ÍNDICE

2 CONCEPTOS BÁSICOS.. 1
2.1 Panorámica del lenguaje...1
2.1.1 Bloques de código PHP...1
2.1.2 Sentencias en PHP. 2
2.1.3 Las instrucciones echo y print()...3
2.1.4 Funciones en PHP. 4
2.1.5 Comentarios en PHP..4
2.1.6 Errores en PHP. 5
2.2 Variables...6
2.2.1 Datos en PHP..6
2.2.2 Nombre de variables...6
2.2.3 Asignación de datos a variables..7
2.2.4 Constantes. 7
2.3 Operadores..8
2.3.1 Operadores aritméticos. 9
2.3.2 Operadores de comparación. 10
2.3.3 Operadores lógicos. 11
2.3.4 Operador de unión de cadenas..12
2.3.5 Operadores de asignación. 13
2.3.6 Precedencia de operadores. 13
2.4 Sentencias de control. 14
2.4.1 Estructuras condicionales. 15
2.4.2 Estructuras iterativas. 18
2.4.3 Anidamiento de bloques de código. 21
2.4.4 Terminación de los bucles. 23

2 CONCEPTOS BÁSICOS

2.1 Panorámica del lenguaje.

En este módulo y en el siguiente se van a describir los elementos y estructuras del lenguaje. Quien esté familiarizado con la programación estructurada encontrará familiares la mayoría de instrucciones y construcciones sintácticas.

Concretamente, la sintaxis de *PHP* es heredada del lenguaje *C*, *Java* y *Perl*. De hecho, si conocemos *C* o *Java* tendremos el 90% del camino recorrido, en lo que a sintaxis se refiere.

Las explicaciones vienen acompañadas por ejemplos para facilitar su comprensión.

Todos los ejemplos pueden ser probados tal y como se explica en la práctica del capítulo anterior. Para ello hay que crear un fichero con un editor de textos, como puede ser el *bloc de notas* (ficheros planos), dándole un nombre y guardándolo con extensión "*.php*" en la carpeta de publicación *C:\CursoPHP\htdocs\practicasPHP.*

Los resultados de los programas que hagamos se visualizarán a través del navegador, escribiendo en la barra de direcciones la *URL*, o dirección web, de nuestra página. Nuestras páginas tendrán nombres similares a:

http://localhost/practicasPHP/miprograma.php

Pasamos pues a dar una visión general del lenguaje.

2.1.1 Bloques de código PHP.

Una página web básica en *PHP* generalmente contiene dos tipos de código: el código *HTML* y el código *PHP*. El código de *PHP* que va a

procesar el intérprete debe encontrarse dentro de alguna de las siguientes etiquetas:

```
<?php ... ?>
<script language="php"> ... </script>
<% ... %>
<? ... ?>
```

Aunque las cuatro etiquetas anteriores pueden ser válidas, las dos últimas tienen el inconveniente de que hay que activarlas expresamente en la configuración de *PHP*. En este curso recomendamos el uso de la primera etiqueta ya que en la actualidad es la más estandarizada, y no requiere configuración específica.

Ejemplo 2.1.1. En este ejemplo podemos ver una página muy sencilla. Tiene tres bloques de código: al comienzo la cabecera *HTML*, después un fragmento en *PHP* y finalmente el cierre de la página otra vez en *HTML*.

```
<html>
  <head>
    <title>Hola Mundo</title>
  </head>
  <body>
    <p>

<?php
  echo "Hola Mundo";
?>

    </p>
  </body>
</html>
```

En un fichero *PHP* pueden alternarse tantos bloques *HTML* con bloques *PHP* como se quiera, siempre que estos últimos estén abiertos y cerrados por las etiquetas adecuadas.

2.1.2 Sentencias en PHP.

Una sentencia es una instrucción básica que realiza alguna acción. A veces se les llama línea de código, aunque estrictamente hablando no tienen porqué ocupar exactamente una línea en el fichero.

Es imprescindible que las sentencias estén bien diferenciadas. Para ello en *PHP* toda sentencia debe ser terminada con un punto y coma, ";". La única excepción a esto es la última sentencia antes del fin de un bloque de código *PHP*, indicado por "*?>*". Por claridad, y para evitar errores posteriores se recomienda no hacer excepciones, y terminar todas las sentencias con punto y coma, ";".

Ejemplo 2.1.2. Los siguientes bloques de código están bien formados, ya que todas las sentencias, salvo la última, terminan con un punto y coma:

```
<?php
  echo "Esto es un bloque ";
  echo "de código correcto";
?>

<?php
  echo "Esto es un bloque ";
  echo "de código correcto"
?>

<?php echo "Esto es un bloque de código correcto" ?>
```

2.1.3 Las instrucciones echo y print().

Una de las instrucciones básicas de *PHP* es la instrucción *echo.* Se limita a interpretar el texto que se pone a continuación e insertarlo en la página web. Una forma alternativa de conseguir esto es haciendo uso de la etiqueta:

```
<?= ... ?>
```

La función *print()* también es equivalente a *echo*.

Aunque aún es posible encontrar esta etiqueta, se desaconseja su uso, a favor de la instrucción *echo* ya que, su funcionamiento depende de su activación en la configuración del servidor.

