

Universidad Nacional de Río Cuarto
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Departamento de computación
Análisis Comparativo de Lenguajes (1956)
Año 2009



Estudio del lenguaje de programación
PHP – Hypertext Preprocessor

Alumno: Fessia Matias

Índice

1	Conociendo PHP	
1.1.	Historia.....	1
1.2.	Introducción	1
1.3.	Identificación de bloques.....	2
1.4.	Separación de instrucciones	2
1.5.	Comentarios.....	2
2	Tipos	
2.1.	Definición	3
2.2.	Clasificación de tipos	3
2.2.1.	Bolean	3
2.2.2.	Integer	3
2.2.3.	Float.....	4
2.2.4.	String	4
2.2.5.	Array.....	4
2.2.6.	Object	5
2.3.	Sistema de tipos	6
2.3.1.	Tipado dinámico	6
2.3.2.	Equivalencia de tipos	6
2.3.3.	Principio de completitud de tipos	6
2.3.4.	Type Safe (tipado seguro).....	7
2.3.5.	Transformaciones de tipo (casting).....	7
3	Storage (sistema de almacenamiento)	
3.1.	Variables	8
3.2.	Variables Locales.....	8
3.3.	Variables Globales	8
3.4.	Variables externas	9
3.5.	Variables variables.....	10
3.6.	Constantes.....	10
4	Bindings (ligaduras)	
4.1.	Ligadura dinámica o estática.....	11
4.2.	Definiciones	11
4.2.1.	Declaraciones de tipo.....	11
4.2.2.	Declaraciones de variable	12
4.2.3.	Declaraciones Secuenciales	12
4.2.4.	Declaraciones Recursivas.....	12
4.3.	Scope.....	12
5	Operadores	
5.1.	Operadores de Aritmética.....	13
5.2.	Operadores de Asignación	13
5.3.	Operadores Bit a Bit	13
5.4.	Operadores de Relación.....	14
5.5.	Operadores de Control de Errores.....	14
5.6.	Operadores de ejecución	14
5.7.	Operadores de incremento/decremento	14
5.8.	Operadores de lógica	15
5.9.	Operadores de cadena.....	15
5.10.	Operadores de matrices	15
5.11.	Operadores de tipo.....	16
6	Estructuras de Control	
6.1.	Bifucaciones.....	17
6.1.1.	Sentencia If	17
6.1.2.	Sentencia else.....	17
6.1.3.	Sentencia elseif	18
6.1.4.	Sentencia switch	18
6.2.	Bucles	18

6.2.1.	Sentencia while	18
6.2.2.	Sentencia do..while	19
6.2.3.	Sentencia for	19
6.2.4.	Sentencia foreach	20
6.3.	Sentencias break y continue.....	20
6.3.1.	Sentencia break	20
6.3.2.	Sentencia continue	21
6.4.	Sentencia return.....	21
6.5.	Bloque try and catch.....	21
6.6.	Sentencias include y require.....	21
6.6.1.	Sentencia require	21
6.6.2.	Sentencia incluye	21
7	Funciones	
7.1.	Funciones Condicionales	22
7.2.	Funciones dentro de funciones.....	22
7.3.	Parámetros.....	22
7.3.1.	Pasar parámetros por referencia	23
7.3.2.	Parámetros por defecto.....	23
7.4.	Retorno de valores	23
8	Referencias	
8.1.	Destruir una referencia	24
8.2.	Referencias colgadas	24
9	Programación Orientada a Objetos	
9.1.	Objetos.....	25
9.2.	Clases	25
9.2.1.	Atributos	25
9.2.2.	Métodos.....	25
9.3.	Constructores y destructores.....	26
9.3.1.	Constructores	26
9.3.2.	Destructores	27
9.4.	Visibilidad (scope)	28
9.5.	Clases y métodos abstractos.....	28
9.6.	Interfaces	28
9.7.	Herencia.....	30
9.8.	Constantes de clase	31
9.9.	Atributos o métodos abstractos	31
9.10.	Final	32
9.11.	Clonado de Objetos.....	33
9.12.	Type Hinting	33
9.13.	Polimorfismo	33
9.13.1.	Polimorfismo por sobrecarga	33
9.13.2.	Polimorfismo por inclusión	34
9.13.3.	Polimorfismo paramétrico	35
10	Excepciones.....	36
11	Concurrencia	
11.1.	Como funcionan los semáforos	37
11.2.	Gestión de semáforos	37
11.3.	Ejemplo	38
12	Extendiendo PHP	
12.1.	Configuración e instalación de extensiones.....	39
12.2.	Creación de una nueva biblioteca.....	39
12.3.	PHP Java Bridge	39
12.4.	COM.....	40
13	Conclusión	41
14	Bibliografía	42

1 Conociendo PHP

1.1 Historia

PHP es el heredero de un producto anterior, llamado PHP/FI, el cual, fue creado por Rasmus Lerdorf en 1995, inicialmente como un simple conjunto de scripts de Perl para controlar los accesos a su trabajo online. Llamó a ese conjunto de scripts 'Personal Home Page Tools'. Según se requería más funcionalidad, Rasmus fue escribiendo una implementación en C mucho mayor, que era capaz de comunicarse con bases de datos, y permitía a los usuarios desarrollar sencillas aplicaciones Web dinámicas. Rasmus eligió liberar el código fuente de PHP/FI para que cualquiera pudiese utilizarlo, así como arreglar errores y mejorar el código.

PHP/FI, que se mantuvo para páginas personales y como intérprete de formularios, incluía algunas de las funcionalidades básicas de PHP tal y como lo conocemos hoy. Tenía variables como las de Perl, interpretación automática de variables de formulario y sintaxis embebida HTML. La sintaxis por sí misma era similar a la de Perl, aunque mucho más limitada, simple y algo inconsistente.

Por 1997, PHP/FI 2.0, la segunda escritura de la implementación en C, tuvo un seguimiento estimado de varios miles de usuarios en todo el mundo, con aproximadamente 50.000 dominios informando que lo tenían instalado, sumando alrededor del 1% de los dominios de Internet. Mientras había mucha gente contribuyendo con bits de código a este proyecto, era todavía en su mayor parte el proyecto de una sola persona.

PHP/FI 2.0 no se liberó oficialmente hasta Noviembre de 1997, después de gastar la mayoría de su vida en desarrollos beta. Fue sucedido en breve tiempo por las primeras versiones alfa de PHP 3.0.

PHP 3.0 era la primera versión que se parecía fielmente al PHP tal y como lo conocemos hoy en día. Fue creado por Andi Gutmans y Zeev Zuraski en 1997 reescribiéndolo completamente, después de que encontraran que PHP/FI 2.0 tenía pocas posibilidades para desarrollar una aplicación comercial que estaban desarrollando para un proyecto universitario. En un esfuerzo para cooperar y empezar a construir sobre la base de usuarios de PHP/FI existente, Andi, Rasmus y Zeev decidieron cooperar y anunciar PHP 3.0 como el sucesor oficial de PHP/FI 2.0, interrumpiéndose en su mayor parte el desarrollo de PHP/FI 2.0.

Luego siguieron las versiones PHP 4 y la última de hoy en día PHP 5.

1.2 Introducción

PHP (Hypertext Preprocessor) es un lenguaje de "código abierto" interpretado (ver nota) de propósito general y está diseñado especialmente para desarrollo web ya que puede ser embebido (incluido) dentro de código HTML.

Es usado principalmente en interpretación del lado del servidor (server-side scripting) pero actualmente es posible crear aplicaciones con una interfaz gráfica para el usuario usando las bibliotecas Qt o GTK. También puede ser usado desde la línea de órdenes (Command Line Interface), de la misma manera como Perl o Python pueden hacerlo.

Un ejemplo sencillo para ver mejor que es PHP.

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <?php
      echo "Hello World";
    ?>
  </body>
</html>
```

El beneficio es que en vez de escribir un programa con muchos comandos para crear una salida en HTML, escribimos el código HTML con cierto código PHP embebido (incluido) en el mismo, que producirá cierta salida (en nuestro ejemplo, producirá un texto). El código PHP se incluye entre etiquetas especiales de comienzo y final que nos permitirán entrar y salir del modo PHP.

Lo que distingue a PHP de la tecnología Javascript, la cual se ejecuta en la máquina cliente, es que el código PHP es ejecutado en el servidor. Si tuviésemos un script similar al de nuestro ejemplo en nuestro servidor, el cliente solamente recibiría el resultado de su ejecución en el servidor, sin ninguna posibilidad de determinar qué código ha producido el resultado recibido.

Nota: Se conoce como lenguaje interpretado a un lenguaje de programación que fue diseñado para ser ejecutado por medio de un intérprete, en contraste con los lenguajes compilados. También se les conoce como lenguajes de script.

1.3 Identificación de bloques

Para interpretar un archivo, PHP simplemente interpreta el texto del archivo hasta que encuentra uno de los caracteres especiales que delimitan el inicio de código PHP. Luego el intérprete ejecuta todo el código hasta que encuentra una etiqueta de fin de código. Este mecanismo permite embeber código PHP dentro de HTML: todo lo que está fuera de las etiquetas PHP se deja tal como está, mientras que el resto se interpreta como código.

Las diferentes formas de escribir bloques de código PHP embebido en HTML, son:

- `<?php. . .?>`
- `<? ... ?>`
- `<?= expression ?>`
- `<script language="php">. . .</script>`

El primer método, es el más conveniente, ya que permite el uso de PHP en código XML como XHTML. El resto de etiquetas no siempre están disponibles ya que estas son configuradas desde el archivo "php.ini".

1.4 Separación de instrucciones

Las separaciones de instrucciones se hacen terminando cada declaración con un punto y coma. La etiqueta de fin de bloque (`?>`) implica el fin de la declaración, por lo tanto los dos siguientes scripts son equivalentes:

```
<?php
    echo "hello world";
?>

<?php echo "hello world" ?>
```

1.5 Comentarios

Los comentarios se usan para dejar ciertas aclaraciones ya que resulta fácil leer un código cuando este está correctamente comentado.

Para dejar comentarios (que no se procesan como código) en PHP hay tres diferentes maneras; con la doble barra, con la barra-asterisco y con el carácter numeral.

