Tema 4
Sentencias de la programación estructurada

Generalmente, las instrucciones de un programa se ejecutan una tras otra, a no ser que alguna condición nos lleve a otro punto del código. Teniendo esto en cuenta, podríamos decir que la programación se basa en el flujo de ejecución de un programa. Sin embargo, los primeros lenguajes introducían instrucciones de salto incondicional, como **goto**, que hacían que los lenguajes fuesen poco legibles y difíciles de comprender.

A finales de los años 60, surgió una nueva forma de programar que, no sólo daba lugar a programas fiables y eficientes, sino que además estaban escritos de manera que facilitaban su comprensión posterior. Para conseguir estos objetivos, se definieron una serie de criterios de diseño claros y bien definidos, que permitían hacer frente a la complejidad creciente de las aplicaciones.

Este fue el nacimiento de la **programación estructurada**, cuyos pilares se asientan el teorema de Böhm-Jacopini, refinado años más tarde por el holandés Edsger Dijkstra, que demuestra que todo programa puede escribirse utilizando únicamente tres estructuras de control: **secuencia**, **selección** e **iteración**. Los programas así diseñados y que no hacen uso de la sentencia **goto**, se denominan **estructurados**, y pueden leerse de forma secuencial, sin dificultad para seguir la traza de la tarea.

Actualmente, la programación estructurada no se basa únicamente en el uso de estructuras de control, sino también de segmentos de código que resuelven tareas concretas, dividiendo el problema global en partes más manejables. Si hacemos una segmentación apropiada, los diferentes segmentos se comunicarán entre sí de forma fluida y controlada, produciendo una mínima o nula dependencia (**acoplamiento**) entre ellos. Esta técnica de programación conlleva una serie de ventajas.

a. Facilita el trabajo simultáneo de distintos programadores.
b. Los cambios efectuados en uno de estos módulos durante la fase de desarrollo o mantenimiento no afectará a otros puntos del programa.
c. Facilita la reutilización del código.
d. Dividir el problema original en subtareas aisladas facilita la tarea de desarrollo.

Sentencias de selección

También conocidas como **sentencias condicionales** o **de toma de decisión**, permitirán ejecutar diferentes conjuntos de instrucciones en base al resultado obtenido al evaluar una condición. De esta manera, este tipo de sentencias redirigen el flujo de ejecución del programa hacia un conjunto de instrucciones u otro, en base a una determinada condición impuesta a la sentencia. Encontramos cuatro tipos de sentencias de selección: **simples**, **dobles**, **compuestas** o **múltiples**.

Sentencia IF

Usamos la sentencia if para tomar una decisión. Obviamente, se tomará una decisión en función de una condición impuesta a la sentencia, que deberá estar escrita entre paréntesis, y podrá tomar como posibles valores **verdadero** o **falso**.

Básicamente, una sentencia **if** podría interpretarse como: "*si se cumple esta condición haz esto y si no, haz esto otro*". Supongamos que hemos construido un programa que pregunta al usuario si es mayor de edad. La condición para ser mayor de edad es tener 18 o más edad. Si dicha condición se cumple, se mostrará un determinado mensaje de bienvenida.

```
if ($edad >= 18) echo "Bienvenido a nuestra web!" ;
```

Si necesitamos realizar un conjunto de sentencias más extenso cuando se cumpla la condición, tendremos que agruparlas encerrándolas entre llaves, creando de esta manera un bloque de código. Volviendo a nuestro ejemplo anterior, supongamos que queremos añadir algo de estilo al mensaje.

```
if ($edad >= 18) {
  echo "<p style=\"font-weight:bold;\"> ;
  echo "¡Bienvenido a nuestra web!" ;
  echo "</p>" ;
}
```

De esta manera, cuando la condición es cierta, se ejecutarán las tres instrucciones contenidas en su cuerpo. Las **sentencias de selección dobles** añaden a la anterior la posibilidad de realizar un conjunto de acciones en caso de que la condición no se cumpla, y antes de continuar con el resto del programa. La sentencia **else** nos permite tomar esta alternativa.