Ejemplo 2.1.3. Los siguientes bloques de código *PHP* son equivalentes y por lo tanto producen el mismo resultado:

```
<?php echo "Este texto se mostrará en la página web." ?>

<?= "Este texto se mostrará en la página web." ?>

<?php print("Este texto se mostrará en la página web.") ?>
```

Ejemplo 2.1.4. En muchas ocasiones querremos insertar un salto de línea en la página de salida. En *HTML* el salto de línea se marca con la etiqueta "*
*", por lo que para conseguir este resultado lo haremos como se muestra a continuación:

```
<?php
  echo "Esta es la primera línea.";
  echo "<br>";                        //Salto de línea HTML
  echo "Esta es la segunda línea."
?>
```

2.1.4 Funciones en PHP.

En *PHP* hay dos tipos de funciones, las que están definidas por el propio lenguaje y las que puede crear a medida el programador, que se verán más adelante.

Pero en los trozos de código que vienen a continuación nos encontraremos que, para hacer acciones complejas deberemos usar las funciones predefinidas del lenguaje *PHP*.

Éstas se caracterizan por un identificador seguido por un paréntesis, dentro del cual ponemos los parámetros que tenemos que pasarle a dicha función.

A medida que las vayamos usando, iremos describiendo las funciones predefinidas. Un ejemplo de este tipo de funciones que hemos visto ya es la instrucción *print().*

2.1.5 Comentarios en PHP.

Los cometarios son ignorados por el intérprete a la hora de generar la página. El objetivo de los comentarios es aumentar la legibilidad del código para su posterior mantenimiento. Mediante los comentarios describiremos la función de determinados fragmentos de código.

En PHP hay tres formas de poner comentarios:

```
<?php
  // Este es un comentario de una sola línea
  /*
  Este es un comentario
  que ocupa varias líneas
  */
  # Esta forma de poner comentarios de una línea se usa menos
?>
```

Hay que tener en cuenta que si bien, los comentarios ayudan a comprender el código y por lo tanto facilitan el mantenimiento del código por nuestra parte o por la de otras personas, un abuso de los mismos puede hacer difícil distinguir a primera vista el propósito del código.

Lo ideal es escribir un código limpio que resulte lo más legible posible, y limitar el uso de comentarios a aquellas ocasiones en los que haya que hacer una aclaración imprescindible.

También es habitual insertar anotaciones en el código a medida que se está construyendo, y aún no ha tomado su forma final. Este tipo de comentarios ayudan a revisar y corregir el código, y es conveniente eliminarlos cuando dejan de ser necesarios. Los términos más habituales de este tipo de comentarios son:

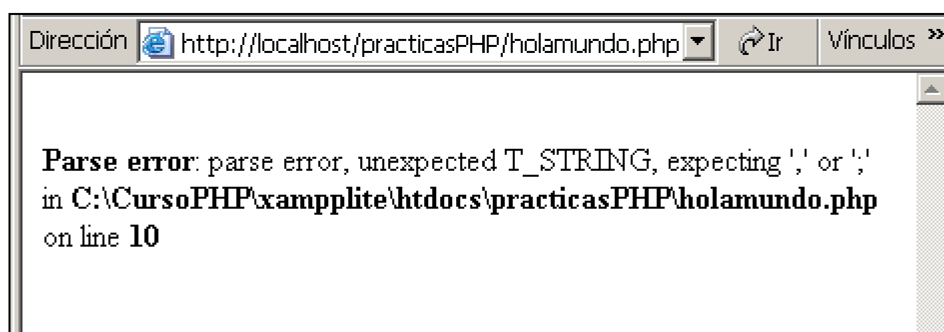
```
<?php
  //FIXME: Marca un trozo de código que habría que mejorar
  //DOME: Marca una característica sin programar
?>
```

No está de más recordar que los comentarios son parte del código *PHP*. Por tanto, debemos ubicarlos dentro de los bloques de código *PHP*. Es decir, entre la etiqueta de abertura "*<¿php*" y la etiqueta de cierre "*?>*".

2.1.6 Errores en PHP.

Sin lugar a dudas, cuando programemos, cometeremos diferentes errores. Los más habituales son: olvidar poner un punto y coma al final de una instrucción, o no cerrar adecuadamente unas comillas.

Cuando vayamos a intentar ver el resultado en el navegador aparecerá uno o más mensajes de error similares a este.



Dado que estos mensajes se visualizan en inglés, es posible que nos dificulte la comprensión del error aunque, sin ninguna duda, nos ayudará a identificar la ubicación del error. Este error en concreto nos indica que en la línea 10 del fichero de código falta una coma o, lo que es más probable, un punto y coma que separe diferentes instrucciones.

2.1.6.1 Inhibir errores.

En ocasiones no querremos que el error se vea por pantalla. Para conseguir esto pondremos el símbolo arroba "@" antes de la sentencia en la que se produce el fallo.

Ejemplo 2.1.5. El primer trozo de código muestra un mensaje de error al intentar abrir un archivo que no existe. En cambio en el segundo fragmento se hace uso del símbolo @ para evitar que dicho error se muestre por pantalla.

```
<?php
  $archivo = fopen("noexiste.txt", "r");
?>

<?php
  @$archivo = fopen("noexiste.txt", "r");
?>
```

2.2 Variables.

2.2.1 Datos en PHP.

En *PHP* definimos los datos con los que vamos a trabajar como variables. Una variable no es más que una estructura de datos cuyo valor puede cambiar con el tiempo. *PHP* trabaja con tres tipos de datos sencillos: *integer* (número entero), *double* (número con decimales) y *string* (cadena de texto).

El separador de decimales en *PHP* es el punto, como en la mayoría de los lenguajes de programación.

