Estructuras de control en PHP

Las estructuras de control en PHP permiten controlar cómo el código fluye en un script

Contenido modificable

Si ves errores o quieres modificar/añadir contenidos, puedes crear un pull request. Gracias

Un script PHP está contruido a partir de **sentencias**. Una sentencia puede ser una asignación, una llamada a una función, un loop, una sentencia condicional o una sentencia vacía. Las sentencias normalmente finalizan con un punto y coma.

Las **estructuras de control** son sentencias que permiten controlar cómo el código fluye en nuestro script basándose en ciertos factores. Por ejemplo, cuando queremos realizar una acción sólo si cierta variable está definida, o cuando queremos mostrar un **array** de datos a través de un **loop**.

Las estructuras de control son mayoritariamente **condicionales** (if, switch, etc) o **loops** (for, foreach, etc).

Indice de contenido

6. for	14. require/require_once
7. foreach	15. goto
8. break	16. Sintaxis alternativas

1. if

La estructura de control **if** permite la ejecución condicional de fragmentos de código.

```
<?php
if ($x > $y) {
    echo "$x es mayor que $y";
}
```

Las sentencias if se pueden incluir unas dentro de otras indefinidamente.

2. else

Sirve para ejecutar una sentencia cuando otra no se cumple. *else* extiende una sentencia *if*, y se ejecuta cuando ésta es **false**. Siguiendo el ejemplo anterior:

```
<?php
if ($x > $y) {
    echo "$x es mayor que $y";
} else {
    echo "$y es mayor que $x";
}
```

3. elseif/else if

Es una combinación entre *if* y *else*. Se ejecuta cuando if es **false**, pero sólo si la expresión condicional del *elseif* es **true**.

```
<?php
if ($x > $y) {
    echo "$x es mayor que $y";
} elseif ($x == $y) {
    echo "$x es igual que $y";
} else {
    echo "$y es mayor que $x";
}
```

Puede haber varios *elseif* dentro de una sentencia *if* (aunque en ese caso suele ser más recomendable usar *switch*). No hay diferencias de comportamiento entre *elseif* y *else if*.

evaluadas como **false**.

4. while

Es el tipo más sencillo de **loop** en **PHP**. Se ejecutan las sentencias dentro del ***while*** siempre y cuando se evalúen como **true**. El valor de la expresión se comprueba cada vez al inicio del **loop**, y la ejecución no se detendrá hasta que finalice la **iteración** (cada vez que PHP ejecuta las sentencias en un loop es una iteración). Si la expresión ***while*** se evalúa como **false**, las sentencias no se ejecutarán ni siquiera una vez.

También es posible agrupar varias instrucciones ***while*** dentro de una.

```
<?php
$i = 1;
while($i <= 10){
    echo $i;
    $i++;
}
```

5. do-while

Muy similares a los loops ***while***, simplemente aquí la expresión para el loop se verifica al final en lugar de al principio, esto garantiza que el código se ejecute por lo menos la primera vez.

```
<?php
$i = 0;
do {
    echo $i;
} while ($i > 0);
```

Este loop se ejecutaría sólo una vez, ya que después no cumple la condición.

6. for

Los loops ***for*** son los más complejos en PHP.

```
<?php
for ($i = 1; $i <= 10; $i++) {
    echo $i;
} // Devuelve 123456789
```

- Las expresiones o conjunto de expresiones van separadas por punto y coma **;** y sólo hay 3.

- La segunda expresión, `$i <= 10`, es una **condición**, si es true, se ejecutará la tercera expresión.
- La tercera expresión, `$i++`, es la acción a realizar si se cumple la segunda expresión.

Cada una de las expresiones pueden estar **vacías** o contener **múltiples expresiones**, lo que resulta útil en ciertas ocasiones. Si la expresión 2 está vacía, el loop será definido como **true**.

```
<?php
// Todos los siguientes ejemplos son válidos y devuelven lo mismo, 123456789
// EJEMPLO 1
for($i = 1; $i <= 10; $i++) {
    echo $i;
}

// EJEMPLO 2
for ($i = 1; ; $i++){
    if($i > 10) {
        break;
    }
    echo $i;
}

// EJEMPLO 3
$i = 1;
for( ; ; ){
    if($i > 10){
        break;
    }
    echo $i;
    $i++;
}

// EJEMPLO 4
for ($i = 1, $j = 0; $i <= 10; $j += $i, print $i, $i++);
```

Dependiendo de donde se coloque el operador **incremento** o **decremento**, dará un resultado distinto:

```
for ($i = 1; $i <= 10; print $i++); // 12345678910
for ($i = 1; $i <= 10; print ++$i); // 234567891011
```

Se pueden hacer **iteraciones por medio de arrays**:

```
$gente = array(
    array('nombre' => 'Carlos', 'salt' => 123),
    array('nombre' => 'Maria', 'salt' => 124));
```

```
// Loop que da un salt de 4 digitos aleatorio
for ($i=0; $i < $size; ++$i) {
    $gente[$i]['salt'] = mt_rand(0000, 9999);
}
var_dump($gente);
/*
* array (size=2)
  0 =>
    array (size=2)
      'nombre' => string 'Carlos' (length=6)
      'salt' => int 2029
  1 =>
    array (size=2)
      'nombre' => string 'Maria' (length=5)
      'salt' => int 9724
*/
```

7. foreach

foreach permite una forma fácil de iterar sobre arrays u objetos.

