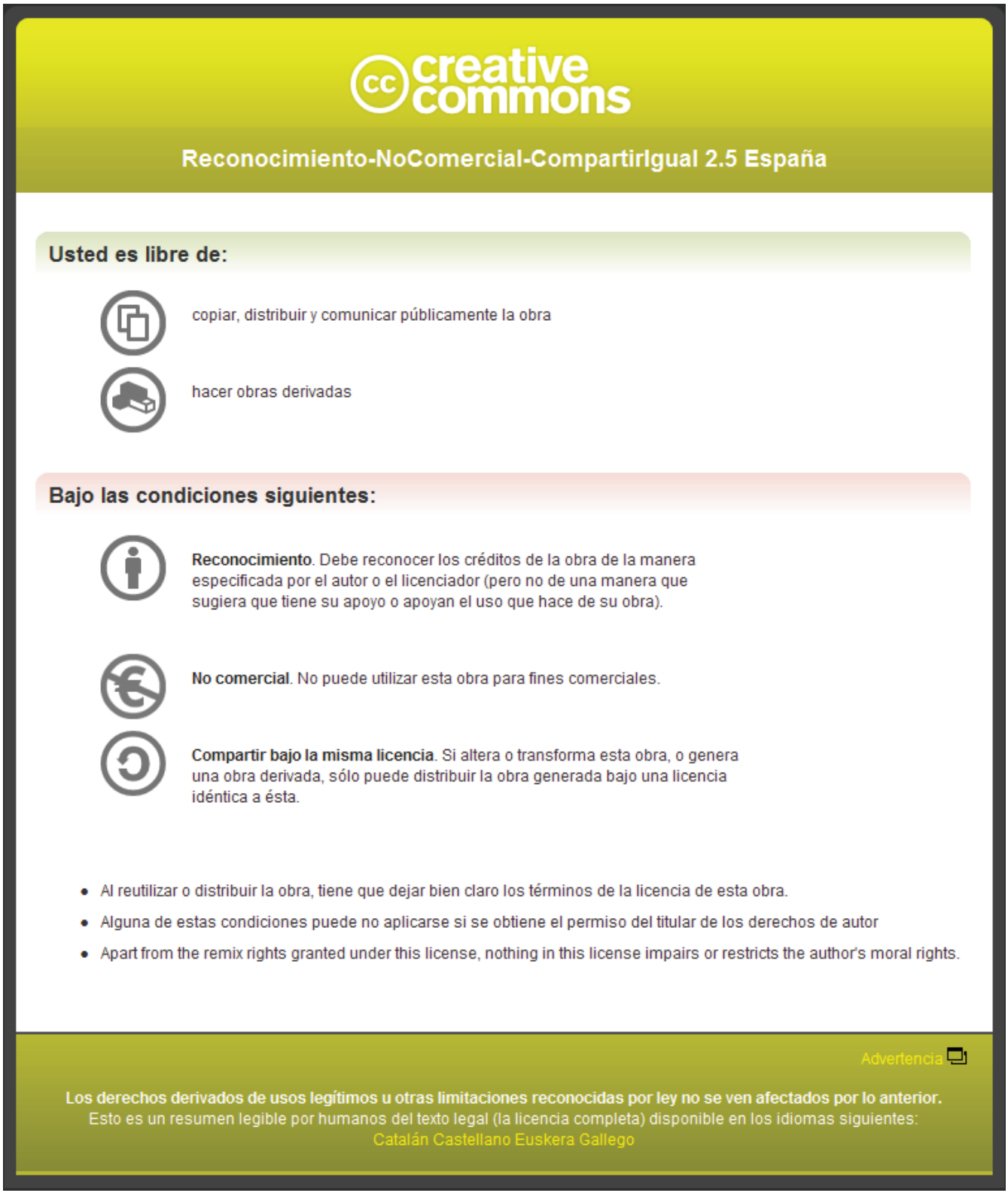

Para ver una copia completa de la licencia, acudir a la dirección
http://creativecommons.org/licenses/by-nc-sa/2.5/es/legalcode.es

índice

(2) creación de Aplicaciones Web mediante scripts de servidor. PHP

(2.1) ¿qué es PHP?

(2.1.1) lenguajes de script de servidor

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.

(2.1.2) PHP

Se trata indudablemente del 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 puede 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.

(2.1.3) ventajas de PHP

(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.

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.

(2.2) 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 permite 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 famoso 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**.