Supongamos ahora que, si el usuario no es mayor de edad debemos mostrarle un mensaje de despedida, además de redireccionarlo a otra página, cosa que más adelante veremos cómo se hace.

```php
if ($edad >= 18) {
  echo "<p style=\"font-weight:bold;\"> ;
  echo "¡Bienvenido a nuestra web!" ;
  echo "</p>" ;
} else {
  echo "Este no es lugar para tí, intruso... <br/>" ;
}
```

Como vemos, estas sentencias permiten variar el flujo de ejecución en base a la evaluación de una expresión lógica, y además nos permiten la posibilidad de construir estructuras complejas utilizando mecanismos de **anidamiento**. Imaginemos que ahora queremos que aquellos usuarios accedan a diferentes secciones de la web en función de su edad; no sucederá igual con los menores de 18, que seguirán teniendo prohibida la entrada. Conseguimos todo esto, anidando una sentencia **if** dentro de la anterior.

```php
if ($edad >= 18) {
  if ($edad >= 65) {
    echo "Esta es la sección para jubilados.<br/>" ;
  } else {
    echo "Esta es la sección para jóvenes.<br/>" ;
  }
} else {
  echo "Este no es lugar para tí, intruso...<br/>" ;
}
```

Observamos que hemos introducido (anidado) una sentencia **if** dentro de otra, y si interpretamos su significado, resuelve el problema que teníamos entre manos. Esto es, la edad del usuario es mayor o igual a 18, tendremos dos nuevas posibilidades. Si es mayor o igual de 65, se mostrará un mensaje en pantalla; en otro caso, se mostrará otro diferente. Por último, si el usuario no es mayor de edad y no se cumple la primera condición, el flujo de ejecución del programa continuará en la rama **else** del primer **if**. Obviamente, el fragmento de código anterior puede mejorar: hazlo antes de continuar.

Sea como fuere, la legibilidad deberá ser siempre una de nuestras prioridades, por lo que deberemos dejar de lado los anidamientos si el uso de estos complica demasiado el código; en situaciones como esta, utilizaremos, si es posible, **encadenamientos**, dando lugar a **sentencias de selección compuestas**.

```php
if ($edad >= 65) {
  echo "Esta es la sección para jubilados<br/>" ;
elseif ($edad >= 18) {
  echo "Esta es la sección para jóvenes.<br/>" ;
else {
  echo "Este no es lugar para tí, intruso...<br/>" ;
}
```

La flexibilidad de PHP nos permite escribir **elseif**, o **else if**. Sea como fuere, debe recordarse que únicamente se ejecutará una de las ramas de la sentencia. Volvamos al ejemplo anterior. Si hubiésemos dejado **$edad >= 18** como primera condición, siempre que el valor de la variable fuese mayor o igual que 18, se habría mostrado el mensaje **Esta sección es para jóvenes**. Sin embargo, al cambiar el orden de las condiciones, hacemos que éstas sean mutuamente excluyentes, de manera que garantizamos que sólo una de ellas sea cierta.

Operador ternario

Se conoce con el nombre de operador ternario al operador condicional ?:, que nos permitirá construir expresiones del tipo:

```
(expr1) ? (expr2) : (expr3)
```

La sentencia anterior evaluará **expr2** si **expr1** tiene como resultado **true**; en otro caso, se evaluará **expr3**. Veamos un ejemplo:

```
echo ($edad >= 18) ? "Mayor de edad" : "Menor de edad" ;
```

Como vemos, la función de este operador es similar a una sentencia **if-else**. Podemos simplificar el operador ternario construyendo la sentencia obviando la **expr2**, de esta manera:

```
(expr1) ?: (expr3)
```

En este caso, si **expr1** es **true** se retorna **expr1** y, en otro caso, se devolverá **expr3**. Esta variante podemos utilizarla únicamente a partir de la versión 5.3 de PHP, y no en las anteriores. El operador abreviado **?:** se conoce como **Elvis**. Se recomienda en todo caso no apilar construcciones de este tipo, ya que, no sólo tendríamos sentencias más complejas, sino que además se vería afectada la legibilidad del código.

Sentencia SWITCH

Hasta ahora hemos estudiado las sentencias de selección **simples**, **dobles** y **compuestas**. PHP incorpora además una sentencia condicional **múltiple**, encargada de comparar sucesivamente el valor de una expresión con una lista de constantes. Cuando se encuentra una correspondencia, se ejecutan las sentencias asociadas.