```
<?php
    // Tipo de comentario de una línea
    /* Bloque de comentario de mas de una línea*/
    # Tipo de comentario de una línea
?>
```

2 Tipos

2.1 Definición

Para entender mejor el concepto de tipo primero tenemos que ver que es un valor.

Un valor es cualquier cosa que puede ser evaluada, almacenada, incorporada en una estructura de datos, pasada como un argumento de un procedimiento o una función, retornada como resultado de una función, etc..

Mientras que un tipo es un conjunto de valores bajo un mismo comportamiento y operaciones asociadas a estos valores.

2.2 Clasificación de tipos

Podemos clasificar los tipos que soporta php en dos grandes grupos:

- Tipos primitivos: es aquel cuyo valor es atómico y por lo tanto no puede descomponerse en valores simples.
 - boolean
 - integer
 - float
- Tipos compuestos: es aquel que está estructuralmente compuesto por valores simples.
 - string
 - array
 - object

2.2.1 Boolean (booleano)

Un booleano expresa un valor de verdad. Puede ser verdadero o falso, y se usa para realizar comprobaciones lógicas. La representación de almacenamiento para un booleano es un solo bit, 0 = false y 1 = true.

Sintaxis

Para especificar un literal booleano, se utiliza una de las palabras clave "true" o "false". Ambas no son sensibles a mayúsculas y minúsculas.

```
<?php
    $bool = True; // asigna el valor TRUE a $bool
    $bool = False; // asigna el valor FALSE a $bool
?>
```

2.2.2 Integer (entero)

Los números enteros son una generalización del conjunto de números naturales que incluye números negativos. Así los números enteros están formados por un conjunto de enteros positivos que podemos interpretar como los números naturales convencionales, el cero, y un conjunto de enteros negativos que son los opuestos de los naturales.

Sintaxis

Los enteros pueden ser especificados en notación decimal (base-10), hexadecimal (base-16) u octal (base-8), opcionalmente precedidos por un signo (- o +).

```
<?php
    $a = 1234; // número decimal
    $a = -123; // un número negativo
    $a = 112; // número octal (equivalente al 74 decimal)
    $a = 03E7; // número hexadecimal (equivalente al 999 decimal)
?>
```

El tamaño de un entero es dependiente de la plataforma. Dicho tamaño puede determinarse a partir de la función `php_int_size`, o el valor máximo de `php_int_max`.

2.2.3 Float (numero de punto flotante)

Los números de punto flotante (también conocidos como "flotantes", "dobles" o "números reales") son aquellos que poseen una expansión decimal.

La representación de almacenamiento de un tipo de número float se basan en una mantisa (los dígitos significativos del número) y un exponente.

El tamaño de un flotante depende de la plataforma, aunque un valor común consiste en un máximo de ~1.8e308 con una precisión de aproximadamente 14 dígitos decimales

Sintaxis

```
<?php
    $a = 0.99;
?>
```

2.2.4 String (cadena)

Un string es una secuencia de caracteres. En PHP, un carácter es lo mismo que un byte, es decir, hay exactamente 256 tipos de caracteres diferentes para cada uno. Esto implica también que PHP no tiene soporte nativo de Unicode.

Sintaxis

Para denotar cadenas en PHP lo hacemos a través de las comillas simples o dobles. La diferencia entre las dos es que con las comillas simples solo se puede escapar con la barra invertida, mientras que con las comillas dobles se permiten mas secuencias de escape para caracteres especiales.

```
$cadena = "Hello World";
```

Acceso a cadenas y modificación por carácter

Los caracteres al interior de una cadena pueden ser consultados y modificados dando su ubicación comenzando desde cero hasta la longitud de la cadena.

```
echo $cadena[4]; // Imprime "o"
```

2.2.5 Arreglo (array)

Una matriz en PHP es en realidad un mapa ordenado. Un mapa es un tipo de datos que asocia valores con claves. Este tipo es optimizado en varias formas, de modo que puede usarlo como una matriz real, o una lista (vector), tabla asociativa (caso particular de implementación de un mapa), diccionario, colección, pila, cola y probablemente más. Ya que puede tener otra matriz PHP como valor, es realmente fácil simular árboles.

Sintaxis

Para declarar y usar una variable arreglo en un programa en PHP, lo hacemos de las siguientes maneras:

```
$arreglo = array(3,"tres", false); // inicializándolo
$arreglo = array(); //sin inicializar
```

Recordar en PHP que la primera posición o renglón de una lista es la posición 0 (cero).

Una buena forma de definir arreglos en PHP es tomar un cierto número de parejas (clave => valor) separadas con coma.

```
array( clave => valor
      , ...
      )
// clave puede ser un integer o string
// valor puede ser cualquier valor

<?php
    $matriz = array("num" => "22", 5 => true);
    echo $matriz["num"]; // sale 22
    echo $matriz[5];    // sale 1
?>
```

Si no especifica una clave para un valor dado, entonces es usado el máximo de los índices enteros, y la nueva clave será ese valor máximo + 1. Si especifica una clave que ya tiene un valor asignado, ése valor será sobrescrito.

```
<?php
// Esta matriz es la misma que ...
array(5 => 43, 32, 56, "b" => 12);

// ...esta matriz
array(5 => 43, 6 => 32, 7 => 56, "b" => 12);
?>
```

Tipos enumerados

Es una lista ordenada de valores distintos. Por ejemplo supongamos el sexo de una persona:

```
enum = {Femenino, Masculino}
```

Cada valor de la serie de enumeración esta representado por un número entero.

Por ejemplo 1 = Femenino y 2 = Masculino.

PHP no tiene tipo de datos enumeración pero se pueden simular con los con arreglos. Por ejemplo:

Supongamos que quisiéramos declarar el tipo enumerado que representa a los meses del año que poseen 31 días:

```
TypeNumeric = (01,03,05,07,08,10,12)
```

Donde enero = 01, marzo =02.....diciembre = 12.

Con un array lo podemos simular fácilmente:

```
$TypeNumeric = array(01,03,05,07,08,10,12);
```

Y si lo emprolijamos un poco:

```
$meses = array("01"=>"Enero", "02"=>"Febrero", etc ....);
```

2.2.6 Objetos (object)

Ver capitulo 9 - Programación Orientada a Objetos

2.3 Sistema de Tipos

2.3.1 Tipado dinámico

PHP es un lenguaje de tipado dinámico, eso quiere decir que solo los valores tienen un tipo fijo. Una variable o parámetro no tiene un tipo fijo designado, puede tomar valores de diferentes tipos en diferentes tiempos. Esto implica que los operandos deben tener un chequeo de tipos inmediato antes de ejecutar una operación en tiempo de ejecución.

La ventaja es la flexibilidad y la implementación del polimorfismo pero tiene su costo; como el chequeo de tipos es en tiempo de ejecución, se hace más lento la ejecución del programa.

2.3.2 Equivalencia de tipos.

La equivalencia de tipos se usa para chequear si un valor X es de algún tipo de datos determinado. Por ejemplo consideremos una operación que espera como argumento un valor de tipo T y nosotros le pasamos algo de tipo T' , luego debemos chequear que el tipo T sea equivalente a T' .

PHP soporta equivalencia de tipo nominal, donde dos tipos son equivalentes si sus nombres coinciden.

En el siguiente ejemplo crearemos dos clases con misma estructura y veremos si la equivalencia es por nombre o por estructura.

```
<?php
class clase1 {
    public $valor;
}

class clase2 {
    public $valor;
}

$c1 = new clase1();
$c2 = new clase2();

$c1->valor = 3;
$c2->valor = 3;

// con el operador igual
if ($c1 == $c2) {
    echo "son equivalentes";
} else {
    echo "no son equivalentes";
}

// con el operador identico
if ($c1 === $c2) {
    echo "son equivalentes";
} else {
    echo "no son equivalentes";
}
?>
```

Nota: Ambas consultas arrojan que no son equivalentes.

2.3.3 Principio de completitud de tipos

Este principio define que ninguna operación debería restringir, arbitrariamente los tipos sobre los que opera.

En un lenguaje de programación nos encontramos, por lo general, con dos tipos de valores:

- Valores de primera clase: estos pueden evaluarse, asignarse, ser componentes de otros valores, pasarse como parámetros de una función o procedimiento, ser retornados por funciones, etc.
- Valores de segunda clase: no pueden hacer alguna de las operaciones anteriores.

Para que un lenguaje cumpla con este principio, debe suceder, que todos los componentes del lenguaje sean de primera clase.

En PHP este principio no se respeta ya que los objetos o arreglos no son en su totalidad valores de primera clase ya que por ejemplo a un arreglo no se lo puede sumar con un entero.

```
<?php
    $int = 1;
    $numbers = array(3,4,3);
    echo ($numbers + $int); // PHP Fatal error:  Unsupported operand types
?>
```

2.3.4 Type Safe (seguridad de tipos)

La seguridad de tipos afirma que ninguna operación, luego de compilar, va a ejecutarse con tipos incorrectos.

PHP no es tipado seguro ya que el chequeo de tipos se realiza en tiempo de ejecución, por lo que no asegura que una operación este operando sobre tipos correctos.

El siguiente ejemplo arroja un warning de división por cero:

```
<?php
    $result = 5/round(0,2); // PHP Warning:  Division by zero
    print $result;
?>
```

Nota: también podríamos haber colocado un cero en vez de redondear el 0,2 ya que al ser un lenguaje que no se compila el chequeo se realiza en tiempo real y nos arrojará el warning de todas formas.

2.3.5 Transformaciones de tipo (casting)

En varias ocasiones es necesario transformar el tipo de una variable a otro, por ejemplo si queremos asignarle un entero a un string, es necesario transformar ese entero a un string para luego poder asignarlo.

Para lograr esto en PHP basta con poner el nombre del tipo entre paréntesis adelante de la variable que se le va a aplicar el casting.

Ejemplo

```
<?php
    $int = 3;
    $str = "hello";
    $str = (string) $int; // acá hago el casting
    echo $str; // imprime 3
?>
```

En PHP es posible hacer transformaciones de los siguientes tipos:

- (int), (integer)
- (bool), (boolean)
- (float), (double), (real)
- (string)
- (array)
- (object)

Ahora si lo que se quiere hacer es transformar el tipo de una variable se usará la función `settype()` que le asigna un tipo a una variable. Veamos un ejemplo

```
<?php
    $int = 3;
    settype($int, "string"); // ahora la variable int es un string
?>
```

Esta función se puede usar con los siguientes tipos:

- boolean
- integer
- float
- string
- array
- object

3 Storage (Sistema de almacenamiento)

3.1 Variables

Una variable es un objeto que contiene un valor que puede cambiar a lo largo del programa. Son actualizadas mediante asignaciones y tienen un tiempo de vida corta.

Se representa comúnmente como almacenamiento en la memoria de la computadora y esta representado por un patrón de bits.

Sintaxis

En PHP las variables se representan como un signo de dólar seguido por el nombre de la variable y no se puede especificar el tipo de dato que esta contendrá.

El nombre de la variable es sensible a minúsculas y mayúsculas.