Cuando *foreach* inicia su ejecución, el puntero apunta directamente al primer elemento del array, por lo que no es necesario llamar a la función *reset()* antes que un loop *foreach*. Es recomendable no cambiar el puntero dentro del loop.

Se puede iterar de las siguientes dos formas:

```
<?php
// Devuelve directamente el value de cada key, comenzando desde el primero
foreach ($array as $value) {
    // Lo que se desee mostrar
}
// Devuelve cada key con cada value, para poder trabajar con cualquiera de los dos
foreach ($array as $key => $value){
    // Lo que se desee mostrar
}
```

Se pueden modificar los elementos del array dentro del loop, anteponiendo & a *$value* (asignándolo por referencia).

```
<?php
$array = array(1, 2, 3, 4);
foreach ($array as &$value){
    $value = $value * 2;
}
// cada valor del array vale ahora : 2, 4, 6, 8
unset($value);
```

y el último elemento del array permanecen aún después del loop foreach.

foreach y *list* pueden devolver los mismos resultados de la siguiente forma:

```
<?php
$array = array("uno", "dos", "tres");
reset($array);
while (list($clave, $valor) = each($array)){
    echo "Clave: $clave; Valor: $valor </br>";
}

foreach ($array as $clave => $valor){
    echo "Clave: $clave; Valor: $valor </br>";
}
```

La función *list()* coge un *array* y lo convierte en **variables** individuales.

each() coge un *array* y devuelve el key y value actuales, es decir, donde está apuntando el cursor en el array. Es necesario hacer un *reset()* del *array* para asegurarse que el cursor comienza desde el principio (cosa que no es necesaria con el foreach).

Suele ser más óptimo y legible utilizar el *foreach* para iterar, aunque ambos se utilizan más o menos con la misma frecuencia.

Con **arrays multidimensionales** se puede hacer un **doble foreach**:

```
<?php
$x = array();
$x[0][0] = "a";
$x[0][1] = "b";
$x[1][0] = "y";
$x[1][1] = "z";

foreach ($x as $primero){
    foreach ($primero as $segundo){
        echo "$segundo" . "</br>";
        }
    }
```

En el ejemplo anterior, si no se hace doble foreach con *$x* y se hace sólo un foreach surgirá un *Notice: Array to string conversion*.

Desde PHP 5.5 se puede recorrer un **array de arrays** y utilizar el segundo array para asignar variables:

```
    array('azul', 'rojo'),
    array('verde', 'amarillo')
    );

foreach ($array as list($a, $b)) {
    echo "A: $a; B: $b" . "</br>";
}
// Devuelve:
```

8. break

break termina la ejecución de las siguientes estructuras: *for*, *foreach*, *while*, *do-while* y *switch*.

Se puede añadir un **argumento numérico opcional** que indica de cuántas estructuras debe salir. El valor por defecto es 1:

```
// Ejemplo sin valor numérico
$array = array('uno', 'dos', 'parar', 'tres');
while(list(, $valor) = each($array)){
    if($valor == 'parar'){
        break;
    }
    echo "$valor</br>";
}
// Ejemplo con valor numérico
$i = 0;
while (++$i){
    switch($i){
        case 5:
            echo "He llegado a 5 </br>";
            break 1; // Aquí sólo saldría del switch
        case 10:
            echo "He llegado a 10 </br>";
            break 2; // Sale del switch y del while
        default:
            break;
    }
}
```

9. continue

Se utiliza **dentro de las estructuras iterativas** para **saltar** el resto de la iteración actual del loop y **continuar a la siguiente iteración** para su ejecución:

```
<?php
for ($i=0; $i < 10; $i++) {
    if($i % 2 == 0)
        continue;
```

El código anterior hace que cuando el número sea par, *continue* se ejecute y no muestre el print de después, sino que vuelva a empezar con la siguiente iteración. *print* sólo se ejecutará cuando *$i* sea impar.