Ejemplo 2.2.1. Como se puede observar, los datos de tipo *string* se escriben entre comillas, que pueden ser simples o dobles. Fíjese que en los dos últimos el dato es de tipo cadena, aunque la última cadena contenga un número.

```
11                  // Esto es un dato de tipo integer
43.95               // Esto es un dato de tipo double
"Código postal"     // Esto es un dato de tipo string
"08080"             // Esto también es un dato de tipo string
```

Ejemplo 2.2.2. Si queremos escribir un dato que contenga comillas dobles lo escribiremos encerrado entre comillas simples. Y si queremos escribir un dato que contenga comillas simples lo encerraremos en comillas dobles.

```
'Leopoldo Alas "Clarin" fué un gran escritor...'
"Leopoldo Alas 'Clarin' fué un gran escritor..."
```

2.2.2 Nombre de variables.

Todas las variables en *PHP* se distinguen por comenzar con el símbolo del dólar, "$", seguido del nombre de la variable. El nombre de una variable ha de empezar necesariamente por una letra o el símbolo guión bajo "_" seguido por cualquier combinación de números, letras o guiones bajos.

Ejemplo 2.2.3. Diferentes ejemplos de nombres de variables, válidos y no válidos.

```
$apellido1;       // nombre de variable válido
$_tfno;           // nombre de variable válido
$1apellido;       // no válido, empieza por un número
$email@;          // no válido, contiene el símbolo @
```

Es importante tener en cuenta que en los nombres de variables se distingue entre mayúsculas y minúsculas.

Ejemplo 2.2.4. Mediante este ejemplo se puede ver que *PHP* distingue entre mayúsculas y minúsculas en los nombres de variables.

```
<?php
  $precio = 33;
  $Precio = 55;
  echo "La variable precio contiene el valor: ";
  echo $precio;
  echo "<br>";
  echo "La variable Precio contiene el valor: ";
  echo $Precio;
?>
```

A pesar de que tenemos gran libertad a la hora de poner nombres a las variables es recomendable seguir los siguientes consejos para que el código resulte más legible:

- Usar nombres de variable que tengan significado.
- No usar nombres de variable demasiado largos (12-16 caracteres deberían ser suficientes en la mayoría de los casos).
- No usar como nombre de variable palabras clave de *PHP*.

Ejemplo 2.2.5. Por lo motivos anteriores, los siguientes nombres de variable no son recomendables.

```
$x37                        // No tiene significado
$preciodelproductoconiva    // Demasiado largo
$echo                       // Echo es una palabra clave de PHP
```

2.2.3 Asignación de datos a variables.

Como se ha ido viendo en los ejemplos, a las variables se les asigna un dato mediante el símbolo igual "=".

Ejemplo 2.2.6. Asignación de un dato de tipo cadena a una variable.

```
$telefono = "918903220";
```

Hay que tener en cuenta que *PHP* es un lenguaje poco tipado, Las variables no tienen un tipo de datos predefinido al crearlas, sino que el tipo vendrá dado según el contenido de la variable. Este es el punto más débil del lenguaje, aunque le aporta una sencillez y flexibilidad propias.

2.2.4 Constantes.

Las constantes son una manera de asignar a un identificador un valor simple que no varíe. La declaración de una constante se realiza mediante la función *define()*.

El nombre de la constante debe seguir las mismas reglas que las variables. La única diferencia consiste en que no lleva delante el símbolo del dólar, "$", ya que no es una variable. Una práctica habitual es poner los nombres de las constantes en mayúsculas.

Ejemplo 2.2.7. Sabemos a ciencia cierta que el número PI no va a cambiar. En lugar de tener que teclearlo varias veces a lo largo del código podemos definir una constante con el identificador PI.

```
<?php
  define("PI", 3.1416);
  echo PI; //No se pone el símbolo dolar
?>
```

2.3 Operadores.

Los operadores son utilizados para realizar cálculos y manipular datos en el ámbito de una expresión. Las expresiones más comunes que conocemos, como la multiplicación, implican la participación de dos operandos (los datos) y un operador (el símbolo de la multiplicación). En *PHP* puede haber:

- Operadores unarios: la expresión manipula un único dato.
- Operadores binarios: la expresión manipula dos datos.
- Operadores ternarios: la expresión manipula tres datos.

Ejemplo 2.3.1. Los datos pueden ser introducidos directamente o estar almacenados en variables. El resultado de la operación se suele asignar a una variable, aunque puede formar parte de una expresión mayor. Esta variable a la que se asigna el resultado puede ser parte de la expresión.

```
<?php
  echo "La variable z vale ";
  $z = 1 + 5; //Los datos se introducen directamente
  echo $z;
  echo "<br>";

  echo "La variable z vale ";
  $x = 2;
  $y = 3;
  $z = $x + $y; //Los datos están almacenados en variables
  echo $z;
  echo "<br>";

  echo "La variable z vale ";
  $z = $x + $z; //La variable $z aparece en la expresión
  echo $z;
?>
```

Lo habitual es clasificar a los operadores según el tipo de cálculo o manipulación que realizan.