```
<?php
    $var = "Esta variable contiene una cadena";
?>
```

3.2 Variables Locales

Es aquella declarada dentro de un bloque para utilizarla dentro de ese bloque.

Veamos un ejemplo:

```
<?php
function CountToFive(){
    // se define una variable local $i
    for ($i = 1; $i < 5 ; $i++){
        echo $i;
    }
}

CountToFive(); // imprime "12345"
echo $i; // no imprime nada ya que la variable $i no esta definida en
este bloque
?>
```

3.3 Variables Globales

Es aquella que se declara dentro de un bloque más externo del programa.

En PHP cuando declaramos una variable global se cargara automáticamente en el arreglo \$GLOBALS, y por este mismo podemos acceder a las variables globales desde cualquier parte del programa.

Veamos el siguiente script de ejemplo:

```
<?php
$i;

function inc(){
    $GLOBALS["i"]++; // ver nota 1
    $i = 10; // ver nota 2
}

$i = 1;
inc();
echo $i; // imprime "2"
?>
```

Nota 1: en esta línea de nuestro script accedemos al arreglo \$GLOBALS con el nombre de la variable como índice.

Nota 2: esta línea no modifica la variable global \$i sino que lo que se esta haciendo es declarar una variable local con el mismo nombre y asignarle un valor.

Ahora, si se quisiera crear una variable global dentro de una subrutina se usa la sentencia *global*. Veamos un uso:

```

<?php
    function inc(){
        global $i; // por defecto un entero se crea con el valor "0"
        $i++;
    }

    inc();
    echo $i; // imprime "1"
?>

```

No sería lo correcto declarar variables globales dentro de una subrutina, pero un buen uso de la sentencia global es para nombrar las variables globales ya creadas una sola vez y no tener que recurrir al arreglo \$GLOBALS cada vez que quisiéramos usarlas. A continuación un script de ejemplo con un buen uso de la sentencia global:

```

<?php
    $op1 = 1;
    $op2 = 2;
    $result = 0;

    function sum(){
        global $op1, $op2, $result;
        $result = $op1 + $op2;
    }

    sum();
    echo $result; // imprime "3"
?>

```

En este ejemplo veremos como perdemos una variable local de una función cuando luego la declaramos como global dentro de la función.

```

<?php
    $op1 = 1;
    $op2 = 2;
    $result = 0;

    function sum(){
        $op1 = 10;
        echo $op1; // imprime 10
        global $op1, $op2, $result;
        echo $op1; // imprime 1 por lo que perdimos la variable local
        $result = $op1 + $op2;
    }

    sum();
    echo $result; // imprime "3"
?>

```

3.4 Variables externas

En el ámbito de la programación web, hay mucha información externa que pueda ser necesitada por el programador, por ejemplo cuando se envía un formulario web, o datos del equipo en donde se este ejecutando el programa, etc.

Para esto PHP define las siguientes matrices en donde cada una contiene un conjunto de variables que proporcionan alguna información externas al lenguaje.

- `$_SERVER`: Variables definidas por el servidor web ó directamente relacionadas con el entorno en donde el script se esta ejecutando.
- `$_GET`: Variables proporcionadas al script por medio de HTTP GET.
- `$_POST`: Variables proporcionadas al script por medio de HTTP POST.
- `$_COOKIE`: Variables proporcionadas al script por medio de HTTP cookies.
- `$_FILES`: Variables proporcionadas al script por medio de la subida de ficheros vía HTTP.
- `$_ENV`: Variables proporcionadas al script por medio del entorno.
- `$_REQUEST`: Variables proporcionadas al script por medio de cualquier mecanismo de entrada del usuario y por lo tanto no se puede confiar en ellas.
- `$_SESSION`: Variables registradas en la sesión del script.

3.5 Variables variables

Con PHP es posible declarar variables con nombres dinámicos, es decir, que el nombre de una variable sea a la vez, variable.

Sintaxis

Para que una variable tome el nombre de otra variable, utilizaremos dos signos de pesos en vez de uno.

Veamos un ejemplo:

```
<?php
    $var = "hello";
    $$var = "world";
    echo $var.' '.$hello; // imprime "hello world"
?>
```

3.6 Constantes

Una constante es una variable que esta ligada de forma permanente a un valor durante su tiempo de vida.

En PHP, una constante es sensible a mayúsculas por defecto. Por convención, los identificadores de constantes suelen declararse en mayúsculas.

El alcance de una constante es global, es decir, es posible acceder a ellas sin preocuparse por el ámbito de alcance.

Sintaxis

Se puede definir una constante usando la función *define()*. Una vez definida, no puede ser modificada ni eliminada.

Solo se puede definir como constantes valores de tipo; booleano, entero, flotante y cadena.

```
<?php
    define("HW", "Hello World.");
    echo HW; // imprime "Hello world"
    echo Constant("HW");
    // imprime "Hello World"
?>
```

Para obtener el valor de una constante solo es necesario especificar su nombre. A diferencia de las variables, no se tiene que especificar el prefijo \$. También se puede utilizar la función *constant()*, para obtener el valor de una constante.

Para obtener una lista de todas las constantes declaradas en nuestro programa, existe la función *get_defined_constants()*.

4 Bindings (ligaduras)

Significa ligar un identificador a una entidad en un lugar o ambiente.

4.1 Ligadura dinámica o estática

Para determinar el tipo de binding que PHP posee primero veremos un ejemplo del libro.

Nota: en el libro el ejemplo esta dado con constantes, pero en PHP no es posible reproducirlo ya que no permite definir constantes locales.

```
<?php
    $i = 9;

    function scaled($var){
        return $var * $i; // no encuentra la variable i por lo tanto la
                          // crea local e inicializada en 0
    }

    function call_scaled($var){
        $i = 2;
        return scaled($var);
    }

    echo call_scaled(1);
?>
```

Bueno, ahora nos vamos a concentrar en la función *scaled()*, mas precisamente en la variable *\$i* que no esta declarada como local en la función o pasada como argumento de la función. Entonces, ¿como resuelve esta variable PHP? En la teoría esto se explica con el tipo de ligadura:

- Con binding estático, resuelve la variable *\$i* en el bloque que la contiene, es decir, en este caso, el valor de *\$i* en la función *scaled()* seria nueve (9).
- Con binding dinámico, resuelve la variable *\$i* en el bloque donde fue invocada la función, es decir, en la función *call_scaled()*, por lo que el valor de *\$i* debería ser dos (2).

Curiosamente, PHP en este experimento, resuelve la variable de otra manera, la considera como que no esta definida y la declara como una variable local e inicializada en cero (0).

Por lo que no me quedan argumentos para definir el tipo de ligadura que soporta PHP, a no ser que me base en la definición del libro que afirma que el binding estático se hace en tiempo de compilación, como PHP no se compila, se interpreta y además el chequeo de asignaciones, tipos, etc, se hacen en tiempo de ejecución, podría decirse que PHP tiene binding dinámico pero hasta no encontrar un caso que me lo demuestre, sigue siendo un pendiente de este manual.

4.2 Definiciones

Definimos una definición como una simple declaración que su único efecto es producir bindings. Esto permite que a una entidad (constante, tipo o función) sea definida una sola vez y después utilizarla donde sea requerida mediante el uso del identificador que la denota.

4.2.1 Declaraciones de tipos

Una definición de tipo sirve únicamente para ligar un identificador a un tipo existente. Otra alternativa de declaraciones de tipos es cuando se crea un tipo nuevo y distinto, una declaración de tipo nueva.

En PHP no podemos hacer declaraciones de nuevos tipos, pero podemos hacer una declaración que ligue ese identificador a un tipo cuando se le asigna un valor de un tipo dado.

```
<?php
    $count = 0; // count es ligado al tipo int
    $count = "hello"; // ahora count es ligado al tipo string
?>
```

4.2.2 Declaraciones de variables

Una definición de variable sirve únicamente para ligar un identificador a una variable.

```
<?php
    $count = 0;
?>
```

Nota: en este script se crea una variable nueva y la liga el identificador *count* a ella. Además la inicializa en cero.

4.2.3 Declaraciones Secuenciales

Una declaración secuencial permite que si se realiza una declaración D1 y dentro de ella una declaración D2, los bindings producidos en D1 se puedan usar en D2.

PHP no soporta este tipo de declaración y lo vemos con el siguiente ejemplo en donde debería retornar el valor 6 si soportase este tipo de declaraciones.

```
<?php
    function inc(){
        $i++;
        return $i;
    }

    function call_inc(){
        $i = 5;
        return inc();
    }

    echo call_inc(); // imprime 1
?>
```

Nota: cuando lo ejecutamos nos devuelve un warning "Undefined variable: i" en la línea 3 (\$i++)

4.2.4 Declaraciones Recursivas

Una declaración recursiva es aquella que utiliza varios bindings para producirse ella misma. Es posible en PHP realizar funciones recursivas.