En una sentencia **if**, la condición puede ser verdadera o falsa. Sin embargo, en la sentencia **switch**, la condición puede tomar cualquier valor de tipo entero, cadena o real. Será necesario el uso de una sentencia **case** para determinar cada una de los posibles valores de de la condición. De esta manera, se irá comprobando de forma ordenada si el valor de la condición coincide con el valor indicado en cada sentencia case. Cuando se encuentra una correspondencia, se ejecutan las instrucciones asociadas, hasta que se encuentra la

sentencia **break**, o se alcance el final del **switch**. Se puede incluir opcionalmente la sentencia **default**. Esta define un bloque de instrucciones que se ejecutarán cuando no se ha encontrado ninguna correspondencia con los valores especificados en cada sentencia **case**.

```
switch($op) {
  case 1:
  case 2:
  case 3:
    echo "El valor está comprendido entre 1 y 3" ;
    break ;
  case 4: $op += 6 ;
  case 5:
    $op *= 2 ;
    echo "El resultado es: $op" ;
    break ;
  default:
    echo "*** ERROR ***" ;
}
```

Técnicamente, las sentencias **break** dentro de la sentencia **switch** son opcionales. Se utilizan para finalizar la secuencia de sentencias asociada con cada constante. Si se omite la sentencia **break**, la ejecución continúa en la siguiente sentencia case hasta que se alcanza una sentencia **break** o el final del **switch**.

En el ejemplo anterior se ilustra lo que acabamos de explicar. Además de esto, se muestra cómo algunas condiciones case pueden no tener sentencias asociadas. Cuando esto ocurre, el flujo de ejecución continúa pasando al siguiente case. Esto último resulta de gran utilidad cuando queremos que se realice una determinada acción en caso de que se verifiquen varias condiciones. Por ejemplo, las sentencias asociadas al tercer **case** se harán siempre que **$op** valga 1, 2 ó 3.

Sentencia MATCH

PHP8 introduce una nueva sentencia que corrige las limitaciones de la anterior: **match**. Ésta se comporta como una expresión permitiéndonos guardar su resultado en una variable, devolverlo o utilizarlo en cualquier sentencia o expresión. A diferencia de **switch**, no es necesario utilizar **break** ya que cada alternativa acepta una expresión de una sóla línea.

```
return match($dia) {
  1, 2, 3, 4, 5 => "día laborable",
  6, 7          => "fin de semana",
  default       => "el valor es erróneo"
} ;
```

Como vemos, la expresión **match** es similar a **switch**, aunque presenta ciertas diferencias.

Tendremos que recordar que cada rama realiza una comparación de manera estricta. En este sentido, el valor “1” será diferente a 1. Además, en una rama podemos separar con comas diferentes expresiones.

NOTA: se lanzará una excepción de tipo **UnhandledMatchError** si no existe una expresión que verifique el predicado.

Sentencias de iteración

Si las sentencias de selección permiten tomar decisiones en función de una condición, las **estructuras iterativas** facilitarán la repetición de un conjunto de acciones tantas veces como se desee. Estos **bucles** o **lazos**, ejecutarán todas las instrucciones contenidas en su cuerpo, en base a una determinada **condición de control** que deberá ser modificada de algún modo en el interior del bucle para garantizar que éste finalice en algún momento.

Sentencia WHILE

La manera más simple de construir un bucle en PHP es utilizando la sentencia **while**. Su forma general es:

```
while ( condición ) { ... }
```

La **condición** será cualquier expresión que provoque la **iteración** del bucle mientras ésta sea cierta. De esta manera, las instrucciones contenidas en el cuerpo de la sentencia **while**, se ejecutarán de forma indefinida, mientras la condición impuesta al bucle sea cierta. Cuando la condición es falsa, el control del programa pasa a la instrucción situada inmediatamente después de la construcción **while**.

```php
$i = 1 ;
while ($i <= 100) {
  echo $i ;
  $i++ ;
}
```

El bucle anterior mostrará en pantalla los números del 1 al 100, ambos incluidos. El bucle **while** comprueba la condición al principio, lo cual implica que, en algunas ocasiones, puede no ejecutarse el cuerpo del bucle.

Sentencia FOR

Generalmente utilizamos una sentencia **while** tal y como hemos visto en el anterior ejemplo, inicializando la **variable de control** antes de entrar al bucle, comprobando la

condición antes de cada iteración, y modificando al final del mismo la variable de control. La sentencia **for** nos permite escribir todo esto de forma más compacta. Su forma general es:

```
for ( expr1 ; condición  ; expr2 ) { ... }
```

La **expr1** corresponde generalmente a la inicialización de la variable de control. A diferencia de lo que ocurría con la sentencia **while**, y aunque es posible hacerlo, se recomienda encarecidamente no modificar en el cuerpo de este bucle el valor de dicha variable de control. Generalmente, la encargada de esta tarea será la **expr2**, que se ejecutará siempre al final de cada iteración. Reescribimos el bucle del ejemplo anterior utilizando la sentencia **for**.

```
for($i=1; $i<=100; $i++)  echo $i ;
```

Funcionalmente, ambos bucles (**while** y **for**) son idénticos. Ninguno es mejor que otro, aunque sí es cierto que cada uno de ellos es más útil según qué situación. Por ejemplo, un bucle while suele ser más habitual cuando desconocemos *a priori* el número de iteraciones. Sin embargo, la estructura **for** se utiliza cuando sabemos exactamente el número de repeticiones del bucle. Aunque esto no es siempre así.