2.3.1 Operadores aritméticos.

En *PHP* tenemos los siguientes operadores aritméticos:

Operador	Nombre	Descripción	Ejemplo	Result.
+	Suma	Devuelve la suma de dos operandos.	2 + 2	4
-	Resta	Devuelve la diferencia entre dos operandos.	5 - 3	2
*	Producto	Devuelve la multiplicación entre dos operandos.	2 * 3	6
/	Cociente	Devuelve el cociente de la división entre dos operandos.	8 / 2	4
%	Módulo	Devuelve el resto de la división entre dos operandos.	7 % 4	3
-	Cambio de signo	Devuelve el valor del operando con el signo cambiado.	$x=6; $y=-x; $y;	-6

Ejemplo 2.3.2. Aquí se experimenta con varias operaciones aritméticas.

```
<?php
  echo "El resultado de sumar 5 y 3 es ";
  $z = 5 + 3;
  echo $z;
  echo "<br>";
  echo "El resultado de restar 3 a 5 es ";
  $z = 5 - 3;
  echo $z;
  echo "<br>";
  echo "El resultado de multiplicar 5 y 3 es ";
  $z = 5 * 3;
  echo $z;
  echo "<br>";
  echo "El resultado de dividir 5 entre 3 es ";
  $z = 5 / 3;
  echo $z;
  echo "<br>";
  echo "El resultado de calcular 5 módulo 3 es ";
  $z = 5 % 3;
  echo $z;
  echo "<br>";
  echo "El resultado de cambiar el signo a la variable $z es ";
  $z = -$z;
  echo $z;
?>
```

2.3.2 Operadores de comparación.

En el siguiente módulo se tratará de las sentencias de control. Éstas utilizan el resultado de unas expresiones especiales para decidir cual será la instrucción que ejecutarán luego. Las expresiones básicas que utilizan son el resultado de una condición. Este resultado sólo puede tomar dos valores: *verdadero* o *falso*. *PHP* tiene dos constantes predefinidas para designar los valores *verdadero* y *falso* que son, como cabía esperar, TRUE y FALSE. Se pueden usar también en minúsculas, aunque conviene recordar que la práctica habitual en programación es usar las constantes en mayúsculas.

Cuando el resultado es verdadero la expresión de comparación devolverá TRUE, si la evaluación de la expresión de comparación es falsa *PHP* devolverá como resultado FALSE.

Operador	**Nombre**	**Descripción**	**Ejemplo**	**Result.**
==	Es igual	Devuelve verdadero si los dos operandos son iguales.	2 == 2 2 == 5	TRUE FALSE
!=	No es igual	Devuelve verdadero si los dos operandos son diferentes.	2 != 2 2 != 5	FALSE TRUE
<	Es menor	Devuelve verdadero si el primer operando es menor que el segundo.	2 < 5 5 < 2	TRUE FALSE
>	Es mayor	Devuelve verdadero si el primer operando es mayor que el segundo.	2 > 5 5 > 2	FALSE TRUE
<=	Es menor o igual	Devuelve verdadero si el primer operando es menor o igual que el segundo.	2 <= 5 2 <= 2	TRUE TRUE
>=	Es mayor o igual	Devuelve verdadero si el primer operando es mayor o igual que el segundo.	5 >= 2 2 >= 2	TRUE TRUE

A nivel interno *PHP* no almacena TRUE o FALSE, sino que interpreta el valor 0 como *falso* e, históricamente, el 1como verdadero.

Realmente en *PHP* cualquier valor que sea diferente de 0 se interpretará como *verdadero*. Por ejemplo, en *PHP* el valor 5 es interpretado como *verdadero*.

Ejemplo 2.3.3. En este ejemplo aparece la construcción if... else que se verá en la próxima unidad didáctica.

```php
<?php
  echo "¿Es 5 igual a 3? ";
  if (5 == 3) {
    echo "TRUE";
  } else {
    echo "FALSE";
  }
  echo "<br>";
  echo "¿Es 5 diferente a 3? ";
  if (5 != 3) {
    echo "TRUE";
  } else {
    echo "FALSE";
  }
  echo "<br>";
  echo "¿Es 5 menor que 3? ";
  if (5 < 3) {
    echo "TRUE";
  } else {
    echo "FALSE";
  }
  echo "<br>";
  echo "¿Es 5 mayor que 3? ";
  if (5 > 3) {
    echo "TRUE";
  } else {
    echo "FALSE";
  }
?>
```

2.3.3 Operadores lógicos.

Los operadores lógicos sirven para relacionar los resultados de condiciones. Es decir, combinan los resultados de valores TRUE y FALSE. Son útiles para representar expresiones del tipo ¿Es cierto que Madrid es la capital de España y que a la vez Madrid es la ciudad más poblada de España?

Operador	Nombre	Descripción	Ejemplo	Result.
&& And	AND (y)	Devuelve verdadero si ambos operandos son verdaderos.	TRUE && FALSE	FALSE
|| Or	OR (o)	Devuelve verdadero si al menos uno de los operandos es verdadero.	TRUE && FALSE	TRUE
Xor	XOR (o exclusivo)	Devuelve verdadero si uno de los operandos es verdadero pero no ambos.	TRUE &&TRUE TRUE && FALSE	FALSE TRUE
!	NOT (negación)	Devuelve verdadero si el operando es falso y falso si es verdadero	TRUE FALSE	FALSE TRUE

Ejemplo 2.3.4. Al conjunto de combinaciones de valores que pueden tomar los operandos y el resultado de la operación se le conoce en lógica como tabla de verdad.