```
<?php
    function factorial($n){
        if ($n == 0){
            return 1;
        }else {
            return $n * factorial($n - 1);
        }
    }

    echo factorial(5); // imprime "120"
?>
```

4.3 Scope

Las distintas formas de hacer bindings que soporta PHP, difieren en el alcance de sus ligaduras:

- En una declaración de variable la ligadura se extiende desde el final de la declaración hasta el final del bloque que la incluye.
- Con una declaración recursiva el alcance de una declaración se incluye en ella misma.

5 Operadores

Las variables en un lenguaje pueden ser creadas, modificadas y comparadas con otras por medio de los operadores.

Estos se clasifican según su funcionalidad y por los tipos en cuales opera.

5.1 Operadores de Aritmética

Todo programa necesita hacer infinidad de operaciones de cálculo del tipo matemático, que se realizan con las siguientes operaciones.

Operador	Ejemplo	Resultado
Negación	-\$x	el opuesto de x
Adición	\$x + \$y	suma de x e y
Substracción	\$x - \$y	diferencia entre x e y
Multiplicación	\$x * \$y	producto entre x e y
División	\$x / \$y	división entre de x e y
Modulo	\$x % \$y	resto de x dividido por y

Nota: El operador de división ("/") devuelve un valor flotante a no ser que los dos operandos sean enteros o los números sean divisibles sin residuos, en dicho caso retorna un valor entero

5.2 Operadores de Asignación

Los operadores de asignación permiten asignar un valor a una variable.

Sintaxis

El operador de asignación esta dado por el símbolo "="

```
$x = 3 // el valor de la variable $x es igual a 3
```

En conjunto con el operador básico de asignación, existen "operadores combinados" para todos los operadores de aritmética binaria, unión de matrices y de cadenas, que le permiten usar un valor en una expresión y luego definir su valor como el resultado de esa expresión.

Por ejemplo:

```
<?php
    $a = 2;
    $a += 3; // toma lo que tenia antes $a y le adiciona el entero 3
    echo $a; // imprime 5
?>
```

5.3 Operadores Bit a Bit

Los operadores bit a bit le permiten activar o desactivar bits individuales de un entero. Si los parámetros tanto a la izquierda y a la derecha son cadenas, el operador bit a bit trabajará sobre los valores ASCII de los caracteres.

Operador	Ejemplo	Resultado
Y	\$a & \$b	Los bits que están activos tanto en \$a como en \$b son activados
O	\$a \$b	Los bits que están activos ya sea en \$a o en \$b son activados
O exclusivo (XOR)	\$a ^ \$b	Los bits que estén activos en \$a o \$b, pero no en ambos, son activados
No	~ \$a	Los bits que estén activos en \$a son desactivados, y vice-versa
Desplazamiento a izquierda	\$a << \$b	Desplaza los bits de \$a, \$b pasos a la izquierda (cada paso quiere decir "multiplicar por dos")
Desplazamiento a derecha	\$a >> \$b	Desplaza los bits de \$a, \$b pasos a la derecha (cada paso quiere decir "dividir por dos")

5.4 Operadores de relación

Estos operadores permiten comparar dos valores. El resultado de estos operadores es siempre un valor boolean (true o false) según se cumpla o no la relación considerada.

Operador	Ejemplo	Resultado
Igual	<code>\$x == \$y</code>	verdadero si x es igual a y
Idéntico	<code>\$x === \$y</code>	verdadero si x es igual a y , y si son del mismo tipo
Diferente	<code>\$x != \$y</code> o <code>\$x <> \$y</code>	verdadero si x no es igual a y
no idéntico	<code>\$x !== \$y</code>	verdadero si x no es igual a y , o si no son del mismo tipo
Menor	<code>\$x < \$y</code>	verdadero si x es estrictamente menor que y
menor o igual	<code>\$x <= \$y</code>	verdadero si x es menor o igual que y
Mayor	<code>\$x > \$y</code>	verdadero si x es estrictamente mayor que y
mayor o igual	<code>\$x >= \$y</code>	verdadero si x es mayor o igual que y

5.5 Operadores de Control de Errores

El símbolo arroba (@) es el elegido para representar al operador de control de error. Cuando este es colocado al comienzo de una expresión en PHP, cualquier mensaje de error que pudiera generarse a causa de esa expresión será ignorado.

Si la característica `track_errors` está habilitada, cualquier mensaje de error generado por la expresión será almacenado en la variable `$php_errormsg`. La variable será sobrescrita en cada instancia de error.

```
<?php
    $mi_archivo = @file($path); // si el archivo no existe no arrojara error
?>
```

5.6 Operadores de ejecución

Es posible escribir líneas de comandos en la consola del sistema operativo que nos encontremos.

Para esto usaremos las comillas invertidas (`).

```
<?php
    $salida = `ls -al`;
    echo $salida;
?>
```

Nota: PHP intentará ejecutar el contenido entre las comillas como si se tratara de un comando del intérprete de comandos.

5.7 Operadores de Incremento/Decremento

PHP ofrece soporte de operadores de pre- y post-incremento y decremento, estilo-C

Operador	Ejemplo	Resultado
Pre-incremento	<code>++\$x</code>	Incrementa \$x en uno, y luego devuelve \$x .
Post-incremento	<code>\$x++</code>	Devuelve \$x , y luego incrementa \$x en uno.
Pre-decremento	<code>--\$x</code>	Decrementa \$x en uno, luego devuelve \$x .
Post-decremento	<code>\$x--</code>	Devuelve \$x , luego decrementa \$x en uno.

5.8 Operadores de Lógica

Los operadores lógicos se utilizan para construir expresiones lógicas, combinando valores lógicos (true y/o false) o los resultados de los operadores relacionales.

Operador	Ejemplo	Resultado
Conjunción	(\$x and \$y) o también (\$x && \$y)	TRUE si tanto \$x como \$y son TRUE.
Disyunción	(\$x or \$y) o también (\$a \$b)	TRUE si cualquiera de \$x o \$y es TRUE.
O exclusivo (Xor)	\$x xor \$y	TRUE si \$x o \$y es TRUE, pero no ambos
Negación	!\$x	TRUE si \$x no es TRUE.

Nota: La razón para tener las dos variaciones diferentes de los operadores "and" y "or" es que ellos operan con precedencias diferentes.

5.9 Operadores de Cadena

Existen dos operadores para datos tipo string.

- El primero es el operador de concatenación ('.'), el cual devuelve el resultado de concatenar sus argumentos a lado derecho e izquierdo.
- El segundo es el operador de asignación sobre concatenación ('.='), el cual adiciona el argumento del lado derecho al argumento en el lado izquierdo.

```
<?php
$a = ";Hello ";
$b = $a . "World!"; // ahora $b contiene ";Hello World!"

$a = ";Hello ";
$a .= "World!";     // ahora $a contiene ";Hello World!"
?>
```

5.9 Operadores de Matrices

Proporcionan funcionalidades para manipular los arreglos.

Operador	Ejemplo	Resultado
Unión	\$a + \$b	Unión de \$a y \$b.
Igualdad	\$a == \$b	TRUE si \$a y \$b tienen las mismas parejas llave/valor.
Identidad	\$a === \$b	TRUE si \$a y \$b tienen las mismas parejas llave/valor en el mismo orden y de los mismos tipos.
No-igualdad	(\$a != \$b) o (\$a <> \$b)	TRUE si \$a no es igual a \$b.
No-identidad	\$a !== \$b	TRUE si \$a no es idéntico a \$b.

Nota: El operador + adiciona elementos de las claves restantes de la matriz del lado derecho a aquella al lado izquierdo, al mismo tiempo que cualquier llave duplicada NO es sobrescrita.

```
<?php
$a = array("a" => "manzana", "b" => "banano");
$b = array("a" => "pera", "b" => "fresa", "c" => "cereza");

$c = $a + $b; // Unión de $a y $b
echo "Unión de \$a y \$b: \n";
var_dump($c);
?>
```

Cuando sea ejecutado, este script producirá la siguiente salida: Unión de \$a y \$b:

```
array(3) {
  ["a"]=>
  string(7) "manzana"
  ["b"]=>
  string(6) "banano"
  ["c"]=>
  string(6) "cereza"
}
```

Los elementos de las matrices son considerados equivalentes en la comparación si éstos tienen la misma clave y valor.

```
<?php
    $a = array("manzana", "banano");
    $b = array(1 => "banano", "0" => "manzana");

    var_dump($a == $b); // bool(true)
    var_dump($a === $b); // bool(false)
?>
```

5.10 Operadores de Tipo

El operador *instanceof* permite saber si un objeto pertenece o no a una determinada clase. Por ejemplo:

```
<?php
    class MiClase {
    }

    class NoMiClase {
    }

    $a = new MiClase;
    $c = 'MiClase';
    var_dump($a instanceof MiClase);
    var_dump($a instanceof NoMiClase);
    var_dump($a instanceof $c);
?>
```

El resultado del ejemplo sería:

```
bool(true)
bool(false)
bool(true)
```

Otras utilidades de este operador pueden ser:

- Para determinar si una variable es una instancia de objeto de una clase que hereda de una clase padre,
- Para determinar si una variable es una instancia de objeto de una clase que implementa una interfaz.

6 Estructuras de Control

Todo script PHP se compone de una serie de sentencias. Una sentencia puede ser una asignación, una llamada a función, un bucle, una sentencia condicional e incluso una sentencia que no haga nada (una sentencia vacía). Las sentencias normalmente acaban con punto y coma. Además, las sentencias se pueden agrupar en grupos de sentencias encapsulando un grupo de sentencias con llaves. Un grupo de sentencias es también una sentencia. En este capítulo se describen los diferentes tipos de sentencias.

6.1 Bifurcaciones

Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el flujo de ejecución de un programa.

6.1.1 Sentencia If

Permite la ejecución condicional de fragmentos de código. Las sentencias *if* se pueden anidar indefinidamente dentro de otras sentencias *if*.

```
if (expr) {  
    código...  
}
```

Donde *expr* se evalúa su valor condicional (boolean). Si *expr* se evalúa como TRUE, se ejecutará el *código*, caso contrario ignora el código. Veamos un ejemplo:

```
<?php  
    $x = 1;  
    $y = 2;  
    if ($x < $y) {  
        echo "$x en menor que $y";  
        $x = $y - $x;  
    }  
?>
```

Nota: Si el código contiene una sola expresión se pueden omitir las llaves que marcan el comienzo y fin del bloque que acondiciona el *if*.

6.1.2 Sentencia else

A menudo queremos ejecutar una sentencia si se cumple una cierta condición, y una sentencia distinta si la condición no se cumple. Esto es para lo que sirve *else* que extiende la sentencia *if* para ejecutar determinado código en caso de que la condición del *if* se evalúe como falso.

