Tema 5
Funciones y Recursividad

Los lenguajes de programación nos van a permitir escribir un programa de forma modular, permitiéndonos desarrollar una única vez fragmentos de código que, posteriormente podrán ser reutilizados en cualquier otro punto de nuestro programa.
Para ello, afrontamos nuestros proyectos tomando como base un **diseño descendente**, donde el problema inicial se dividirá en subproblemas o módulos, para conseguir una resolución más sencilla del mismo. Al código del módulo raíz o principal le denominaremos **algoritmo principal**, en tanto que al código correspondiente a cada uno de los restantes módulos les llamaremos **subprograma** o **subalgoritmo**.

Sin duda, este enfoque proporciona numerosas ventajas. No solo **simplifica el diseño** de nuestras aplicaciones, sino que además **facilita la comprensión** de las mismas, permitiendo la **programación aislada** y la **reutilización del código**. Como se ha visto, la modularidad nos permite aislar o **encapsular** las diferentes tareas que componen un programa, permitiéndonos alterar el método de resolución de una tarea, evitando que dicho cambio influya en cualquier otra tarea.

Definición y llamada

Los subalgoritmos, dependiendo de su definición y uso, podemos dividirlos en **procedimientos** y **funciones**. Los primeros se definen como un subprograma donde una llamada al mismo, por sí sola constituye una acción en el cuerpo del algoritmo llamante. Generalmente, el objetivo de un procedimiento es el de realizar una acción. Sin embargo, una función devolverá un valor al algoritmo llamante justo antes de terminar.

En PHP, al igual que en lenguajes como C, Java o JavaScript, no diferenciamos entre ambos, por lo que a partir de ahora, cualquier subprograma en PHP será para nosotros una función. Su forma general será:

```
function nombre_funcion(par1, par2, ... ) { ... }
```

Utilizamos la palabra **function** seguida de un identificador o nombre de la función, que deberá seguir las reglas de nombrado habituales. Teniendo en cuenta que éstos no siempre

serán necesarios, a continuación y entre paréntesis, una lista de parámetros separados por comas.

```
function mostrarMenu() {
  echo "0. Salir.<br/>" ;
  echo "1. Introducir datos.<br/>" ;
  echo "2. Editar perfil.<br/>" ;
  echo "3. Borrar perfil<br/>" ;
}
```

Provocar la ejecución de cualquier función desde otro punto de nuestro código, bastará con invocar el nombre de dicha rutina y proporcionarle los parámetros necesarios.

```
nombre_funcion(par1, par2, ... )
```

Al invocar a una función se crean las variables especificadas en la zona de declaraciones y el flujo de ejecución continuará por la primera instrucción del cuerpo del subprograma llamado. Podríamos decir que la función del ejemplo anterior es un procedimiento, ya que realiza una determinada acción pero no proporciona un valor de vuelta al algoritmo llamante. Sin embargo, una función propiamente dicha deberá retornar al final de la misma, un determinado valor. De esta manera, la última instrucción que deberá ejecutarse es **return**, que devolverá de forma inmediata el control al algoritmo llamante, proporcionándole un valor como resultado.

```
function sumar($a, $b) {
  $c = $a + $b ;
  return $c ;
}
```

Cualquier instrucción escrita dentro de la función y a continuación del **return** no se ejecutará jamás. Como una función puede devolver cualquier valor al algoritmo que ha invocado su ejecución, esto obliga a almacenar, o utilizar dicho valor en aquellas expresiones en que se emplearían valores de igual tipo. Por este motivo se dice que la llamada a una función, por sí misma, no puede constituir una sentencia del algoritmo llamante; debe aparecer dentro de alguna expresión del algoritmo principal en la que el valor devuelto por la función sea empleado de alguna manera. Para el ejemplo anterior, serían válidas las siguientes llamadas a la función.

```
$resultado = sumar(5, 9) ;
echo "El resultado es igual ". sumar(5, 9)."<br/>" ;
printf("El resultado es igual a %d <br/>", sumar(5, 9)) ;
```

Sin embargo, aunque no generará ningún error, la siguiente llamada no será válida, ya que el valor devuelto por la función no se utiliza en ninguna expresión.

```
sumar($a, $b) ;
```

Parámetros

Los argumentos o parámetros empleados en las llamadas a subrutinas, reciben el nombre de **parámetros reales**, en tanto que los empleados en la definición de la función se conocen como **parámetros formales**. Ambos constituyen lo que se conoce como **interfaz**, que establecerá la manera en que se comunican y cooperan los programas. Los errores en el uso de subprogramas se presentan fundamentalmente debido a una interfaz incorrecta entre el algoritmo llamante y el llamado. Para diseñar la interfaz hemos de tener en cuenta la información que necesita la función y qué datos de salida producirá.