Al igual que con *break*, se puede añadir un número a *continue* que indica el número de niveles de loops debe saltar:

```
<?php
$i = 0;
while ($i++ < 5){
    echo "Uno </br>";
    while (1) {
        echo "Dos </br>";
        while (1) {
            echo "Tres </br></br>";
            continue 3;
        }
        echo "Esto no aparece.";
    }
    echo "Esto tampoco aparece.";
}
```

En este ejemplo, cuando llega a *continue 3*, comienza de nuevo la iteración, y se comprueba otra vez la condición ($i++ < 5). Si se pone, por ejemplo, *continue 2*, salta al segundo while, cuya condición siempre se cumple, y se produce un loop infinito que imprime "Dos Tres" continuamente.

10. switch

switch es como una serie de sentencias *if*. Es útil para comparar una misma variable o expresión con valores direrentes y ejecutar un código diferente a otro dependiendo de esos valores.

```
<?php
switch ($i) {
    case "perro":
        echo "\$i es un perro";
        break;
    case "gato":
        echo "\$i es un gato";
        break;
    case "avestruz":
        echo "\$i es un avestruz";
        break;
}
```

final o hasta que choca con un *break*, que entonces finaliza la iteración. Si se omite *break*, swith ejecutará todos los *cases* restantes cuando encuentra uno que cumpla con la condición:

```
<?php
switch ($i) {
    case 0:
        echo "i es igual a 0";
    case 1:
        echo "i es igual a 1";
    case 2:
        echo "i es igual a 2";
}
```

En el ejemplo anterior, si se define a *$i* como 0, se mostrarán todos los *echo* restantes porque no hay ningún *break*. Si se define a *$i* como 1, se mostrarán los *echo* del *case* 1 y *case* 2. Si se define a *$i* como 2, sólo se mostrará el *case* 2.

Con *switch* la condición sólo se evalúa una vez, y su valor es comparado con cada uno de los *case*, a diferencia de lo que ocurre con *elseif*, que la condión se va evaluando continuamente con el loop.

case puede no ejecutar ningún código, pero hace que se activen los cases posteriores hasta que se encuentre con un break:

```
<?php
switch ($i) {
case 0:
case 1:
case 2:
    echo "i es menor que 3 no negativo";
    break;
case 3:
    echo "i es 3";
}
```

En caso de que no haya ningún *case* válido, puede utilizarse *default*, para ejecutar algo cuando no se cumplen los case:

```
<?php
switch($i) {
    case 0:
        echo "i es igual a 0";
        break;
    case 1:
        echo "i es igual a 1";
        break;
    default:
```

- En los cases sólo se permiten **tipos simples**: *int*, *float* y *string*. Los **arrays** y **objetos** pueden utilizarse si se muestran como un tipo simple.
- Es posible escribir punto y coma ";" en lugar de dos puntos ":" después de un *case*.

11. declare

declare sirve para fijar directivas de ejecución para un bloque de código.

```
declare (directive)
    statement
```

La sección directive permite configurar el comportamiento de declare. Sólo hay dos directivas: *ticks* y *encoding*.

- No se pueden emplear variables ni constantes para las directivas.

Se pueden usar corchetes para fijar el statement o si no se aplican afecta a todo el archivo:

```
<?php
declare(ticks=1){
   // Código afectado
}

declare(ticks=1);
// Código afectado
```

En este segundo ejemplo si el archivo con el declare ha sido incluído, no afectará al archivo padre.

Ticks

La ejecución de un **script PHP** puede representarse como una ejecución de muchas sentencias. La mayoría de las sentencias provocan un **tick** (aunque no todas). Por ejemplo en el siguiente ejemplo habría 3 ticks:

```
<?php
$x = 1;
$y = 2;
$z = 3;
```

Con **declare(ticks=N)** y _register_tick_function()_ se puede ejecutar código entre los **ticks**. La función _register_tick*function* especifica la función a llamar cuando

- Con declare(ticks=1) y _register_tick*function('miFuncion')*; llamará a *miFuncion()* después de cada tick.
- Con declare(ticks=3), *miFuncion()* se ejecutará cada 3 ticks.

```
<?php
function handler(){
    echo "Hola";
}

register_tick_function("handler");
$i = 0;
declare(ticks = 3) {
    while($i < 9)
        echo ++$i;
} // Devuelve 123Hola456Hola789Hola
```

Encoding

La codificación puede ser especificada para cada script con encoding:

```
<?php
declare(encoding='ISO-8859-1');
// Código a ejecutar
```

12. return

return devuelve el control del programa al módulo que lo invoca. La ejecución vuelve a la siguiente declaración después del módulo que lo invoca.