```
<?php  
    $x = 1;  
    $y = 2;  
    if ($x < $y) {  
        echo "$x en menor que $y";  
        $x = $y - $x;  
    } else {  
        echo "$x en mayor que $y";  
        $y = $x - $y;  
    }  
?>
```

6.1.3 Sentencia elseif

Es una combinación de *if* y *else*. Como *else*, extiende una sentencia *if* para ejecutar una sentencia diferente en caso de que la expresión *if* original cuando se evalúa como FALSE. No obstante, a diferencia de *else*, ejecutará esa expresión alternativa solamente si la expresión condicional *elseif* se evalúa como TRUE.

Puede haber varios *elseif* dentro de la misma sentencia *if*, también se puede escribir *else if* (con dos palabras).

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

6.1.4 Sentencia switch

Cuando se quiere comparar la misma variable (o expresión) con muchos valores diferentes, y ejecutar una parte de código distinta dependiendo de a qué valor es igual, existe la sentencia switch. Veamos el siguiente ejemplo:

```
<?php
    $semaforo = "verde";
    switch ($semaforo) {
        case "rojo":
            echo "No puede pasar";
            break;
        case "amarillo":
            echo "Precaución";
            break;
        case "verde":
            echo "Puede pasar";
            break;
    }
?>
```

Nota: PHP continúa ejecutando las sentencias hasta el final del bloque switch, o la primera vez que vea una sentencia break. Si no se escribe una sentencia break al final de un bloque switch, seguirá ejecutando las sentencias del siguiente case.

6.2 Bucles

Un bucle se utiliza para realizar un proceso repetidas veces. El código incluido dentro de un bucle se ejecutará mientras se cumpla unas determinadas condiciones. Pueden ocurrir bucles infinitos, cuando la condición de finalizar no se llega a cumplir nunca.

6.2.1 Sentencia while

Este bucle es de la forma:

```
while (condición){
    sentencia
}
```

Su función es muy simple, mientras que la expresión *condición* se evalúe como TRUE se ejecutará la *sentencia*. Veamos un ejemplo práctico:

```
<?php
    $i = 1;
    while ($i <= 5){
        echo "$i\n";
        $i++;
    }
    // imprime 1 2 3 4 5
?>
```

En este ejemplo vemos que la variable i se ira incrementando hasta que llegue a 10, una vez que pase ese valor, la condición de continuación retornara FALSE por lo que no se ejecutara mas las sentencias que están dentro del bucle.

Otro ejemplo equivalente al anterior puede ser:

```
<?php
    $i = 0;
    while ($i++ <= 5){
        echo "$i\n";
    }
    // imprime 1 2 3 4 5
?>
```

Nota: la variable i se ira auto incrementando encada iteración del bucle.

6.2.2 Sentencia do..while

Estos bucles son muy similares a los bucles while, excepto que las condiciones se comprueban al final de cada iteración en vez de al principio.

Veamos el siguiente ejemplo:

```
<?php
    $i = 1;
    do {
        echo "$i\n";
        $i++;
    } while ($i < 6);
    // imprime 1 2 3 4 5
?>
```

6.2.3 Sentencia for

Estos son los bucles son un poco mas complejos que el resto. Se escriben de la forma:

```
for (expr1; expr2; expr3){
    sentencia
}
```

La expresión $expr1$ se ejecuta una vez al principio del bucle (inicialización).

Al comienzo de cada iteración, se evalúa la expresión $expr2$; Si se evalúa como TRUE, el bucle continúa y las sentencias anidadas se ejecutan. Si se evalúa como FALSE, la ejecución del bucle finaliza.

Al final de cada iteración, se ejecuta la expresión $expr3$.

En el siguiente script vemos una simple iteración del entero 1 al 10:

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

Nota: Cada una de las expresiones puede estar vacía.

6.2.4 Sentencia foreach

Esta sentencia funciona solamente con arreglos y si se intenta utilizar con otro tipo de dato se arrojará un error

Su función es muy sencilla, consiste en iterar sobre arreglos, y se puede usar de dos formas:

- `foreach($arreglo as $corriente) sentencia`
Esta forma recorre el arreglo dado por `$arreglo`. En cada iteración, el valor del elemento actual se asigna a la variable `$corriente` y el puntero interno del arreglo se incrementa en una unidad, así en el siguiente paso, se estará posicionado en el siguiente elemento.

```
<?php
    $frutas = array("manzana", "naranja", "banana");

    foreach ($frutas as $corriente) {
        echo $corriente;
    }
?>
```

- `foreach($arreglo as $clave => $corriente) sentencia`
La segunda forma hace lo mismo, salvo que, a diferencia de la primera, la clave del elemento actual será asignada a la variable `$clave` en cada iteración

```
<?php
    $frutas = array("enero" => 1, "febrero" => 2, "marzo" => 3);

    foreach ($frutas as $clave => $corriente) {
        echo $corriente;
    }

    /*
    ** imprime "enero => 1 febrero => 2 marzo => 3"
    */
?>
```

6.3 Sentencias break y continue

6.3.1 Sentencia break

Esta sentencia escapa de una iteración de un bucle (`while`, `do..while`, `for`, `foreach`) o de la secuencia de un `switch`.

Como funcionalidad especial es que esta sentencia acepta un parámetro opcional, el cual determina cuantas estructuras de control hay que escapar. Por ejemplo:

```
<?php
    echo "contando... ";
    $i = 0;
    while ($i < 15) {
        switch ($i) {
            case 5:
                echo "llego a \n ";
                break 1; // se escapa solo del switch pero el bucle
                        // while continua
            case 10:
                echo "llego a diez y para acá\n";
                break 2; // se escapa del switch y del while
        }
        echo "$i,\n";
        $i++;
    }
?>
```

Nota: observar que la salida de este script no contendrá en número 10 (*contando... 0, 1, 2, 3, 4, llego a 5, 6, 7, 8, 9, llego a diez y para acá*).

6.3.2 Sentencia continue

Se usa dentro de la estructura del bucle para saltar el resto de la iteración actual del bucle y continuar la ejecución al comienzo de la siguiente iteración.

6.4 Sentencia return

Retorna el valor que se le pasa como parámetro, además si es llamado desde una función, termina inmediatamente la ejecución, esto también sucede si es llamado desde un script PHP.

6.5 Bloque try and catch

Estos tipos de sentencias permiten controlar el manejo de excepciones que se pueden producir en tiempo de ejecución de un programa. Para mayor información ver el capítulo de excepciones

6.6 Sentencias include y require

Estas sentencias tienen la función de incluir archivos. El formato de estos puede ser variado; por ejemplo podemos incluir en nuestro script un archivo de texto o un documento html, además, podemos incluir archivos de código php para usar las definiciones que se encuentren en dicho archivo.

Existen dos tipos de sentencias de inclusión.

6.6.1 Sentencia require

La sentencia *require* incluye y evalúa el archivo especificado. Si no encuentra el archivo en la ubicación pasada como parámetro, se desplegará un error fatal.

```
<?php
    require 'archivo.php';
    require $archivo;
    require ('archivo.txt');
?>
```

Nota: para estar seguro de que se el archivo se incluye una sola vez, existe la sentencia *require_once()* la cual verifica que si el archivo se incluyó una vez, no se volverá a incluir de nuevo.

6.6.2 Sentencia include

Esta sentencia tiene la misma funcionalidad que su par, *require*. La única diferencia es en el modo de actuar ante un error. La sentencia *include* produce una advertencia mientras que *require* produce un error fatal.

Nota: para estar seguro de que se el archivo se incluye una sola vez, existe la sentencia *include_once()* la cual verifica que si el archivo se incluyó una vez, no se volverá a incluir de nuevo.

7 Funciones

Una función incorpora una expresión para ser evaluada y cuando sea llamada retornara un valor como resultado.

Para definir una función se realiza por medio de la siguiente sintaxis:

```
<?php
    function HelloWorld ($arg_1, $arg_2, ..., $arg_n){
        echo "Hello World\n";
    }
?>
```

7.1 Funciones Condicionales

En PHP es posible definir una función dependiendo de una condición lógica.

```
<?php
    $i = 1;
    if ($i == 0) {
        function HelloWorld(){
            return "hello world";
        }
    }

    echo HelloWorld(); // no imprime nada ya que la función no esta
                      // definida
?>
```

7.2 Funciones dentro de funciones

También se pueden declarar funciones dentro de una función.

```
<?php
    function mat($arg){
        function inc($arg){
            $arg++;
            return $arg;
        }
        echo inc($arg);
    }

    mat(1); // imprime 2

    inc(3); // imprime 4
?>
```

7.3 Parámetros

Un identificador utilizado dentro de una función denota un argumento que es llamado parámetro. Dicho parámetro puede ser de cualquiera de los tipos que soporta PHP.

Los parámetros de una función se pasan por valor, de manera que se copia el parámetro actual de la función en el parámetro formal y todas las modificaciones hechas a los valores del parámetro formal durante la ejecución de la función, se pierden cuando éste termina.

```
<?php
    function inc($arg){
        $arg++;
        return $arg;
    }

    $i = 0;
    inc($i);
    echo $i; // imprime 0
?>
```

7.3.1 Pasar parámetros por referencia

Para poder simular el pasaje de parámetros por valor-resultado lo debemos hacer por medio de referencias; esto puede ser pasando ya en la llamada una referencia del parámetro, o anteponiendo el símbolo & delante del parámetro en la cabecera de la función. De esta última forma siempre, la función, tomara una referencia del parámetro pasado.

```
<?php
    function inc(&$arg){
        $arg++;
        return $arg;
    }

    $i = 0;
    inc($i);
    echo $i; // imprime 1
?>
```

7.3.2 Parámetros por defecto

En PHP es posible definir valores por defecto a aquellos parámetros que no se pasaron cuando se invoca a la función.

```
<?php
    function hello($arg = "world"){
        return "hello ".$arg."!\n";
    }

    echo hello(); // imprime "hello world!"

    echo hello("moto"); // imprime "hello moto!"
?>
```

7.4 Retorno de valores

Los valores se retornan, en una función, usando la sentencia *return*. Puede devolverse cualquier tipo de valor, incluyendo listas y objetos.