En el siguiente ejemplo construiremos la tabla de verdad del operador AND, al igual que podríamos hacer con el resto de los operadores.

```
<font face="Courier New">
<?php
  echo "Tabla de verdad del operador AND ";
  echo "<br>";
  echo "<br>";
  echo "opdo1 | opdo2 |opdo1 AND opdo2";
  echo "<br>";
  echo "------+-------+---------------";
  echo "<br>";
  echo "TRUE. | TRUE. | ";
  if (TRUE && TRUE) {
    echo "TRUE.";
  } else {
    echo "FALSE";
  }
  echo "<br>";
  echo "TRUE. | FALSE | ";
  if (TRUE && FALSE) {
    echo "TRUE.";
  } else {
    echo "FALSE";
  }
  echo "<br>";
  echo "FALSE | TRUE. | ";
  if (FALSE && TRUE) {
    echo "TRUE.";
  } else {
    echo "FALSE";
  }
  echo "<br>";
  echo "FALSE | FALSE | ";
  if (FALSE && FALSE) {
    echo "TRUE.";
  } else {
    echo "FALSE";
  }
?>
</font>
```

2.3.4 Operador de unión de cadenas.

El operador concatenar nos permite unir dos cadenas. Esto nos permite, por ejemplo, usar una sola instrucción *echo* para mostrar un texto, en lugar de usar una instrucción *echo* por cada fragmento.

Op.	Nombre	Descripción	Ejemplo	Result.
.	Concatenar	Une dos cadenas de texto.	"Hola" . "Mundo"	"Hola Mundo"

Ejemplo 2.3.5. Se pueden concatenar directamente cadenas de texto con datos de tipo entero. Los datos que se concatenan se pueden introducir directamente o extraer de variables.

```
<?php
  $x = 2;
  $y = 3;
  $z = $x + $y;
  echo "El resultado de sumar 2 y 3 es " . 5 . "<br>";
  echo "El resultado de sumar 2 y 3 es " . (2 + 3) . "<br>";
  echo "El resultado de sumar 2 y 3 es " . $z . "<br>";
?>
```

2.3.5 Operadores de asignación.

En la unidad didáctica sobre variables hemos visto el operador básico de asignación, que es el símbolo igual "=". Pero hay otros operadores que permiten hacer una operación y una asignación a la vez y por lo tanto producen un código más comprimido.

El uso de estos operadores es desaconsejable para el programador principiante ya que puede llevar a un código poco legible y como consecuencia a asignaciones erróneas.

Los operadores de asignación comprimidos y sus equivalencias son:

Operador	Ejemplo	Equivalencia
=	$x = 4	$x = 4
+=	$x += 4	$x = $x + 4
-=	$x -= 4	$x = $x – 4
*=	$x *= 4	$x = $x * 4
/=	$x /= 4	$x = $x / 4
%=	$x %= 4	$x = $x % 4
.=	$x .= "mundo"	$x = $x . "mundo"
++	$x++	$x = $x + 1
--	$x--	$x = $x - 1

2.3.6 Precedencia de operadores.

Un último aspecto que conviene mencionar es, ¿qué pasa cuando en una expresión compleja aparece más de un operando? En ese caso, ¿qué operaciones se evalúan antes?

Cuando hay expresiones complejas con más de un operando hay unos que se evalúan antes de otros siguiendo un orden concreto.

Este orden se representa en la siguiente tabla, teniendo en cuenta que los de arriba se evalúan antes que los de abajo y que los de la misma fila tienen igual precedencia (se evalúan de izquierda a derecha):

Operador lógico de negación	!
Operadores multiplicativos	* / %
Operadores aditivos	+ - .
Operadores de comparación	< <= > >=
Operadores de comparación (igualdad)	== !=
Operadores lógicos (salvo la negación)	&& || xor
Operadores de asignación	= (y operadores comprimidos)

En cualquier caso podemos alterar la precedencia mediante la inclusión de paréntesis alrededor de las expresiones. De hecho es muy recomendable hacerlo incluso aunque exista una regla de precedencia que produzca el mismo resultado para dar mayor claridad al código.

Ejemplo 2.3.6. La multiplicación tiene precedencia sobre la suma por lo que sin paréntesis, en la expresión 2 + 5 * 4 se evalúa primero 5 * 4 y luego se le suma dos.

```
<?php
  echo "El resultado de (2 + 5) * 4 es " . ((2 + 5) * 4);
  echo "<br>";
  echo "El resultado de 2 + 5 * 4 es " . (2 + 5 * 4);
  echo "<br>";
  // El último caso resulta más legible si ponemos
  echo "El resultado de 2 + 5 * 4 es " . (2 + (5 * 4));
?>
```

2.4 Sentencias de control.

Hasta ahora solo hemos visto instrucciones muy sencillas. Además, lo único que podíamos hacer con ellas es ejecutarlas secuencialmente. Pero, ¿tenemos herramientas para elegir una alternativa entre dos? ¿O para escribir 5 veces un texto por pantalla? Para hacer frente a estas situaciones se incluyen en cualquier lenguaje de programación las sentencias de control.

2.4.1 Estructuras condicionales.

2.4.1.1 Instrucción if.

Las instrucciones condicionales permiten ejecutar diferentes fragmentos de código en función del resultado de una expresión condicional. La instrucción básica es *if.* La sintaxis es:

```
if (condición) {
  // Código a ejecutar si la condición es cierta
}
```

Cuando el código a ejecutar si la condición es cierta consiste en una única sentencia, se pueden omitir las llaves "{" y "}". En cualquier caso, es preferible no omitirlos ya que dan mayor claridad a cuales son exactamente las sentencias que se ejecutarán, en el caso de cumplirse la condición.