(2.3) bases de PHP

El código PHP se escribe en un documento de texto al que se debe indicar la extensión **.php**. En realidad se trata de una página web HTML (o XHTML) normal a la que se le añaden etiquetas especiales.

(2.3.1) etiqueta <? php?>

Cuando en un documento web queremos añadir código **php** se indica por esta etiqueta:

```
<?php
                    ...código PHP
¿>
```

El código PHP se coloca en la zona de la página web donde más nos interese hacerlo. Así un primer documento PHP podría ser (suponiendo que incrustamos PHP en un documento de tipo XHTML 1.0 estricto):

```
<?php
    echo '<?xml version="1.0" encoding="UTF-8"?>'
?>

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="es" lang="es">
  <head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
    <title>Título</title>
  </head>
  <body>
    <?php
          echo  “Hola”;
    ?>
  </body>
</html>
```

El primer código PHP es necesario porque actualmente se consideran las páginas web XHTML como un documento XML más y por ello deben de llevar la cabecera <?xml versión=”1.0” encoding=”utf-8”?> obligatoria en todo documento XML. EL problema es que ese código en el servidor web daría problemas porque podría entender que es

código PHP (al empezar igual que la etiqueta <?php) e intentaría traducirlo. Por ello es mejor escribirlo con la función echo para explicar que es código del documento final HTML y no PHP.

(2.3.2) HTML usa PHP y PHP usa HTML

Como se ha visto anteriormente el código PHP se incrusta dentro del código HTML. Lo interesante es que se puede incrustar código HTML en el PHP y tendría sentido ya que las funciones de escritura (**echo** y **print**) en realidad escriben hacia el resultado final, que en realidad es una página HTML por lo que se puede hacer algo como:

```
<?php
    echo "Mi nombre es <strong>Jorge</strong>";
?>
```

(2.3.3) comentarios

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

- **Comentarios de varias líneas**. Al igual que en lenguaje C, comienzan por **/*** y terminan por ***/.** Ejemplo:

```
<?php
    /*
            Soy un comentario de
            varias líneas
    */
    $x=10;
?>
```

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

```
<?php
    $x=10; //esto es un comentario
?>
```

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

```
<?php
    $x=10; #esto es un comentario
?>
```

(2.3.4) bases de escritura

Las normas básicas para escribir lenguaje PHP se basan en los lenguajes padres de este, es decir C y Perl. Son:

- 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 no es estricto. Es decir PHP entiende que ***WHILE***, ***while*** e incluso ***wHiLe*** es lo mismo al ser una palabra reservada. Sin embargo ***$var*** y ***$VAR*** no son iguales al ser el nombre de una variable.

(2.3.5) escribir en la salida

Aunque hay muchas funciones de escritura (para escribir en lo que será la página final) las fundamentales son **echo** y **print**.

echo es la más utilizada y en realidad es un comando del lenguaje. Tras echo se pasa uno o más textos (más adelante diremos ***expresiones de cadena***) que cuando son literales se escriben entre comillas. Si se usa más de un texto, se separan con comas:

```
<?php
 echo "Primer texto ", "segundo texto"
?>
```

El texto saldrá seguido en la página. Hay una forma abreviada de usar echo que es:

```
<?= “Primer texto “, “segundo texto”
?>
```

Se trata de no usar la etiqueta **<?php** para el código PHP. Si usamos **<?=** estaremos indicando que dentro de esa etiqueta hay texto a sacar con echo. No se puede colocar código de otro tipo dentro de **<?=** (porque entendería que es texto a mostrar con echo).

print funciona casi igual, pero tiene dos importantes diferencias:

- Devuelve un valor verdadero o falso dependiendo de si se pudo escribir el texto o no (en código complejo es muy útil)
- No admite varios textos, sólo uno. Aunque se le puede encadenar con el operador punto (**.**):

```
print "Primer texto ", "segundo texto"; //error
print "Primer texto "."segundo texto"; //arreglado
```

(2.4) variables

(2.4.1) introducción a 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 guión 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***.

(2.4.2) declarar

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 tremendamente la tarea de depurar nuestros programas al no ser nada rígido el lenguaje y permitir casi todo.

(2.4.3) predefinidas

El servidor web que aloje las páginas PHP pone a disposición del programador variables de sistema ya definidas para su uso en el programa. La mayoría son simplemente informativas. Todas suelen llevar el símbolo de subrayado en el segundo carácter además de escribirse en mayúsculas. La mayoría son arrays (se explican más adelante). Ejemplo:

```
echo $_SERVER[“SERVER_PORT”];
```

Escribirá el número de puerto por el que se comunica el servidor web.