Normas de uso de parámetros
El uso de parámetros es **opcional**
Los parámetros de la función pueden ser de **cualquier tipo**
El **número** de parámetros formales debe ser igual al número de parámetros reales
El **i-ésimo** parámetro formal se corresponde con el **i-ésimo** parámetro real
El tipo del **i-ésimo** parámetro formal se corresponde con el tipo del **i-ésimo** parámetro real
El nombre de un parámetro formal y su correspondiente **no tienen que ser iguales**

Paso de parámetros por valor

Generalmente, cuando pasamos un parámetro a una función, estos pueden modificar su valor en el interior de la misma sin que dichas alteraciones se vean reflejadas fuera de la función. Conocemos este mecanismo como **paso de parámetros por valor**, y es utilizado por defecto en PHP.

Cuando utilizamos parámetros por valor, se realiza una copia local del parámetro real sobre el parámetro formal, de manera que cualquier modificación sobre éste último no afecte al primero. Esta técnica **aísla** las variables y permite como parámetro real **constantes** y **expresiones**.

Paso de parámetros por referencia

Es muy sencillo utilizar el paso por valor a funciones, pero, ¿qué sucede con el paso por referencia? PHP utiliza para ello lo que se conoce como **referencias**. Otros lenguajes hacen uso del concepto de puntero, pero una referencia en realidad es lo que su propio nombre indica, un alías de una variable en la tabla de símbolos. Básicamente es como tener dos nombres para una misma variable. Observa el siguiente fragmento de código e indica cuál será el resultado por pantalla.

```
$a = 8 ;
$b = $a ;
```

```
echo "a es igual a $a y b igual a $b<br/>" ;
$a++ ;
echo "a es igual a $a y b igual a $b<br/>" ;
```

Efectivamente, pasa lo que esperábamos. El primer **echo** mostrará que **$a** y **$b** tienen el mismo valor. Sin embargo, al final **$a** valdrá 9 y **$b** seguirá valiendo 8. Utilizamos ahora el operador *ampersand* **&** para definir una referencia a **$a**.

```
$a = 8 ;
$b = &$a ;
echo "a es igual a $a y b igual a $b<br/>" ;
$a++ ;
echo "a es igual a $a y b igual a $b<br/>" ;
```

¿Qué mostrará el código anterior por pantalla? Pruébalo antes de continuar y comprobarás que, el primer **echo** seguirá mostrando el mismo resultado que antes, pero no ocurre así con el segundo. Ahora, tanto **$a** como **$b** tienen el mismo valor. Esto ha sucedido porque hemos definido **$b** como una referencia a **$a** de manera que, cuando modifiquemos el valor de esta última, se verá también alterado el valor de todas aquellas variables que la referencian.

Veamos gráficamente cómo funcionan las referencias. Supongamos definimos dos variables **$m** y **$n** y les asignamos respectivamente los valores 20 y 10, por lo que le mapa de memoria de PHP podría ser:

Índice	Valor	Variables
1	20	$m
2	10	$n
...	...	...

Si ahora hacemos **$n = &$m** el mapa cambiará de la siguiente manera:

Índice	Valor	Variables
1	20	$m, $n
...	...	...

PHP asocia ambas variables con el mismo valor, de manera que, si modificamos el valor de una, estamos modificando el de la otra. Si ahora utilizamos la función **unset** de PHP para destruir **$m**, ¿qué ocurriría con **$n**? ¿También se destruye o conserva su valor? La respuesta es sencilla: conserva su valor.

Índice	Valor	Variables
1	20	$n
...	...	...