```
<?php
    function inc($arg){
        return $i++;
    }

    echo inc(0); // imprime 1
?>
```

8 Referencias

Las Referencias en PHP son un medio para acceder al mismo contenido de una variable pero con diferentes nombres. No son como los punteros de C, sino que son alias en la tabla de símbolos. Hay que tener en cuenta, que en PHP el nombre de una variable y el contenido de una variable son diferentes, de manera que el mismo contenido, puede tener varios nombres. Por ejemplo:

```
<?php
    $var = 3;
    $ref =& $var;
    $var = 0;
    echo $ref; // imprime 0
?>
```

Significa que *\$var* y *\$ref* apuntan al mismo contenido, pero no es que *\$var* esté apuntando a *\$ref* o viceversa, sino que tanto *\$var* como *\$ref* apuntan al mismo lugar.

8.1 Destruir una referencia

Para borrar o destruir una referencia se usa la sentencia *unset()*.

Cuando se borra una referencia, solo se rompe esa unión entre el nombre de la variable y el contenido. Esto no significa que el contenido haya sido destruido. Por ejemplo:

```
<?php
    $var = 3;
    $ref =& $var;
    unset($ref); // no destruiría $b, solo $a.
    echo $var; // imprime 3
?>
```

8.2 Referencias colgadas

Una referencia a una variable cuyo tiempo de vida no es largo (o ha finalizado) se denomina una referencia colgada.

Veamos un ejemplo.

```
<?php
    $var = 3; // defino una variable y la inicializo con el valor 3
    $ref = $var; // referencio ref a la variable var
    unset($var); // destruyo la variable var
    echo $ref; // imprime "3"
    echo $var; // arroja un error "Undefined variable"
?>
```

Nota: la referencia *ref* sigue reverenciando al valor de la variable *var*, a pesar que esta última no existe más.

9 Programación Orientada a Objetos

PHP soporta el paradigma de programación orientada a objeto. En sí, lo que cambia, es que no se tienen más valores aislados y operaciones que se aplican a esos valores, sino que se habla de objetos, en los cuales, ya tenemos encapsulado tanto el valor como las operaciones dentro de él.

Las principales características de la POO son:

- *Abstracción*: Denota las características esenciales de un objeto. Además, estos objetos, definen su comportamiento sin revelar cómo se implementan.
- *Encapsulamiento*: Reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción.
- *Principio de ocultación*: Cada objeto está aislado del exterior y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas.
- *Polimorfismo*: significa la capacidad de un operador o método para responder en función de los parámetros utilizados durante su invocación.
- *Herencia*: Una clase puede heredar la estructura (atributos) y comportamiento (métodos) de otra clase, de esta forma se establece una jerarquía de clases. Además es posible redefinir los métodos heredados por el padre.

9.1 Objetos

Son entidades provistas de un conjunto de propiedades (atributos) y de comportamiento (métodos). Un objeto es una instancia a una clase.

9.2 Clases

Dicha encapsulación se logra a través de las clases, que definen un tipo (objeto) con los valores y operaciones que este tipo puede tener.

Una clase es un contenedor de objetos que tienen una misma estructura y comportamiento.

Para definir una clase lo hacemos mediante la palabra *class*, seguida por un nombre de clase, el cual puede ser cualquier nombre que no esté en la lista de palabras reservadas en PHP. Seguida por un par de llaves curvas, las cuales contienen la definición de atributos y métodos.

9.2.1 Atributos

Los atributos definen la estructura de los objetos de una clase, es decir, la información que estos contienen.

Para definir los atributos de una clase simplemente se declaran como las variables y hasta es posible inicializarlas si se quiere. También es posible definir la visibilidad de cada atributo, mas adelante se pasara a explicar este concepto.

9.2.2 Métodos

Los métodos definen lo que el objeto puede hacer. Estos pueden producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.

Para definir los métodos de una clase es de la misma forma que se definen las funciones. También es posible definir la visibilidad de cada método, mas adelante se pasara a explicar este concepto.

Una buena costumbre de programación OO es que, en la definición de una clase, declarar para cada atributo las operaciones *get()* y *set()*, las cuales son las encargadas de acceder y modificar los atributos de una clase. Esto se hace ya que es recomendable, por seguridad, que los atributos de una clase solo puedan utilizarse dentro de la clase, es decir con visibilidad en *private* (ver visibilidad).

Ejemplo

Supongamos que queremos definir un tipo nuevo que represente un vehículo:

```
<?php

class Vehiculo {

    //Atributos
    private $patente;
    private $marca;

    // Metodos
    public function getPatente(){
        return $this->patente;
    }

    public function getMarca(){
        return $this->marca;
    }

    public function setPatente($newPatente){
        $this->patente = $newPatente;
    }

    public function setMarca($newMarca){
        $this->marca = $newMarca;
    }
}

?>
```

Para poder instanciar una la clase vehiculo y de esa forma obtener un objeto vamos a ver el siguiente script:

```
<?php

include ("vehiculo.php"); // necesito incluir si o si la clase
$auto = new vehiculo(); // se realiza la instancia
$auto->setPatente("TYU890"); //convoco a un método el objeto instanciado
echo $auto->getPatente();

?>
```

Nota: cada vez que queramos crear un objeto de una clase debemos usar la sentencia *new* y luego el método que la construye.

9.3 Constructores y destructores

Los constructores y destructores son, desde el punto de vista sintáctico, métodos que ejecutan alguna tarea relativa a un objeto particular.

Desde el punto de vista desde la programación orientada a objetos, estos, son métodos que permiten establecer el estado inicial y final de un objeto.

9.3.1 Constructores

Un constructor de una clase es un método que se ejecuta cada vez que se crea un nuevo objeto, como es el caso del ejemplo anterior en la línea:

```
$auto = new vehiculo();
```

El constructor siempre tendrá el nombre de la clase y el objetivo de este, además de construir un objeto de la clase, es para cualquier inicialización que el objeto puede necesitar antes de ser usado o la invocación a alguna rutina.

Ejemplo

Para definir este método en una clase debemos declarar una función con el nombre de `__construct` la cual puede tomar parámetros y con ellos inicializar algún atributo.

```
<?php
class Vehiculo {

    // Atributos
    private $patente;
    private $marca;

    // Constructor
    function __construct($pat, $marc) {
        $this->patente = $pat;
        $this->marca = $marc;
    }
}
?>
```

Ahora bastara con solo instanciar la clase con los parámetros y en una sola línea tenemos ya inicializado nuestro objeto:

```
$moto = new moto("TYU890", "enduro");
```

Nota:

Si se quisiera solo inicializar los atributos, no hace falta crea un constructor, solo uno inicializa los atributos cuando los define dentro de la clase.

9.3.2 Destruidores

El método destructor será llamado tan pronto como todas las referencias a un objeto en particular sean removidas o cuando el objeto sea explícitamente destruido, o en cualquier orden en la finalización de la ejecución.

Sólo debemos definirlo si deseamos hacer alguna rutina cuando un objeto se elimine de la memoria.

Ejemplo

Para definir este método en una clase debemos declarar una función con el nombre de `__destruct`.

```
<?php
class Vehiculo {

    // Atributos
    private $patente;
    private $marca;

    // Constructor
    function __construct($pat, $marc) {
        $this->patente = $pat;
        $this->marca = $marc;
    }

    // Destructor
    function __destruct() {
        print ("destruyendo...");
    }
}
?>
```

Cuando se destruya el objeto se ejecutara nuestro código que definimos dentro del destructor. En este caso vamos a destruirla de manera explícita con la función `unset()`:

```
$m = new moto();
unset($m); // Se destruye el objeto
```

9.4 Visibilidad (scope)

La visibilidad de un atributo o método puede ser definida al anteponerle a la declaración con las palabras reservadas. Ellas pueden ser:

- `public`: pueden ser accedidos de todos lados.
- `protected`: solo las clases hijas podrán acceder.
- `private`: únicamente se acceden dentro de la clase que se definieron.

Nota: si no es definido la visibilidad, se toma por defecto *public*.

9.5 Clases y métodos abstractos

Las clases abstractas presentan un nivel de abstracción tan elevado que no sirven para instanciar objetos de ellas. Representan los escalones más elevados de algunas jerarquías de clases y solo sirven para derivar otras clases, en las que se van implementando detalles hasta que finalmente presentan un nivel de definición suficiente que permita instanciar objetos concretos. Se suelen utilizar en aquellos casos en que se quiere que una serie de clases mantengan una cierta característica o interfaz común.

En PHP, cualquier clase que contenga por lo menos un método abstracto debe también ser abstracta. Los métodos definidos como abstractos simplemente declaran los métodos pero no pueden definir la implementación.

Cuando se hereda desde una clase abstracta, todos los métodos marcados como abstractos en la declaración de la clase padre, deben de ser definidos por la clase hijo.

Ejemplo

Si nos abstraemos un poco, no necesitaremos instanciar una clase vehiculo, y si, tener objeto como moto, auto, camiones, etc. Todos estos comparten la estructura pero a su vez cada uno tiene características diferentes. Por lo que definir la clase vehiculo como abstracta hace un mejor modelamiento de la herencia.

Además si observamos tenemos un método abstracto en la clase vehiculo que luego se implementara en cada una de sus hijos, en este caso en la clase moto.

```
<?php
    abstract class Vehiculo{

        private $patente;
        private $marca;

        abstract protected function getMaxSpeed();
    }
?>

<?php
include ("vehiculo.php");
class moto extends vehiculo{

    private $tipo;

    function getMaxSpeed(){
        return 60;
    }
}
?>
```

9.6 Interfaces

Las interfaces de objetos permiten crear código el cual especifica métodos que una clase debe implementar, sin tener que definir como serán manejados esos métodos. Todos los métodos en una interface deben ser públicos.

Todos los métodos de una interface deben ser implementados dentro de una clase; de no hacerse así resultará en error fatal. Las clases pueden implementar más de una.

Ejemplo

Supongamos que definimos una interfase que tenga métodos que permiten encender y apagar algo.

```
interface encendible {
    public function encender();
    public function apagar();
}
```

Bien, ahora vamos a definir una clase que implemente estos métodos y cuyo objeto tengo la propiedad de encenderse y apagarse, por ejemplo un foco.

```
class foco implements encendible {
    public function encender(){
        echo "y la luz se hizo";
    }

    public function apagar(){
        echo "estamos a oscuras";
    }
}
```

Como a una interfase puede ser implementada por más de una clase, supongamos ahora que tenemos otra clase que la implemente pero que no comparta en si la estructura con la clase foco. Digamos, por ejemplo, un auto que se puede encender y apagar.