(2.4.4) asignación de valores

Esta operación consiste en dar valor a una variable y se realiza con el símbolo **=**. Ejemplo:

```
$x=12;
```

Como se comentó anteriormente en PHP no es necesario declarar una variable antes de su uso. En el caso de asignar números se escriben tal cual (como en el ejemplo), los decimales se separan con el punto decimal. Los textos se encierran entre comillas (simples o dobles, aunque se aconseja usar las dobles salgo cuando nos venga mejor las simples). Ejemplo:

```
$nombre=”Anselmo”;
```

Se pueden asignar resultados de aplicar fórmulas mediante operadores, como:

```
$beneficios=($gastos-$ingresos) * 0.6;
```

Otra cosa sorprendente es que una variable puede cambiar el tipo de datos que almacena, por ejemplo es válido:

```
$x=18; //asignamos un número
$x=”Hola”; /asignamos un texto
```

(2.4.5) variables sin asignar valores

Un problema surge cuando queremos utilizar variables a las que no se les ha asignado ningún valor. Como:

```
<?php
   echo $x; //es la primera vez que se usa $x. Error
   $y=$x+2; //error, aunque y valdrá 2
   echo $y;
?>
```

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); //avisa de 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 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 un variable e indica con un resultado de verdadero, que la variable tenía asignado un valor; en cualquier otro caso devuelve falso.

(2.4.6) tipos de datos

enteros

A las variables se les puede asignar valores enteros. Los números enteros se usan tal cual. Pueden ser positivos o negativos:

```
$n1=17;
$n2=-175;
```

Se puede usar sistema octal si se coloca un cero antes de la cifra entra:

```
$octal=071;
echo $octal; //escribe 56
```

Además pueden estar en sistema hexadecimal si a la cifra se la antecede de un cero y una equis:

```
$hexa=0xA2BC;
echo $hexa; //escribe 41660
```

coma flotante

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, ocupan poco en memoria, pero desgraciadamente no son exactos. Eso provoca que pueda haber algunos decimales que se pierdan.

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:

```
$n1=234.12;
$n2=12.3e-4; //eso es 12,3·10-4, es decir 0,00123
```

Los números decimales en coma flotante de PHP son equivalentes al formato **double** del lenguaje C.

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. Ejemplo:

```
$nombre=”Jorge Sánchez”;
```

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

```
$frase = ‘Antonio dijo “Hola” al llegar’;
```

Como queremos almacenar en *$frase* el texto con *Hola* entre comillas, entonces englobamos todo el texto con comillas simples.

Otra opción es usar caracteres especiales, concretamente:

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

Y así podremos:

```
$frase = “Antonio dijo \”Hola\” al llegar”;
```

Las secuencias de escape sólo funcionan si están encerradas entre comillas dobles. Es decir, no funciona:

```
$frase = ‘Antonio dijo \”Hola\” al llegar’;
```

Las barras saldrán por pantalla también.

Un hecho de PHP muy interesante es que en un texto se puede incluir el valor de una variable. Por ejemplo:

```
$días=15;
```

```
$texto=”Faltan $días días para el verano”;
echo $texto;//escribe: Faltan 15 días para el verano
```

Por lo que el símbolo **$** sólo se puede asignar a un texto si se usa su secuencia de escape **\$**.

Los textos se pueden concatenar con ayuda del operador punto (**.**). Ejemplo:

```
$nombre="Jorge";
$apellidos="Sánchez Asenjo";
echo "Nombre completo: ".$nombre." ".$apellidos;
//escribe Nombre completo: Jorge Sánchez Asenjo
```

cadenas heredoc

Permiten asignar valores de texto sin usar comillas. Ejemplo:

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

En el ejemplo se colorea de color rojo 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). La línea que tiene el marcador de texto en la columna primera del código (es obligatorio cerrar el marcador en la posición del principio de la línea) marca el final del texto. Es decir, escribe:

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

cadenas nowdoc

Funciona igual que el anterior, sólo que entiende que el texto está encerrado entre comillas, por lo que no se interpretan los nombres de variable ni los códigos de escape. 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;
```

Resultado:

```
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.