Ejemplo 2.4.1. Las dos construcciones siguientes son equivalentes, aunque es preferible usar la primera por claridad:

```
<?php
  $hora = 8; // La hora en formato de 24 horas
  if ($hora == 8) {
    echo "Suena el despertador.";
  }
  echo "<br>";

  if ($hora == 8)
    echo "Suena el despertador.";
?>
```

Ejemplo 2.4.2. Veamos la diferencia entre usar llaves para encuadrar un bloque de texto, y no hacerlo:

```
<?php
  $hora = 11; // La hora en formato de 24 horas
  if ($hora == 8) {
    echo "Suena el despertador. ";
    echo "Es la hora de levantarse.";
  }
  echo "<br>";

  if ($hora == 8)
    echo "Suena el despertador. ";
  echo "Es la hora de levantarse.";
?>
```

2.4.1.2 Instrucción else.

En muchos casos no nos bastará con ejecutar un código si ese cumple la condición, sino que querremos elegir entre dos opciones alternativas y mutuamente excluyentes.

En estos casos la sentencia *if* se completa con un bloque de código que sólo se ejecuta si la condición es falsa:

```
if (condición) {
  // Código a ejecutar si la condición es cierta
} else {
  // Código a ejecutar si la condición es falsa
}
```

Ejemplo 2.4.3. En este caso, necesariamente se ejecutará el primer bloque (si es cierta la condición), o el segundo (si fuera falsa).

```
<?php
  $hora = 17; // La hora en formato de 24 horas
  if ($hora <= 12) {
    echo "Son las " . $hora . " de la mañana";
  } else {
    echo "Son las " . ($hora - 12) . " de la tarde";
  }
?>
```

2.4.1.3 Instrucción else if.

También puede suceder que queramos encadenar diferentes condiciones. Para ello se puede encadenar una sentencia *else* con un nuevo condicional. A su vez la nueva sentencia *if* puede ser completada por un *else*.

```
if (condición1) {
  // Código a ejecutar si la condición1 es cierta
} else if (condición2) {
  // Código a ejecutar si la condición1 es falsa y la 2 cierta
} else {
  // Código a ejecutar si ambas condiciones son falsas
}
```

Ejemplo 2.4.4. Se pueden encadenar varias sentencias else if como se observa en el siguiente programa:

```
<?php
  $hora = 14; // La hora en formato de 24 horas
  if ($hora == 8) {
    echo "Es la hora de desayunar.";
  } else if ($hora == 14) {
    echo "Es la hora de la comida.";
  } else if ($hora == 21) {
    echo "Es la hora de la cena.";
  } else {
    echo "Ahora no toca comer.";
  }
?>
```

2.4.1.4 Instrucción switch.

En el caso anterior se elegía entre varias opciones alternativas y excluyentes.

PHP nos ofrece una instrucción que nos facilita la tarea siempre que haya que elegir entre varias opciones posibles. Se trata de la función *switch*.

A diferencia de cómo ocurría con la instrucción *if*, si queremos que se ejecuten varias instrucciones no será necesario englobarlas entre llaves "{,}", sino que la ejecución seguirá hasta que encuentre el siguiente *break*.

```
switch (expresion) {
  case etiqueta1:
    // Código a ejecutar si expresión == etiqueta1
    break;
  case etiqueta2:
    // Código a ejecutar si expresión == etiqueta2
    break;
  case default:
    // Código a ejecutar si no se verifica ninguna etiqueta
    break;
}
```

El bloque de código detrás del *default* se ejecuta en el caso de que no se haya verificado ninguno de los casos anteriores. Terminar el bloque de instrucciones *default* con una instrucción *break* es opcional, pero es recomendable ya que mejora la legibilidad.

Ejemplo 2.4.5. Puede haber tantas sentencias *case* como sea necesario.

```
<?php
  $hora = 14; // La hora en formato de 24 horas
  switch ($hora) {
    case 9:
      echo "Es la hora de desayunar.";
      break;
    case 14:
      echo "Es la hora de comer.";
      break;
    case 21:
      echo "Es la hora de merendar.";
      break;
    default:
      echo "Ahora no toca comer.";
      break;
  }
?>
```

Ejemplo 2.4.6. Aquí se puede observar dos estrategias que nos pueden ser útiles en algunos casos. La primera es formar un grupo de alternativas que producen un resultado. La segunda es omitir intencionadamente la

instrucción *break* al final de un bloque. Probar el siguiente código con diferentes valores de hora.

```
<?php
  $hora = 19; // La hora en formato de 24 horas
  switch($hora) {
    case 24:
    case 23:
    case 22:
      echo "Ya he cenado.";
    case 21:
    case 20:
    case 19:
    case 18:
    case 17:
    case 16:
    case 15:
      echo "Ya he comido.";
    case 14:
    case 13:
    case 12:
    case 11:
    case 10:
      echo "Ya he desayunado.";
      break;
    default:
      echo "Tengo hambre";
      break;
  }
?>
```

2.4.2 Estructuras iterativas.

En muchas ocasiones al programar, hay una acción que se repite varias veces. Podríamos poner repetir varias veces el mismo bloque de código para conseguir este efecto, pero esto nos conduciría a un código redundante y muy largo. Además, ¿Qué sucede si no conocemos las veces que queremos repetir el código a priori?

2.4.2.1 Instrucción while.

Es la estructura más sencilla. Repite indefinidamente la ejecución de un bloque de código mientras la evaluación de la *condición* sea cierta.

La forma que tenemos de controlar cuantas veces se repite el bucle *while* es a través de la *condición*. Por lo tanto en el cuerpo del bucle se han de modificar las variables que intervienen en la *condición* de tal forma que en algún momento sea falsa.

```
while (condición) {
   // Código a ejecutar mientras la condición sea cierta
}
```

Ejemplo 2.4.7. El ejemplo más típico de ejecución de un bucle es contar. Es importante observar dos cosas:

1. La variable $i se inicializa fuera del bucle.

2. La variable $i se modifica en la última instrucción del cuerpo del bucle. De esta forma cada vez que se evalúa la *condición* ha cambiado.