Es posible que en alguna ocasión si, por ejemplo, el valor inicial de la variable de control viene dado por una entrada de teclado, no será necesario inicializarla en el bucle **for**. En este caso, tendríamos una construcción similar a la que sigue.

```
for( ; $i <= 100; $i++) echo $i ;
```

En resumen, podemos obviar cualquiera de las secciones de definición de un bucle for, hasta el punto de no especificar ninguna de ellas. En tal caso, estaríamos definiendo un **bucle infinito**.

```
for (;;) {...}
```

¿Sería posible hacer esto con un bucle **while**? ¿Cómo? Tengamos en cuenta además, que un bucle **for** puede leerse igual que su homólogo **while**: *Dado un valor inicial, mientras se cumpla la condición ejecutamos un conjunto de instrucciones.* De esta manera, la condición de este tipo de sentencias puede servirnos incluso para buscar fácilmente un valor en un *array*.

```
for ($i=0; $vec[$i]!=5; $i++) ;
```

Observamos que el cuerpo de la sentencia no incluye ningún conjunto de instrucciones a ejecutar. ¿Es esto posible? Sí, lo es. En el ejemplo anterior no es necesario realizar ningún tipo de operación, basta con recorrer el *array* hasta que el valor de la celda examinada sea igual al que estamos buscando.

Sentencia DO ... WHILE

A diferencia de las estructuras anteriores, la sentencia **do ... while** examina la condición del bucle al final. Esto significa que un bucle de este tipo, ejecutará las instrucciones de su cuerpo al menos una vez. Su estructura general es:

```
do { ... } while ( condición ) ;
```

Aunque con **for** y **while** no son necesarias las llaves cuando tan solo hay una sentencia en el cuerpo del bucle, en la estructura **do ... while** son siempre necesarias. Además, debemos tener en cuenta el punto y coma que encontramos tras la condición, que deberá aparecer siempre de manera obligatoria. En este caso, las instrucciones del bucle se ejecutarán mientras sea cierta la condición impuesta.

Sentencia FOREACH

Aunque, como hemos dicho en varias ocasiones, estudiaremos más detenidamente los *arrays* en temas sucesivos, introducimos en este apartado la construcción **foreach**, que nos proporciona un método sencillo de iteración sobre *arrays* u objetos.

```
foreach ( expr1 as $valor) { ... }
```

En esta primera, el bucle recorrerá el *array* indicado en **expr1** y, en cada iteración, el valor del elemento actual se asignará a la variable **$valor**. Automáticamente, el puntero interno del *array* se irá desplazando a lo largo de este, por lo que, tras cada iteración, apuntará a la celda siguiente.

Supongamos un *array* **$vector** que contiene los elementos [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]. El siguiente bucle mostrará en pantalla cada uno de esos valores en líneas diferentes.

```
foreach ($vec as $valor) { echo "$i<br/>" ; }
```

Encontramos una variante a la forma general de **foreach**, que escribiremos como sigue:

```
foreach ( expr1 as $clave => $valor) { ... }
```

Ahora, además de almacenar en **$valor** el valor contenido en cada celda del *array*, también guardaremos en **$clave** el índice de la celda en que nos encontramos en cada iteración.

Anexo

No resulta extraño encontrarnos bibliografía donde se utilice una sintaxis alternativa a

todas y cada una de las sentencias utilizadas. Reemplazamos en este caso el símbolo **{** que indica el comienzo del cuerpo, por dos puntos (**:**), y la **}** de cierre por una nueva palabra reservada, que dependerá de la sentencia que estemos utilizando: **endif**, **endswitch**, **endwhile**, **endfor** o **endforeach**. Obviamente, esta alternativa no afecta a la estructura **do ... while**.

Bibliografía

PHP.net
Guía oficial del lenguaje

PHP and MySQL Web Development
Luke Welling, Laura Thompson
Developer's Library

Learning PHP7
Antonio López
Packt Publishing