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. En concreto hay una relación todos los tipos de datos:

- Enteros: cero=False, resto=True
- Coma flotante: 0.0=False, resto=True
- Cadenas: False si están vacías
- Arrays: False si no posee ningún elemento
- Recurso: False si el recurso no es válido.

conversiones

En PHP como las variables pueden cambiar de tipo cuando se nos antoje, resulta que tiene que intentar que todas las expresiones tengan sentido y así, este código:

```
$v1=18;
$v2="3 de Diciembre";
echo $v1+$v2;
```

Escribe 21 (suma el 18 y el tres). Pero sin embargo:

```
$v1=18;
$v2="3 de Diciembre";
echo $v1.$v2;
```

Escribe *183 de Diciembre*.

No obstante se pueden convertir de forma forzosa los valores al tipo deseado; de esta forma elegiremos nosotros cómo realizar las conversiones. Se trata del habitual *operador de casting* del lenguaje C. Ejemplo:

```
$x=2.5;
$y=4;
$z=(int)$x * $y;
```

$z vale 8 al convertir *$x* en un entero. Posibilidades:

- **(int)** o **(integer)**. Convierte a entero
- **(real), (double)** o **(flat)**. Convierte a coma flotante
- **(string)**. Convierte a forma de texto
- **(array)**. Convierte a forma de array.
- **(object)**. Convierte a un objeto-

(2.4.7) referencias &

Es una de las claves de la programación en cualquier lenguaje. Se trata de variables que sirven para modificar otras variables existentes. Es decir, son variables que en lugar de almacenar valores, almacenan la dirección de otra variable. No es una copia de la otra variable, más bien es un sinónimo de la otra variable. Su uso principal aparece cuando se utilizan funciones. Ejemplo de uso:

```
$nombre="Antonio";
$ref=&$nombre; //ref es una referencia a la variable $nombre
echo $ref,"<br />"; //escribe Antonio
$ref="Marisa";
echo $nombre,"<br />" //escribe Marisa, a través de la referencia se ha
                      // cambiado el nombre
```

(2.4.8) 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. Ejemplo:

```
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;
```

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. Ejemplo de uso:

```
const PI=3.141592;
```

Aunque no es obligatorio, es conveniente usar mayúsculas para las constantes para diferenciarlas de las variables (aunque en PHP el signo dólar $ ya hace esa diferenciación) ya que se considera una norma de buen estilo de escritura en cualquier lenguaje de programación.

(2.4.9) 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

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

Ejemplo de uso:

```
$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
```

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
&&	"Y" lógico
AND	"Y" lógico
||	"O" lógico
OR	"O" lógico
XOR	"OR" exclusivo

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** sino 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;
$granizo=FALSE;
$malTiempo= $nieva || $llueve || $granizo; //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.

operadores de asignación

Ya se ha comentado el operador de asignación que sirve para dar valor a una variable. Ejemplo:

```
$x=$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:

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

operadores de BIT

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

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

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

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

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

Pero hay 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
```

concatenación

El punto (**.**) es un operador que permite unir textos. Su uso es muy sencillo. Ejemplo:

```
$x="Hola ";
$y="a todo el mundo";
$z=$x.$y; //$z vale "Hola a todo el mundo"
```

Hay operador de asignación y concatenación, se trata del operador **.=**. Ejemplo de uso:

```
$x="Hola ";
$x.="a todo el mundo";
echo $x; //escribe "Hola a todo el mundo
```

(2.5) 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 un resultado verdadero o falso.

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

(2.5.1) sentencia if

sentencia condicional simple

Se trata de una sentencia que, tras evaluar una expresión lógica, ejecuta una serie de instrucciones en caso de que la expresión lógica sea verdadera. Si la expresión tiene un resultado falso, no se ejecutará ninguna expresión. Su sintaxis es:

```
if(expresión lógica) {
    instrucciones
    ...
}
```

Las llaves se requieren sólo si va a haber varias instrucciones. En otro caso se puede crear el **if** sin llaves:

```
if(expresión lógica) sentencia;
```

Ejemplo:

```
if($nota>=5){
    echo "Aprobado";
    $aprobados++;
}
```

Sólo se escribe *Aprobado* si la variable *$nota* es mayor o igual a cinco.