Cuando $i llega a valer 11 la *condición* se volverá falsa y, por tanto, el bucle terminará.

```
<?php
  $i = 1;
  while ($i <= 10) {
    echo "Línea " . $i;
    echo "<br>";
    $i++;
  }
?>
```

Ejemplo 2.4.8. Un ejemplo un poco más complejo. En la expresión condición del bucle se hace uso de una constante por primera vez en los ejemplos.

```
<?php
  define(HORA_DE_CENAR, 21); //Constante
  $hora = 12; // La hora en formato de 24 horas
  while ($hora < HORA_DE_CENAR) {
    echo "Son las " . $hora . ". ";
    echo "Faltan " . (HORA_DE_CENAR - $hora) . " horas ";
    echo "para la cena.";
    echo "<br>";
    $hora++;
  }
  echo "Son las " . $hora . ". Es la hora de la cena.";
?>
```

2.4.2.2 Instrucción do ... while.

En *PHP* se dispone también de la estructura *do ... while*, que a diferencia de la anterior evalúa la condición al final en lugar de al principio del bucle.

De esta forma el bucle se ejecuta siempre al menos una vez y a partir de ese momento se repite mientras la condición sea cierta.

```
do {
   // Código a ejecutar
} while (condición);
```

Ejemplo 2.4.9. Uno de los casos en los que se usa esta estructura es cuando se está esperando recibir un dato en concreto, y hay que realizar varias veces la lectura del dato hasta que se recibe el dato esperado. Como aún no conocemos ninguna forma de recibir datos se ha optado por generarlos aleatoriamente. En este ejemplo se simula el lanzamiento de un dado de parchís.

```
<?php
  do {
    $dado = rand(1,6); //rand() devuelve un valor aleatorio
    echo "Tirando el dado... ";
    echo "ha salido un " . $dado . ".";
    echo "<br>";
  } while ($dado != 5);
  echo "¡Bien! Saco una ficha de casa.";
?>
```

Si en este ejemplo se recarga la página, se observarán los resultado de la aleatoriedad generada por la instrucción *rand()*.

2.4.2.3 Instrucción for.

La instrucción *for* es probablemente la más fácil de comprender para las personas que están aprendiendo a programar, dado que en muchos casos se conoce el número de veces que se va a ejecutar el bucle antes de entrar al mismo. La sintaxis que tiene en *PHP* es:

```
for (inicialización; condición; incremento) {
   // Código a ejecutar
}
```

El bucle seguirá ejecutándose de forma continua siempre y cuando la condición siga siendo TRUE.

Ejemplo 2.4.10. Aquí se puede ver un bucle *for* sencillo que, al igual que el bucle *while* del ejemplo 2.4.7, imprime 10 líneas numeradas. Como se puede observar es habitual utilizar la variable $i en el cuerpo del bucle. A esta variable se le llama a veces *índice*.

```
<?php
for ($i = 1; $i <= 10; $i++) {
  echo "Línea " . $i;
  echo "<br>";
}
?>
```

Ejemplo 2.4.11. La instrucción de *incremento* no tiene porqué hacer aumentar el valor, también lo puede hacer retroceder. En este ejemplo se imprime una cuenta atrás.

```
<?php
  echo "Cuenta atrás.";
  echo "<br>";
  for ($i = 10; $i >= 0; $i--) {
    echo $i . " ... ";
    echo "<br>";
  }
  echo "Ignición!";
?>
```

Ejemplo 2.4.12. También se puede hacer que la instrucción incremento aumente en cada paso el valor en más de una unidad. Por ejemplo, el

siguiente programa cuenta los números del 3 al 15 con un incremento de 3 en 3.

```
<?php
  for ($i = 3; $i <= 15; $i = $i + 3) {
    echo $i;
    echo "<br>";
  }
?>
```

Ejemplo 2.4.13. La *condición* no tiene por qué ser necesariamente sobre la variable declarada como *índice*. En este ejemplo se puede observar como la variable utilizada en la comparación es distinta de la variable incrementada, pese a que su valor depende de ella. El resultado obtenido será el mismo que con el ejercicio anterior.

```
<?php
  $j = 3;
  for ($i = 1 ; $j <= 15 ; $i++) {
    $j = $i * 3;
    echo $j;
    echo "<br>";
  }
?>
```

La instrucción *for* de *PHP*, que ha sido heredada del lenguaje *C*, es una instrucción realmente versátil y de gran potencia.

Debido a su particular sintaxis se puede emplear para usos totalmente impensable en otros lenguajes similares.

Por este motivo, es muy recomendable para el programador inexperto seguir algunos consejos básicos durante su aprendizaje.

- La condición debería realizarse sobre el índice.
- También hay que evitar alterar el contenido del índice en el cuerpo del bucle, pues puede llevar a comportamientos impredecibles si no se tiene cuidado.

2.4.3 Anidamiento de bloques de código.

Hasta ahora hemos visto como las sentencias de control actuaban de forma aislada. Sin embargo, en la mayoría de las situaciones nos encontraremos con que, para obtener el resultado esperado, hemos de hacer que trabajen juntas.

Esto no supone ningún problema, ya que un bloque de código dentro de una sentencia de control puede contener varias sentencias condicionales, y a su vez los bloques de código de éstas otras sentencias condicionales nuevas.