El uso de **unset** sobre una variable no implica que también se destruyan aquellas referencias que se hayan definido sobre ésta. Obviamente, el uso de referencias va mucho más allá de lo que acabamos de explicar. Por ejemplo, cuando declaramos una variable con el modificador **global** en realidad estamos creando una referencia a una variable global. Así, hacer **global $var** será equivalente a **$var = &$GLOBALS["var"]**.

PHP permitirá por tanto definir parámetros por referencia de manera que, si modificamos dentro de la función el valor de aquellas variables definidas de esta manera, dicho valor se verá también modificado fuera de ésta.

```
function modificarCadena(&$cad) { $cad .= " mundo!" ; }
$txt = "¡Hola " ;
modificarCadena($txt) ;
echo $txt ;
```

Si ejecutamos el fragmento de código anterior observaremos que en pantalla se mostrará la cadena **¡Hola mundo!**. Esto sucede porque la variable **$cad** se ha definido en la interfaz de la función como una **referencia** al parámetro real utilizado en cualquier llamada a la misma. De esta manera, al modificar el valor de **$cad** estamos modificando también el valor de **$txt**.

Parámetros opcionales

PHP permite igualmente definir un valor por defecto para los parámetros de una función. El valor deberá ser una expresión constante de cualquiera de los siguientes tipos: **bool**, **int**, **float**, **string**, **array** o **NULL**.

```
function mostrarMensaje($msg = "¡Hola mundo!") { echo "$msg<br/>" ; }
```

Si llamamos a **mostrarMensaje** sin ningún parámetro, la función escribirá en pantalla el mensaje por defecto; en otro caso, se mostrará el valor que hayamos proporcionado a la función a través del parámetro **$msg**. . Podemos definir tantos parámetros con valor por defecto como deseemos, pero tendrán que aparecer siempre a la derecha de aquellos para los que no hemos definido un valor predeterminado. Los parámetros por referencia también admiten valores por defecto.

Parámetros con nombre

Supongamos ahora que definimos la función de la siguiente manera.

```
function foo($a, $b = false, $c = "Bienvenido/a") { ... }
```

Como los dos últimos parámetros son opcionales, nos debería permitir llamar a la función de cualquiera de las siguientes maneras.

```
foo(1, true, "Hola") ;
foo(1, true) ;
foo(1, "Hola") ;
```

Obviamente, el primer parámetro aparecerá siempre ya que es obligatorio. Sin embargo, la última llamada podría hacer que la función no haga lo que se desea, o que el compilador muestre en pantalla un error.

Fatal error: Uncaught TypeError: Argument 2 passed to foo() must be of the type string, bool given, called

Todo esto dependerá del modo en que estemos trabajando, cosa que veremos más adelante. Como se dijo anteriormente, existe una correlación entre los parámetros con el orden de sus argumentos y, en este caso, el compilador interpreta que “Hola” se corresponde con el segundo parámetro, cuando ésto realmente no es así. ¿Tiene solución el problema? Ahora sí. PHP 8 incorpora un mecanismo bastante sencillo que ya se utilizaba en otros lenguajes: **argumentos con nombre**.

```
foo(1, c: "a")
```

Como ves, basta con indicar el nombre del parámetro al que corresponde el valor, aumentando la flexibilidad del uso de parámetros en funciones y empleando únicamente los que son estrictamente necesarios.

Funciones variádicas

Aunque en especificaciones anteriores a la 5.6 podíamos definir funciones con un número variable de argumentos, las **funciones variádicas** aparecen para facilitarnos la tarea, creando además un código mucho más limpio y legible. Nos bastará con indicar explícitamente en la interfaz de la función, que el parámetro admite un número indeterminado de valores anteponiendo el operador **...** a su identificador.

```
function sumarValores(...$vector) { echo array_sum($vector) ; }
```

De esta manera, en la llamada a la función podremos indicar tantos valores como deseemos.

```
sumar(6, 4, 1, 3) ;
```

No obstante, este tipo de funciones presentan ciertas restricciones. En primer lugar, tan **sólo el último argumento** definido en la interfaz puede ser de tipo *variádico*. Además, este tipo de argumentos **no admiten valores por defecto**.

Modos coercitivo y estricto

En cuanto al método de validación de argumentos, las declaraciones de parámetros de tipo escalar (cadenas, enteros, decimales y booleanos, arrays y callables) pueden ser de dos tipos: **coercitivo** y **estricto**.