Hay otra posibilidad de sintaxis:

```
if(expresión lógica) :
    instrucciones
    ...
endif;
```

En lugar de las llaves, cierra el bloque **if** con la palabra **endif**. Los dos puntos tras el paréntesis son obligatorios. Se trata de una forma simplificada (que, ciertamente, hay que usar con cuidado).

sentencia condicional compuesta

Es igual que la anterior, sólo que se añade un apartado **else** que contiene instrucciones que se ejecutarán si la expresión evaluada por el **if** es falsa. Sintaxis:

```
if(expresión lógica){
    instrucciones a ejecutar si la exporesión es verdadera
    ....
}
else {
    instrucciones a ejecutar si la exporesión es falsa
    ...
}
```

Como en el caso anterior, las llaves son necesarias sólo si se ejecuta más de una sentencia en el bloque. Ejemplo de sentencia **if-else**:

```
if($nota>=5){
    echo "Aprobado";
    $aprobados++;
}
else {
```

```
    echo "Suspenso";
    suspensos++;
}
```

La forma simplificada de esta sentencia sería:

```
if($nota>=5):
    echo "Aprobado";
    $aprobados++;
else :
    echo "Suspenso";
    $suspensos++;
endif;
```

No se usan las llaves, en su lugar se usan los dos puntos y el cierre con **endif**.

sentencia condicional múltiples

Permite unir varios **if** en un solo bloque. Ejemplo:

```
if($nota<5){
   echo "suspenso";
   $suspensos++;
}
elseif($nota<7){
   echo "aprobado";
   $aprobados++;
}
elseif($nota<9){
   echo "notable";
   $aprobados++;
}
else{
   echo "sobresaliente";
   $aprobados++;
}
```

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.

La versión simplificada sería:

```
if($nota<5):
   echo "suspenso";
   $suspensos++;
elseif($nota<7):
   echo "aprobado";
   $aprobados++;
```

```
elseif($nota<9):
   echo "notable";
   $aprobados++;
else:
   echo "sobresaliente";
   $aprobados++;
}
```

(2.5.2) sentencia switch

```
switch (expresión) {
    case valor1:
            instrucciones del valor 1
            [break]
    [case valor2:
            instrucciones del valor 1
            [break]
    [
            .
            .
            .]
    [default:
            instrucciones que se ejecutan si la expresión no toma
            ninguno de los valores anteriores
            ..]
}
```

Los corchetes indican que su contenido es opcional. Es decir, los **break** son opcionales y la cláusula default también.

Esta instrucción se usa cuando tenemos instrucciones 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 instrucciones de ese **case** y de los siguientes.

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.**

El bloque **default** sirve para ejecutar instrucciones para los casos en los que la expresión no cumpla ninguno de los valores anteriores. Ejemplo:

```
srand(time());
$diaSemana=rand(1,7);
switch ($diaSemana) {
    case 1:
            $dia="Lunes";
            break;
```

```
    case 2:
            $dia="Martes";
            break;
    case 3:
            $dia="Miércoles";
            break;
    case 4:
            $dia="Jueves";
            break;
    case 5:
            $dia="Viernes";
            break;
    case 6:
            $dia="Sábado";
            break;
    case 7:
            $dia="Domingo";
            break;
    default:
            $dia="?";
}
```

No usar break para salir del switch puede servir para cosas como:

```
srand(time());
$diaSemana=rand(1,7);
switch ($diaSemana) {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
            echo "Laborable";
            break;
    case 6:
    case 7:
           echo "Festivo";
            break;
    default:
            echo "?";
}
```

Hay una versión simplificada de switch que permite usar los dos puntos al final de la línea del switch y sustituir la llave final del switch por el texto **endswitch**.

(2.5.3) bucles

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

bucle while

Sintaxis:

```
while (expresión lógica) {
    sentencias que se ejecutan si la condición es true
}
```

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

Ejemplo (escribir números del 1 al 100):

```
$i=1;
while ($i<=100){
    echo $i."<br />";
    $i++;
}
```

while tiene también una versión simplificada. El ejemplo anterior quedaría:

```
$i=1;
while ($i<=100):
     echo $i."<br />";
     $i++;
endwhile;
```

Este tipo de bucles son los fundamentales. Todas las demás instrucciones de bucle se basan en el bucle **while**. Por ello se comentan con él los dos principales tipos de bucle: de centinela y de contador

bucles de 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.