Ejemplo 2.4.14. Aquí se puede ver un caso típico del uso de sentencias de control anidadas.

```
<?php
  for ($i = 1; $i <= 10; $i++) {
    if (($i % 2) == 1) {
      echo "El " . $i . " es un número impar.";
      echo "<br>";
    } else {
      echo "El " . $i . " es un número par.";
      echo "<br>";
    }
  }
?>
```

El principal problema con el que nos podemos encontrar es tener un código poco legible. Para evitarlo es recomendable poner espacios delante de cada línea de código, aumentándolo por cada nuevo bloque de código. De esta forma, las instrucciones se encuentran a diferentes distancias del margen dependiendo del bloque en el que se encuentran. A esta técnica, que se ha aplicado a los ejemplos desde el comienzo del curso, se le llama indentado o sangrado.

El indentado más habitual consiste en 4 espacios en blanco por cada nivel de anidamiento.

El concreto para los ejemplos de este curso, todos los fragmentos de código que se han incluido hasta ahora se han puesto con 2 espacios en blanco de indentado, para facilitar su lectura sin tener que ocupar demasiado espacio en el documento.

Ejemplo 2.4.15. Este código es idéntico al anterior pero sin indentado. Es obvio que este es mucho más difícil de comprender el flujo que sigue el programa.

```
<?php
for ($i = 1; $i <= 10; $i++) {
if (($i % 2) == 1) {
echo "El " . $i . " es un número impar.";
echo "<br>";
} else {
echo "El " . $i . " es un número par.";
echo "<br>";
}
}
?>
```

Indentar no es obligatorio, ya que el código funcionaría igual sin hacerlo pero, como se ha visto en este breve ejemplo, cualquier programa independientemente de su complejidad o tamaño se vuelve prácticamente ilegible.

Ejemplo 2.4.16. Mediante el anidamiento se pueden conseguir resultados interesantes con muy poco código. En el siguiente ejemplo se va a dibujar una tabla de multiplicar.

```
<table>
<?php
  for ($i = 1; $i <= 10; $i++) {
    echo "<tr align='right'>";
    for ($j = 1; $j <= 10; $j++) {
      echo "<td width='25'>" . ($i*$j) . "</td>";
    }
    echo "</tr>";
  }
?>
</table>
```

2.4.4 Terminación de los bucles.

Si no controlamos bien la condición del bucle, ésta nunca se evaluará a falso y el bucle se ejecutará indefinidamente, dando lugar a un bucle infinito.

Éste es uno de los errores más habituales, ya que en bucles complejos puede ser fácil perder de vista cuales son las variables que intervienen en la evaluación de la condición.

Ejemplo 2.4.17. Éste es un bucle infinito que escribirá indefinidamente una frase por pantalla. El bucle continuará ejecutándose hasta que paremos la descarga desde el navegador, o nos quedemos sin memoria.

```
<?php
  while (TRUE) {
    echo "Hasta el infinito ";
    echo "y más allá...";
    echo "<br>";
  }
?>
```

2.4.4.1 Break.

La ejecución natural de los bucles se puede interrumpir mediante la sentencia *break*.

En el cuerpo de cualquiera de los tipos de bucle que hemos visto, una sentencia *break* causa la finalización del mismo sin ejecutarse ninguna sentencia más del cuerpo del bucle. En otras palabras, en cuanto se encuentra una sentencia *break* se interrumpe la ejecución del bucle y se pasa a la siguiente línea de código después del bucle.

Ejemplo 2.4.18. En este caso, hemos añadido una sentencia *break*, al bucle infinito que utilizamos en el anterior ejercicio.

```
<?php
  while (TRUE) {
    echo "Hasta el infinito";
    break;
    echo " y más allá...";
    echo "<br>";
  }
?>
```

Ejemplo 2.4.19. En ocasiones se pone intencionadamente un bucle infinito que se pretende romper con una sentencia *break* en su cuerpo.

```
<?php
  define(ALERTA_MISION, 5);
  $nivel_alerta = 1;
  while (TRUE) {
    echo "Hasta el infinito";
    if ($nivel_alerta == ALERTA_MISION) {
      break;
    }
    echo " y más allá...";
    echo "<br>";
    $nivel_alerta++; // Se incrementa en uno
  }
  echo "<br>";
  echo "Parece que hay alguien en apuros.";
?>
```

2.4.4.2 Continue.

La sentencia *continue* tiene cierta similitud con la sentencia *break*.

Su funcionamiento es el siguiente: en cuanto se encuentra una sentencia *continue* se ignoran el resto de las instrucciones del cuerpo del bucle pero no se sale del mismo, sino que se salta a la sentencia de evaluación de la *condición*. Obviamente si en ese momento la *condición* del bucle es verdadera el bucle comenzará una nueva ejecución.

Ejemplo 2.4.20. Este programa lista los números del 1 al diez, e indica los que son múltiplos por 3, en cuyo caso nos muestra el resultado de dicha división.

```
<?php
  for ($i = 1; $i <= 10; $i++) {
    echo "El número " . $i;
    if (($i % 3) != 0) {
      echo " no es multiplo de 3.";
      echo "<br>";
      continue;
    }
    echo " es multiplo de 3. ";
    echo "Por lo tanto ";
    echo $i . " dividido entre 3 es igual a " . ($i / 3);
    echo "<br>";
  }
?>
```

Las sentencias *break* y *continue* dan lugar por lo general a un código más complejo y propenso a errores, por lo que no se recomienda su uso por parte del programador principiante.