- Si se usa en una función, return hace que la función termine, devolviendo los argumentos que le sigan como valor de la llamada a la función.
- Si se llama globalmente, finaliza la ejecución del script actual.
- Si el archivo actual fue incluido o requerido, el control regresa al archivo que llama al *include* o *require*.
- Si el archivo está incluído con *include*, los argumentos del return se devolverán como valor de la llamada *include*.
- Si return se usa en el archivo principal del script, termina la ejecución del script.
- También termina la ejecución de una sentencia *eval()*.

Es recomendable no usar paréntesis después de return.

13. include/include_once

Los archivos se incluyen en base a la ruta de acceso dada, y si no se proporciona ninguna, se utiliza el include_path. Si el archivo tampoco se encuentra en el include_path se mirará en el propio directorio desde donde se hace la llamada, antes de devolver un mensaje warning. Es en el tipo de mensaje donde difiere con require, que devuelve un error fatal.

- Si se define una ruta absoluta (en Linux comenzando por /) o relativa al directorio actual (comenzando con . o ..) el include_path será ignorado.

Las variables del archivo del include estarán disponibles en el archivo desde el que se solicita:

```
<?php
// archivo1.php
$color = 'azul';
// archivo2.php
echo $color; // Devuelve un Notice: Undefined variable: color
include 'archivo1.php';
echo $color; // Devuelve azul
```

- Si la inclusión se hace dentro de una función, el contenido del archivo es como si estuviera dentro de esa función, por tanto su contenido tendrá el mismo ámbito.
- Cuando se incluye un archivo, el intérprete abandona el modo PHP e ingresa al modo HTML al comienzo del archivo incluído, y se reanuda de nuevo al final. Es por ello que cualquier código que tenga que ser interpretado como PHP debe incluir las etiquetas válidas de comienzo y terminación (<?php ?>).

Si están activadas las envolturas URL include, se puede incluir un archivo a través de una URL (mediante HTTP u otro protocolo). Si el servidor objetivo interpreta el archivo como PHP, las variables se pueden pasar al archivo usando un string de petición como con HTTP GET. El resultado no es lo mismo que en local, pues el archivo se ejecuta en el servidor remoto y el resultado se incluye en el local.

```
<?php
include 'http://www.ejemplo.com/archivo.php?var=1';
$var = 1;
```

Por seguridad es mejor que el archivo remoto se procese en el servidor remoto y se reciba sólo la salida, con *readfile()*.

Es posible devolver valores desde los archivos include mediante return:

```
$var = 'PHP';
return $bar;
?>
// archivo2.php sin return
<?php
$var = 'PHP';
?>
//archivo3.php
<?php
$foo = include 'archivo1.php';
echo $foo; // devuelve PHP

$bar = include 'archivo2.php';
echo $bar; // Devuelve 1 porque el include ha sido exitoso, pero no tiene valor re
```

También se pueden incluir archivos PHP en variables con un buffering de salida:

```
<?php
$string = get_include_contents('archivo.php');

function get_include_contents($filename) {
    if(is_file($filename)) {
        ob_start();
        include $filename;
        return ob_get_clean();
    }
    return false;
}
```

Para incluir archivos automáticamente en scripts ver auto_preprend_file y auto_append_file de *php.ini*.

_include_once_ incluye el archivo especificado sólo una vez, si se incluye más veces tan sólo devuelve true. Es útil en casos donde el mismo fichero se podría incluir y evaluar más de una vez, para evitar así redefinir funciones o reasignar valores a variables.

14. require/require_once

require hace lo mismo que *include* pero en caso de fallar devuelve un error fatal de nivel E_COMPILE_ERROR, por lo que no puede continuar el script. *include* sólo emite un E_WARNING que permite continuar el script.

_require_once *es igual que* require_ pero PHP comprobará si el archivo ya ha sido incluído, y si es así no se incluirá otra vez.

15. goto

```php
<?php
goto x;
echo 'Hola!';

x:
echo 'Adios!';  // sólo se imprimirá Adios!
```

También puede utilizarse en un **loop** en lugar de *break*:

```php
<?php
for($i=0, $j=50; $i<100; $i++) {
    while($j--) {
        if($j==17) goto end;
        }
    }
echo "i = $i";
end:
echo 'j llegó a 17';
```

16. Sintaxis alternativas

PHP ofrece una **sintaxis alternativa** para *if*, *while*, *for*, *foreach* y *switch*. Se cambia el corchete de apertura por dos puntos ":" y el corchete de cierre por *endif*, *endwhile*, *endfor*, *endforeach*, o *endswitch*.

```php
<?php if ($x == 3): ?>
X es 3
<?php endif; ?>
```

"X es 3" es un bloque HTML que se mostraría sólo si se cumple la condición.

Con **else** y **elseif**:

```php
<?php
if($x == 3):
    echo "x igual a 3";
elseif ($x == 4):
    echo "x igual a 4";
else:
    echo "x no es ni 3 ni 4";
endif;
```