```
class auto implements encendible{
    private $gasolina;
    private $bateria;
    private $estado = "apagado";

    function __construct(){
        $this->gasolina = 0;
        $this->bateria = 10;
    }

    public function encender(){
        if ($this->estado == "apagado") {
            if ($this->bateria > 0) {
                if ($this->gasolina > 0){
                    $this->estado = "encendido";
                    $this->bateria--;
                    echo "encendido";
                }else {
                    echo "no hay gasolina";
                }
            }else {
                echo "no hay batería";
            }
        }else {
            echo "ya estaba encendido";
        }
    }

    public function apagar(){
        if ($this->estado == "encendido"){
            $this->estado = "apagado";
            echo "estoy apagado";
        }else {
            echo "ya estaba apagado";
        }
    }
}
```

Notaremos que los métodos encender y apagar se comportan de manera diferente en cada una de las clases que implementa la interfase encendible

En conclusión podemos deducir que tanto los autos como los focos se pueden encender y apagar. Así pues, podemos llamar al método *encender()* o *apagar()*, sin importarnos si es un auto o un foco.

Las interfaces permiten el tratamiento de objetos sin necesidad de conocer las características internas de ese objeto y sin importar de qué tipo son... simplemente tenemos que saber que el objeto implementa una interfaz.

A todo esto sale una pregunta al aire, ¿Cuál es la diferencia entre una clase abstracta y una interfase?.

Ambas contienen declaraciones de métodos sin la necesidad de implementarlos, pero existen diferencias entre las dos:

- Una clase no puede heredar de dos clases abstractas, pero sí puede heredar de una clase abstracta e implementar una interface, o bien implementar dos o más interfaces.
- Las interfaces permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo comportamiento.
- Las interfaces permiten “publicar” el comportamiento de una clase desvelando un mínimo de información.
- Las interfaces tienen una jerarquía propia, independiente y más flexible que la de las clases, ya que tienen permitida la herencia múltiple.

9.7 Herencia

Se puede construir una clase a partir de otra mediante el mecanismo de la herencia. Para indicar que una clase deriva de otra se utiliza la palabra *extends*, como por ejemplo:

```
class moto extends vehiculo {
    . . . .
}
```

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser redefinidas (overridden) en la clase derivada, que puede también definir o añadir nuevas variables y métodos.

Una clase extendida siempre depende de una clase base única, lo que quiere decir que una clase no puede heredar de más de una clase, entonces PHP no soporta herencia múltiple.

Es posible simular una herencia múltiple mediante el uso de interfaces.

Ejemplo:

```
<?php
include ("vehiculo.php");
class moto extends vehiculo {

    //Atributos
    private $tipo;

    // Metodos
    public function getTipo(){
        return $this->tipo;
    }

    public function setTipo($newTipo){
        $this->tipo = $newTipo;
    }
}

?>
```

```

<?php

include ("moto.php");

$moto = new moto();
$moto->setPatente("TYU890");
$moto->setTipo("enduro");

echo $moto->getPatente();
echo $moto->getTipo();

?>

```

9.8 Constantes de clase

En PHP es posible declarar constantes en una clase. Estas, al igual con los miembros estáticos, no pueden ser accedidas desde una instancia de un objeto de dicha clase.

Ejemplo

```

<?php

class HelloWorld {
    const constant = 'Hello World';

    function printConstant() {
        echo self::constant . "\n";
    }
}

?>

```

Probamos nuestra clase:

```

<?php

include ("HelloWorld.php");
$class = new HelloWorld();
$class->printConstant(); // imprime "Hello World"
echo $class->constant; // arroja un error de que no conoce la propiedad
                       "constant" de la clase

?>

```

9.9 Atributos o métodos estáticos.

Un atributo o método declarado como estático no puede ser accedido por medio de una instancia del objeto y tampoco puede ser redefinido en una extensión de la clase.

Es posible acceder a una declaración estática sin haber instanciado la clase donde se encuentre.

Para llegar a ellos es posible mediante el operador de resolución (::)

Veamos un ejemplo de un atributo estático:

```

<?php

class Persona {
    public static $tipoDocumento = 'DNI';

    public function obtenerTipo() {
        return self::$tipoDocuemnto; // self es lo mismo que poner el nombre
                                       // de la clase
    }
}

?>

```

Ahora hacemos otra clase que hereda de Persona

```

class Cliente extends Persona {
    public function getTipoDocumento(){
        return parent::$tipoDocumento;
    }
}

```

Y probamos un poco desde un script

```
// Accedemos por medio del operador ::
print Persona::$tipoDocumento; // imprime "DNI"

// Intentamos acceder por una instancia de un objeto
$pers = new Persona();
echo $pers->tipoDocumento; // Warning: Undefined property
echo $pers->obtenerTipo(); // imprime "DNI"

// Ahora probamos por medio de una clase que extiende
print Cliente::$tipoDocumento; // imprime "DNI"
$clt = new Cliente();
echo $clt->getTipoDocumento(); // imprime "DNI"
echo $clt->$tipoDocumento; // Fatal error: Cannot access empty property
?>
```

Nota: los métodos estáticos operan de la misma forma que los atributos estáticos solo que en vez de un valor fijo ellos contienen un bloque de código.

9.10 Final

Para definir que un método en una clase padre no pueda ser extendido por sus hijos, existe la sentencia final.

Ejemplo:

```
final public function getPatente(){
    . . . .
}
```

Nota: si se quiere redefinir un método que tiene la cláusula final, PHP arrojará un error fatal.

También se puede aplicar este concepto a una clase, lo cual indicará que ninguna clase puede extender de ella.

Ejemplo

```
final class Vehiculo {
    . . . .
}
```

Nota: si se intenta heredar de una clase definida como final, PHP arrojará un error fatal.

9.11 Clonado de Objetos

En PHP es posible copiar un objeto con todas sus propiedades y referencias por medio de la sentencia *clone*.

Ejemplo:

```
<?php
...
$objeto2 = clone $objeto1;
...
?>
```

Nota: El *objeto2* pasa a ser un clon del *objeto1*, es decir que el *objeto2* tiene los mismos valores y las mismas referencias que el *objeto1*

9.12 Type Hinting

Las funciones ahora son capaces de forzar que los parámetros sean objetos especificando el nombre de la clase en el prototipo de la función.

Ejemplo:

Definamos una clase *MyString* que tenga como atributo una cadena

```
<?php
    class MyString {
        public $text;

        function getText() {
            return $this->text;
        }

        function setText($arg) {
            $this->text = $arg;
        }
    }
?>
```

Ahora modifiquemos un poco la clase *HelloWorld* que definimos anteriormente.

```
<?php
    class HelloWorld {
        function printString (MyString $mystring) {
            echo $mystring->text;
        }
    }
?>
```

Nota: Observemos que el argumento de la función *printString()* esta forzado a que sea una instancia de la clase *MyString*.

Ahora vamos a usar un poco la función *printString()* y veremos que pasa si no le pasamos un objeto de la clase *MyString*.

```
<?php
    include ("HelloWorld.php");
    include ("mystring.php");

    $class = new HelloWorld();

    $mystring = new MyString();
    $mystring->setText('hello');
    $class->printConstant($mystring); // imprime "hello"

    $class->printConstant('hello'); // retorna un error fatal: Argument 1
                                    // passedto HelloWorld::printConstant()
                                    // must be an instance of MyString
?>
```

9.13 Polimorfismo

Como dijimos al principio de este capítulo, el polimorfismo significa la capacidad de un operador o método para responder en función de los parámetros utilizados durante su invocación.

9.13.1 Polimorfismo por sobrecarga

Una operación esta sobrecargada si tiene un comportamiento diferente cuando se lo aplica a diferentes tipos. Una operación es cuando los métodos de las subclases se pueden sobrecargar.

Ejemplo

Supongamos que queremos definir un método *sumar()* que tome como argumento dos operándos que pueden ser tanto enteros o cadenas. Para el primer caso retornara la suma de dicho enteros y para el caso de las cadenas simplemente las concatenara.

Para lograr esto usaremos el método `__call()` que se trata de un método "mágico" que permite capturar la invocación de métodos no existentes.

```
<?php
class test {
    function __call($method_name, $arguments) {
        if ($method_name == sumar){
            if ((is_int($arguments[0])) and
                (is_int($arguments[1]))){
                return($arguments[0]+$arguments[1]);
            }else if ((is_string($arguments[0])) and
                (is_string($arguments[1]))){
                return ($arguments[0].$arguments[1]);
            }
        }
        else {
            return ("función no definida");
        }
    }
}
?>
```

Nota: en ve de hacer los cálculos dentro de la función `__call()` podemos llamar a dos métodos, por ejemplo `sumarInt()` y `sumarString()` ya que de esa forma quedaría mas legible.

Ahora la ponemos a prueba.

```
<?php
include ("test.php");
$sum = new test();
echo $sum->sumar("hello ", "world"); // imprime "hello world"
echo $sum->sumar(1,9); // imprime 10
echo $sum->sumar(1, "9"); // no imprime nada
?>
```

Entonces logramos sobrecargar la operación sumar.

9.13.2 Polimorfismo por inclusión

La herencia junto a la sobrecarga de operadores permiten hacer polimorfismo sobre los métodos de la clase padre.

Por ejemplo imaginemos un juego de ajedrez con los objetos *rey*, *reina*, *alfil*, *caballo*, *torre* y *peón*, cada uno heredando el objeto *pieza*.

El método movimiento de la clase padre podría, usando polimorfismo por inclusión, hacer el movimiento correspondiente de acuerdo a la clase objeto que se llama.

```
abstract class pieza {
    abstract protected function mover();
}

class torre extends pieza {
    public function mover(){
        return "moviendo torre";
    }
}

class alfil extends pieza {
    public function mover(){
        return "moviendo alfil";
    }
}

$t = new torre();
echo $t->mover(); // imprime "moviendo torre"

$a = new alfil();
echo $a->mover(); // imprime "moviendo alfil"
?>
```

9.13.3 Polimorfismo paramétrico

También conocido como genericidad, es cuando una operación o declaración tiene argumentos que son variables de tipos.

En PHP no soporta polimorfismo paramétrico ya que no es posible hacer declaraciones pasando tipos como argumentos.

10 Excepciones

Durante la ejecución de un programa, suelen ocurrir sucesos o condiciones que se podrían considerar como excepcionales. En vez de continuar con la ejecución normal del programa se necesita llamar a un subprograma para llevar a cabo cierto procesamiento especial.

Para manejar las excepciones, PHP propone un bloque de control para permitir atrapar una excepción y responder, ante ella, con un determinado procedimiento.

Veamos un ejemplo práctico y pasemos a explicarlo.