Ejemplo:

```
$i=10; /*Valor inicial del contador, empieza valiendo 10
          (por supuesto i debería estar declarada como entera, int) */

while ($i<=200){  /* condición del bucle, mientras i sea menor de
                     200, el bucle se repetirá, cuando i rebase este
                          valor, el bucle termina */

   echo $i."<br />";      /*acciones que ocurren en cada vuelta del bucle
                          en este caso simplemente escribe el valor
                          del contador */

   $i+=10; /* Variación del contador, en este caso cuenta de 10 en 10*/
}
/* Al final el bucle escribe:
10
20
30
...
y así hasta 200
*/
```

bucles de 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.

Ejemplo:

```
$salir=FALSE; /* En este caso el centinela es una variable
                 booleana que inicialmente vale falso */

while($salir==FALSE){ /* Condición de repetición: que salir siga siendo
                         falso. Ese es el centinela.
```

```
                        También se podía haber escrito simplemente:
                        while(!salir)
                        */
    $n=rand(1,500); // Lo que se repite en el
    echo($n);  /* bucle: calcular un número
                        aletorio de 1 a 500 y escribirlo */
    $salir=($n%7==0); /* El centinela vale verdadero si el número es
                        múltiplo de 7
                        */
}
```

Comparando los bucles de 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.

Un bucle podría ser incluso mixto: de centinela y de contador. Por ejemplo imaginar un programa que escriba números de uno a 500 y se repita hasta que llegue un múltiplo de 7, pero que como mucho se repite ocho veces.

Sería:

```
$salir = FALSE; //centinela
$i=1; //contador
while ($salir == false && $i<=8) {
   $n = rand(1,500);
   echo $n."<br />";
   $i++;
   $salir = ($n % 7 == 0);
}
```

do while

La única diferencia respecto a la 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. Es decir 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

Sintaxis:

```
do {
    instrucciones
} while (expresión lógica)
```

Ejemplo (contar de uno a 1000):

```
$i=0;
do {
    $i++;
    echo $i."<br />";
} while ($i<1000);
```

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).

De hecho cualquier sentencia **do..while** se puede convertir en while. El ejemplo anterior se puede escribir usando la instrucción while, así:

```
int i=0;
$i++;
echo $i."<br />";
while (i<1000) {
    $i++;
    echo $i."<br />";
}
```

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:

```
for(inicialización;condición;incremento){
    sentencias
}
```

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

Ejemplo (contar números del 1 al 1000):

```
for($i=1;$i<=1000;$i++){
    echo $i."<br />";
}
```

La ventaja que tiene es que el código se reduce. La desventaja es que el código es menos comprensible. El bucle anterior es equivalente al siguiente bucle **while:**

```
$i=1; /*sentencia de inicialización*/
while($i<=1000) { /*condición*/
    echo $i."<br />";
    $i++; /*incremento*/
}
```

(2.6) funciones

(2.6.1) introducción

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 puede 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 programa; 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:

(1) Las funciones poseen un nombre, un identificador que cumple las reglas indicadas para los demás identificadores que conocemos (como los de las variables). A diferencia de las variables no utilizan el signo $ y además se aconseja que su nombre se escriba en minúsculas.

(2) A las funciones se les indica, aunque no a todas, unos valores (parámetros) que la función necesita para hacer su valor

(3) Las funciones pueden devolver un valor, resultado del trabajo de la misma.

(4) Las funciones contienen el código que permite realizar la tarea para la que se creó la función.

Hay funciones ya incorporadas al lenguaje PHP y que se pueden utilizar, 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.

(2.6.2) declaración y uso de funciones personales

Los programadores de PHP pueden crear sus propias funciones para trabajar. La sintaxis es la siguiente:

```
function nombreDeLaFunción(listaDeParámetros){
    código de la función
}
```

Entre el código de la función se puede encontrar la palabra **return** que sirve para devolver el resultado.

```
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:

```
$x=9.75;
echo doble(8)."<br />";  //escribe 16
echo doble($x);          //escribe 19.5
```

No siempre las funciones devuelven valores, por ejemplo:

```
function negrita($texto){
  echo "<strong>".$texto."</strong>";
}
```

La función no tiene instrucción **return**, puesto que no devuelve valores sino que lo que hace es escribir el texto en negrita. Ejemplo de uso:

```
negrita("Hola");
```

(2.6.3) alcance de las variables

Las variables definidas en una función, al finalizar la función se eliminan. Es decir su ámbito es local a la función. Ejemplo:

```
function f1(){
```

```
  $h=9;
}
echo $h;
```

La instrucción *echo* del ejemplo anterior, provoca un fallo de variable no definida, porque el compilador PHP no reconoce a la variable *$h*, la única $h del código se crea en la función y sólo se puede usar en la función; hacer referencia a $h fuera dela función no tiene sentido, el intérprete del código PHP creerá que hay dos variables $h.

Esta es la prueba de que el alcance de las variables es local a la función en la que se define. Otro ejemplo:

```
$h=5;
function f1(){
  $h=9;
}
echo $h; //escribe 5
```

(2.6.4) 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($x){
          $x=9;
       }
$x=7;
f1($x);
echo $x;
```

En el ejemplo la función *f1* simplemente cambia el valor del parámetro *$x;* al salir de esa función *$x* muere. Por lo que al escribir lo que vale $x, se mostrará el valor de 7, ya que el $x de fuera de la función es diferente al parámetro.

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 valores. 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($x,$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 los valores. 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(&$x,&$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(9,8);
```

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

(2.6.5) 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($base,$exponente=2){
  $valor=1;
  for($i=1;$i<=$exponente;$i++){
    $valor*=$base;
  }
  return $valor;
}
echo potencia(2,3); //Escribe 8
echo potencia(2);   //Escribe 4
```

La segunda vez que se invoca a la función, el parámetro *$exponente* toma el valor por defecto *2*.

(2.6.6) variables globales

Lo comentado en el punto anterior se puede limitar mediante el uso de variables globales. Esto permite que las funciones utilicen variables que se han definido fuera de la función. Ejemplo:

```
function encadenar($número,$carácterRelleno){
   global $texto;
  for($i=1;$i<=$número;$i++){
     $texto.=$carácterRelleno;
  }
}

$texto="Hola";
encadenar(12,"d");
echo $texto."<br />"; //Escribe: Holaddddddddddddd
encadenar(9,"+-");
echo $texto."<br />"; //Escribe: Holaddddddddddddd+-+-+-+-+-+-+-+-+-
```

La función encadenar utiliza la variable global *$texto* gracias al uso de la palabra **global**. Sin embargo en realidad esto es una mala práctica ya que impide modular bien un programa ya que las funciones deberían ser lo más independientes posibles. La función del ejemplo no podría ser transportada a otra página PHP, ya que las variables globales serían otras seguramente.

En resumen, es una posibilidad que ofrece PHP pero que es poco recomendable porque con ella se adquieren malas mañas al programar.

(2.6.7) variables estáticas

Otro uso de variables especiales lo supone las 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. Ejemplo:

```
function estatica(){
  static $cuenta=0;
  $cuenta++;
  echo "Esta es la llamada número $cuenta<br />";
}

for($i=1;$i<=10;$i++){
  estatica();
}
```

La salida del código anterior es:

```
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
```

Luego en cada llamada la variable ***$cuenta*** incrementa su valor. Eso sólo es posible gracias a la línea en la que se define la variable como estática (static $cuenta=0) que solo se ejecuta en la primera llamada. Si en esa línea eliminamos la palabra **static**, entonces la variable se crea en cada llamada y las diez líneas anteriores escribiría lo mismo (***Esta es la llamada número 1***)

(2.6.8) recursividad

La recursividad es una técnica de creación de programas, pensada para soluciones a problemas complejos. Se basa en que dentro del código de la función se invoca a la propia función.

Esta técnica es peligrosa ya que se pueden generar fácilmente llamadas infinitas (la función se llama a sí misma, tras la llamada se vuelve a llamar a sí misma,…). Hay que ser muy cauteloso con ella (incluso evitarla si no es necesario su uso); pero permite soluciones muy originales y abre la posibilidad de solucionar problemas muy complejos. De hecho ciertos problemas serían casi imposibles de resolver sin esta técnica.

La idea general es que en cada llamada a la función, ésta resuelva parte del problema y se invoque a sí misma de nuevo para resolver la parte que queda, y así sucesivamente. En cada llamada el problema debe ser cada vez más sencillo hasta llegar a una llamada en la que la función devuelve un único valor. Tras esa llamada los resultados se encadenan hasta llegar al código que realizó la primera llamada y pasarle la solución.

Es fundamental tener en cuenta cuándo la función debe dejar de llamarse a sí misma; es decir, es importante decidir cuándo acabar. Toda función recursiva debe de tener una condición de fin de las llamadas. De otra forma se corre el riesgo de generar infinitas llamadas. Un bucle infinito utilizando recursividad puede causar graves inestabilidades en el ordenador de trabajo.