El modo **coercitivo** es el utilizado por defecto en PHP y nos permite forzar el tipo de datos de los parámetros, anteponiendo a éstos el nombre del tipo (**bool**, **int**, **float**, **string**, **array**, **callable**).

```
function sumarValores(int ...$vector) { return array_sum($vector) ; }
$var = sumarValores(2, '3', 4.1) ;
```

Si en el ejemplo anterior hacemos un volcado en pantalla de la variable **$var** comprobamos que el tipo de dicha variable es **int**. Sin embargo, podemos optar por no forzar el tipo de los elementos contenidos en la variable **$vector**. ¿Cuál será entonces el tipo del resultado devuelto por la función? Compruébalo y verás que es **float**. Esto es debido a que éste es el tipo más amplio de entre los de cada parámetro enviado a la función.

El modo **estricto**, por su parte, nos permite ser todo lo rigurosos que deseemos con el chequeo de tipos. Utilizamos para ello la directiva **declare** al comienzo del script PHP y, a partir de ese momento el intérprete comprobará los tipos de los parámetros y los valores devueltos por las funciones.

```
declare(strict_types = 1) ;
```

Tipo devuelto por una función

PHP permite definir el tipo para el valor que devuelve una función, forzando de esta manera una tipificación fuerte para nuestro lenguaje. En este caso, el valor devuelto deberá ser del tipo correcto o en otro caso, el intérprete lanzará una excepción. Definimos el valor devuelto por la función indicándose en la interfaz, a continuación de la lista de parámetros.

```
function sumarValores(... $vector): int { return array_sum($vector) ; }
```

Tipos nullable y void

PHP 7 introdujo el **tipo nullable**, permitiendo que tanto parámetros como valores de retorno puedan tener un valor nulo. Indicamos esta posibilidad marcando el tipo con el operador **?**.

```
function foo(int $a, ?bool $b, ?string $c = "función foo"): ?float { ... }
```

El **tipo void** es otra de las novedades que se introdujeron en la versión PHP 7. Aquellas funciones que definen el tipo de retorno como **void** utilizarán la sentencia return vacía o, simplemente se omite su uso. **IMPORTANTE**: **null** no será un valor de retorno válido para una función definida como **void**.

Union Types

A partir de la versión 8 de PHP se incorpora una nueva característica conocida como **unión de tipos** (**union types**) que permite definir parámetros de diferentes tipos al mismo tiempo. Existe una excepción: **void** no se considerará un tipo válido para una **union type**.

```
function foo(int|bool $a) { ... }
```

La función anterior acepta un único parámetro que puede ser de tipo **entero** o **booleano**. En el caso especial de tipos nullable, también podríamos definir los parámetros utilizando esta notación, aunque siempre es preferible utilizar el operador **?** como abreviatura. De esta manera, las siguientes expresiones son equivalentes.

```
function foo(int|null $a) { ... }      // null siempre será el último valor
function foo(?int $a)     { ... }
```

Es importante señalar que, aunque podemos utilizar cualquiera de las dos notaciones anteriores, no debemos mezclarlas.

```
function foo(?int|string $a) { ... }   // declaración ambigua no permitida
```

También podremos utilizar **union types** para expresar el valor devuelto por una función.

```
function foo(...) : int|bool { ... }
```

Union Types: el tipo especial false

Generalmente el valor **false** se utiliza para expresar un resultado negativo. Supongamos que tenemos una función que busca en la base de datos un usuario por su **id** y lo devuelve; en caso de no encontrarlo, la función devolverá **false**. En estos casos, se nos permite definir la función como sigue:

```php
function buscar_usuario(int $id): Usuario|false { ... }
```

No obstante, debemos tener en cuenta los siguientes aspectos:

1. Siempre se utilizará **false** como parte de un **union type** y nunca solo.
2. El pseudotipo **false** se admite allí donde podamos utilizar tipos de datos: propiedades de clases, argumentos de funciones y tipo devuelto por una función.
3. No podrá utilizarse **false** como parte de un **union type** donde también utilicemos **bool**.

Obviamente, tendremos que hacer que la función retorne un valor de tipo cadena y otro de tipo **false**. Curiosamente, si probamos a devolver **true** comprobaremos que realmente estamos devolviendo una cadena con el valor **1**.