```
<?php
    try {
        $conn = MYSQL_CONNECT("localhost", "root", "root");
        // intento conectarme a una base de datos
        if ($conn == false){
            throw new Exception("fallo la conexión a MySQL");
            // como la conexión fallo lanzo una excepción
        }
    } catch (Exception $e) {
        echo $e;
        // imprime "exception 'Exception' with message 'no conecta a
        // MySQL'"
    }
?>
```

El código dentro del bloque *try* está "vigilado": Si se produce una situación anormal y se lanza por lo tanto una excepción el control salta o sale del bloque *try* y pasa al bloque *catch*, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques *catch* como sean necesarios, cada uno de los cuales tratará un tipo de excepción.

Por medio de la sentencia *throw* podemos lanzar una excepción.

11 Concurrencia

Es cuando múltiples procesos son ejecutados al mismo tiempo, y por lo general, interactúan entre sí compartiendo un mismo recurso.

PHP provee de forma nativa funciones que permiten utilizar los semáforos para procesos concurrentes.

11.1 Como funcionan los semáforos

Los semáforos son elementos de sincronización que permiten sincronizar el uso de un recurso compartido en un sistema computacional. En otras palabras, permiten asegurar que solo un número determinado de procesos pueda acceder a un recurso simultáneamente, por ejemplo, escribir en un archivo de texto un contador de visitas.

Primero que todo se tiene que identificar al bloque que puede llegar a producir problemas cuando la concurrencia este presente, a este bloque se lo denomina región crítica, Por ejemplo el momento que abrimos el archivo de texto para poder editarlo.

Antes de entrar a una región crítica, el proceso debe solicitar autorización al semáforo para adquirir al semáforo. Al terminar la región crítica, el proceso libera el semáforo para que otro lo tome.

En otras palabras es que antes de entrar a la región crítica, el proceso intenta de tomar "una banderita" del semáforo; si hay banderitas disponibles entonces el proceso continua su ejecución, de lo contrario espera hasta que el o los otros procesos que pasaron antes "liberen una banderita", para continuar la ejecución.

11.2 Gestión de semáforos

A continuación se describirán las funciones en PHP que permiten la gestión de los semáforos. Esta extensión no esta disponible en plataformas Windows.

sem_get

Permite crear un semáforo en el sistema operativo o recuperar uno existente. Recibe como parámetro obligatorio un entero con el identificador único del semáforo en el sistema.

Como parámetros opcionales tenemos el valor inicial del semáforo (por defecto 1), permisos de acceso (en permisos unix) y un booleano para auto liberación al terminar el script.

```
$sem = sem_get($key, 1);
```

Nota: en este ejemplo se crea un semáforo inicializándolo en 1, es decir, que solo un proceso a la vez podrá estar en la región crítica.

sem_acquire

Adquiere un semáforo ya creado en el sistema recibiendo como parámetro el identificador entregado por sem_get. Si no existe, el comando lanza un warning y retorna FALSE.

```
sem_acquire($sem);
```

Nota: Si existe el semáforo *sem*, se bloqueara la ejecución del script hasta que se libere el semáforo.

sem_release

Realiza una liberación al semáforo indicado habilitando a que otro proceso lo tome.

```
sem_release($sem);
```

sem_remove

Elimina un semáforo del sistema operativo, haciéndolo inaccesible en el futuro.

```
sem_remove($sem);
```

11.3 Ejemplo

Supongamos una función que escribe un mensaje en un archivo, debemos asegurarnos que un solo proceso escriba a la vez en el archivo para mantener la consistencia del mismo. Entonces recurrimos a usar las funciones de semáforos que vimos.

```
function log($mensaje) {  
  
    $sem = sem_get(3, 1); // creamos o obtenemos el semáforo  
    sem_acquire($sem);   // pedimos permiso al semáforo  
  
    $f = fopen("./archivo.log", "a");  
    fwrite($f, $mensaje);  
    fclose($f);  
  
    sem_release($sem); // la liberamos para que otro proceso pueda tomar el  
                       // semáforo  
}
```

12 Extendiendo PHP

PHP maneja el concepto de "extensión" cuando se refiere a ciertas funcionalidades que se le incorporan al intérprete pero que por defecto no vienen disponibles. Por ejemplo una de ellas son los threads.

Entonces para extender PHP es posible hacerlo por medio de bibliotecas dll (Dynamic Linking Library) o so (Shared Object) que solo PHP entienda.

Estas bibliotecas pueden estar creadas en C o en cualquier lenguaje, solo hay que compilarla con las especificaciones de las extensiones PHP para puedan ser usadas de manera transparente usarlas tan "transparente" como a una función por su nombre.

12.1 Configuración e instalación de extensiones

Las extensiones se colocan en la ubicación php/ext y luego es necesario editar el archivo de configuración de PHP, el php.ini.

Lo que se hace allí es cargar la extensión para que PHP lo reconozca y pueda ser usado, esto se realiza simplemente colocando la siguiente línea:

```
extension=php_threads.dll
```

También es posible cargar una biblioteca dinámicamente en tiempo de ejecución por medio del comando *dl()*. Veamos un ejemplo:

```
<?php
    if (!extension_loaded(' threads ')) {
        dl('php_threads.dll');
    }
?>
```

Nota: se usa la función *extension_loaded()* para comprobar si la biblioteca esta ya cargada o no, esta devolverá un booleano. Retorna true si verifica que el nombre, que se le paso como argumento, es una biblioteca que ya esta para usarse.

12.2 Creación de una nueva biblioteca

Como dijimos anteriormente una biblioteca para PHP debe respetar ciertas normas para que PHP pueda usarla sin problemas.

Lo mas común es hacer bibliotecas para PHP implementadas en C ya que *¡PHP esta hecho en C!*, dejo el link de un post que dice con detalles como hacer una biblioteca desde C (<http://www.forosdelweb.com/f18/extensiones-php-380300/#post1474523>).

En delphi, las últimas versiones, existen componentes que permiten crear una dll para PHP mucho más fácil que escribirla desde cero en C. En este link esta el articulo de cómo hacerlo (<http://users.telenet.be/ws36637/php4delphi.html>).

12.3 PHP Java Bridge

Nosotros podemos extender a PHP por medio de las bibliotecas bajo cualquier lenguaje, pero además se puede crear una extensión que haga de puente entre PHP y otro lenguaje.

El puente PHP/Java es un protocolo de red que permite conectar en ambas direcciones scripts en PHP con clases Java, EJB, VB.NET, C#, JRuby, etc.

Realicemos una clase en java:

```
public class HelloWorld {
    public String show(){
        return "Hello World";
    }
}
```

Lo guardamos como HelloWorld.java y lo compilamos:

```
javac HelloWorld.java
```

Una vez creado el .class creamos un jar:

```
jar cvf HelloWorld.jar HelloWorld.class
```

Ahora creamos un script en PHP donde instanciaremos a nuestra clase java.

```
<?php
    java_require("HelloWorld.jar");
    $hw = new Java('HelloWorld'); // instancio la clase hola de java
    echo (String) $hw->show(); // llamamos a un metodo de nuestra clase en
                                // java
?>
```

El resultado de este script, es claro, que imprime por pantalla *Hello World!*.

12.4 COM

Ahora que pasa si queremos usar una biblioteca que no fue especialmente diseñada para PHP. Para ello se puede usar COM (Component Object Model) como interfase para la dll que uno quiera usar.

No es muy util usar el componente COM porque solo es válido en la familia de Windows por lo que no lo recomiendan mucho ya que la mayoría de los servicios de hosting de hoy en día están bajo servidores con Linux.

13 Conclusión

En cuanto al lenguaje

Hoy en día cada vez son más los sistemas que se adaptan a la red de redes, para diferentes motivos, alguno de ellos son:

- Para publicar sus servicios y productos.
- Operar remotamente el sistema con un simple navegador.
- Una fácil interacción con el usuario
- Otros más.

Los motivos que pueden llevar a elegir PHP como un lenguaje de programación a usa es que fue diseñado desde cero con el fin único de diseñar aplicaciones web. Esto quiere decir que las tareas más habituales en el desarrollo de estas aplicaciones, pueden hacerse con PHP de forma fácil, rápida y efectiva. Otros lenguajes, como C, C++, Perl o Java serán sin duda más completos y potentes, pero no fueron diseñados con este enfoque especializado.

A mi criterio las ventajas que tiene php son:

- Es multiplataforma.
- Es open source, por lo tanto podemos encontrar mucho material de aprendizaje en Internet ya que es una comunidad muy grande, además es gratuito y podemos acceder al fuente y modificarlo si quisiéramos.
- La sintaxis de PHP es similar a la del C.
- Puede interactuar con muchos motores de bases de datos por medio de componentes ya incluidos con el lenguaje.
- Es rápido, esta escrito en C y lo podemos ejecutar desde un servidor web como el Apache.

Y las desventajas:

- Es muy flexible, esto a muchos puede parecer mas una ventaja que una desventaja, pero yo considero que PHP al se tan flexible le da la posibilidad al programador a escribir código no estructurado e ilegible.

Terminando, PHP es un lenguaje de pocos años de madures y ha llegado a tener una fuerte adhesión por los programadores. Sin duda este lenguaje que empezó como un conjunto de script promete mucho y la continuidad esta garantizada gracias a que PHP no depende de ninguna organización con fines comerciales, como dije anteriormente, es OPEN SOURCE.

En cuanto a este proyecto

Es bueno conocer los diferentes aspectos de un lenguaje, y que de esta forma queda abierta la puerta al estudio de otros, para luego, elegir bien un lenguaje de programación según las necesidades del sistema que se quiera implementar.

También como experiencia que aporto este trabajo es que, investigando artículos en la web y leyendo algún que otro manual que ande por ahí, me di cuenta que hay que tener los ojos bien abiertos ya que nos podemos encontrar con conceptos básicos, que hacen al estudio de un lenguaje, equivocados y de esto formar un mal aprendizaje.

14 Bibliografía

- The oficial PHP manual. <http://www.php.net/manual/es/>
- Wikipedia, la enciclopedia de contenido libre. <http://es.wikipedia.org/>
- Desarrollo Web, portal de la programación. <http://www.desarrolloweb.com/php/>
- Google, varios foros, blogs, etc. encontrados por el buscador. <http://www.google.org/>
- Lenguajes de programación: diseño e implementación. Tercera Edición. Pratt - Zelkowitz.
- Lenguajes de programación y modelos de computación – Marcelo Arroyo