Como ejemplo vamos a ver la versión recursiva del factorial.

```
function factorial($n){
    if($n<=1) return 1;
    else return $n*factorial($n-1);
}
```

La última instrucción (***return $n*factorial($n-1)***) es la que realmente aplica la recursividad. La idea (por otro lado más humana) es considerar que el factorial de nueve es, nueve multiplicado por el factorial de ocho; a su vez el de ocho es ocho por el factorial de siete y así sucesivamente hasta llegar al factorial de uno, que devuelve uno.

Para la instrucción **factorial(4);** usando el ejemplo anterior, la ejecución del programa generaría los siguientes pasos:

(1) Se llama a la función factorial usando como parámetro el número 4 que será copiado en la variable-parámetro ***n***

(2) Como n>1, entonces se devuelve 4 multiplicado por el resultado de la llamada **factorial(3)**

(3) La llamada anterior hace que el nuevo ***n*** (variable distinta de la anterior) valga 3, por lo que esta llamada devolverá 3 multiplicado por el resultado de la llamada **factorial(2)**

(4) La llamada anterior devuelve 2 multiplicado por el resultado de la llamada **factorial(1)**

(5) Esa llamada devuelve 1

(6) Eso hace que la llamada **factorial(2)** devuelva 2*1, es decir 2

(7) Eso hace que la llamada **factorial(3)** devuelva 3*2, es decir 6

(8) Por lo que la llamada **factorial(4)** devuelve 4*6, es decir **24** Y ese es ya el resultado final

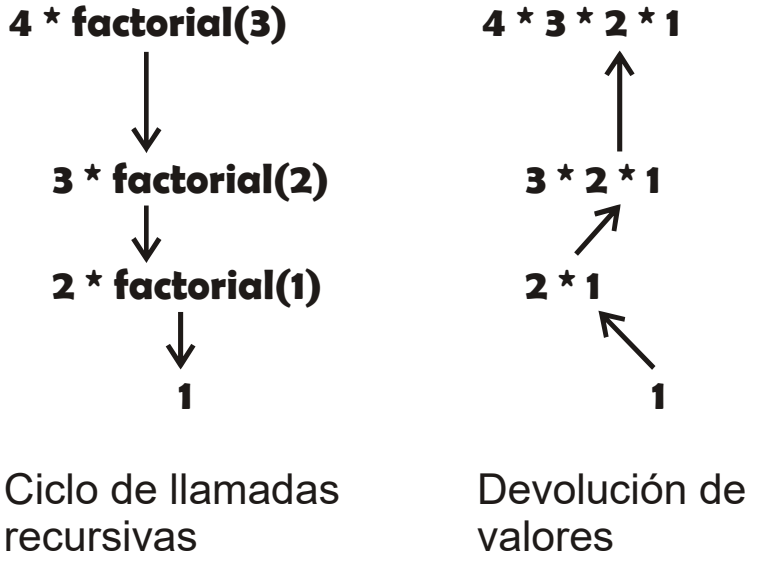

Ilustración 1, Pasos para realizar la recursividad

¿recursividad o iteración?

Hay otra versión de la función factorial resuelta mediante un bucle **for** (solución iterativa) en lugar de utilizar la recursividad. En concreto es:

```
function factorial2($n){
  $res=1;
  for($i=1;$i<=$n;$i++){
    $res*=$i;
  }
  return $res;
}
```

La cuestión es ¿cuál es mejor?

Ambas implican sentencias repetitivas hasta llegar a una determinada condición. Por lo que ambas pueden generar programas que no finalizan si la condición nunca se

cumple. En el caso de la iteración es un contador o un centinela el que permite determinar el final, la recursividad lo que hace es ir simplificando el problema hasta generar una llamada a la función que devuelva un único valor.

Para un ordenador es más costosa la recursividad ya que implica realizar muchas llamadas a funciones en cada cual se genera una copia del código de la misma, lo que sobrecarga la memoria del ordenador. Es decir, **es más rápida y menos voluminosa la solución iterativa de un problema recursivo.**

¿Por qué elegir recursividad? De hecho si podemos resolver la función mediante la solución iterativa, no deberíamos utilizar recursividad. **La recursividad se utiliza sólo si:**

- **No encontramos la solución iterativa a un problema**
- **El código es muchísimo más claro en su versión recursiva y no implica un gran coste de procesamiento al ordenador**

(2.6.9) 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.

(2.7) inclusión de ficheros

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.

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

Ejemplo de uso:

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