# Expreso PHP y la vía correcta

*Versión 0.0.1*

**Yeshua Rodas**

**28 de marzo de 2019**

# Índice general

**Advertencia:** Este libro es un *trabajo en progreso* para proveer de un nuevo libro de referencia sobre como programar *correctamente* con PHP. El libro comenzó como una *traducción libre* de *PHP The Right Way*, y si bien aun hay muchas partes que en efecto eso son, también hay otras que más que traducciones son *adaptaciones* y *modificaciones* sobre el texto original en inglés, así como agregados y partes redactadas por mi propia cuenta. Por ahora, aproximadamente he *terminado* 1/3 del libro, y si se compara con *PHP The Right Way* se verá que **Expreso PHP y la vía correcta** aunque versa sobre la misma idea, se aleja en forma y estructura de aquel.

CAPÍTULO 1

Tabla de contenido

## 1.1 Expreso PHP y La vía correcta

PHP es un lenguaje de programación potente, flexible y dinámico, de código abierto, diseñado principalmente para el *desarrollo Web* y que ha alcanzado una tremenda popularidad. Hoy por hoy muchísimos sitios, portales y aplicaciones Web son potenciadas por PHP, y cada día nuevos proyectos se desarrollan con este lenguaje.

No obstante, por Internet pulula un montón de información desfasada sobre PHP que lleva a los nuevos desarrolladores por el mal camino, propagando malas prácticas de programación y código inseguro. **Expreso PHP y la vía correcta** es una referencia ligera *fácil de leer* para aprender a programar con este lenguaje; expone las convenciones más populares de programación en PHP, provee enlaces a tutoriales y material de referencia de calidad que ronda por Internet, así como se presentan algunas de las mejores prácticas que los contribuyentes han considerado a la fecha.

*No existe una forma canónica para programar con PHP* propiamente dicho. Este texto pretende guiar a los nuevos desarrolladores de PHP hacia temas que podrían no descubrir hasta que es muy tarde, así como también pretende dar a los profesionales ideas frescas sobre esos temas que han estado trabajando por años sin reconsiderarlos. Este texto expone sugerencias sobre herramientas a utilizar de entre la amplia variedad que existe, explicando las diferencias de enfoque y casos de uso, sin mayor preferencia de alguna sobre otra.

Este es un documento vivo y continuará actualizándose con más información útil y ejemplos a medida vayan haciéndose disponibles.

### 1.1.1 El libro

Este texto comenzó como una traducción de *PHP: The Right Way*, pero a medida que lo traducía me encontraba con más y más cosas que deseaba cambiar, explicar de manera distinta, cambiar de orden, incluso agregar otros temas. Al final he terminado haciendo una versión que **no** es una traducción de *PHP: The Right Way*, aunque sí una adaptación muy fiel a la idea del original pero con adaptaciones, modificaciones y agregados que, según mi experiencia e interés, dan al lector hispanohablante un texto de referencia no solo tan rico como lo es *PHP: The Right Way* para el angloparlante, sino también más fluído y fiel a nuestra forma de lectura y de entender las cosas por la idiosincracias de nuestra lengua. Si el lector examina y compara este texto con el original *PHP: The Right Way* notará que el presente **Expreso PHP y la vía correcta** difiere en el orden de algunos capítulos, en la explicación de muchos temas, y que es un poco más amplio que *PHP: The Right Way*. Mientras que *PHP: The Right Way* se enfoca en ser un *sitio de*

*consulta y referencia*, **Expreso PHP y la vía correcta** lo he gestado como un libro completo que además de fungir como consulta y referencia sirva también por sí mismo como texto de introducción a la programación en general y PHP en particular.

De hecho, *PHP: The Right Way* (y sus referencias) no han sido mis únicas fuentes, y he bebido también de otros textos, como *PHP 5 Power Programing*, *Eloquent JavaScript*, entre otros.

#### ¡Corré la voz!

La versión más reciente de **Expreso PHP y la vía correcta** está siempre disponible en línea en ReadTheDocs, si este texto te ha gustado y resulta útil, entonces que maravilla recibir tu apoyo ayudando a que otros desarrolladores de PHP (nuevos o no) puedan encontrar información más útil sobre este lenguaje, y querés ayudar aún más, te invito a adquirir copias (por un precio simbólico) en los formatos PDF, EPUB y/o MOBI vía Leanpub.

## 1.2 Introducción

PHP es un lenguaje interpretado, es decir, los programas escritos con PHP son ejecutados por un *intérprete*, dicho intérprete debe estar instalado siempre en la máquina dónde se espera ejecutar programas PHP, a diferencia de los lenguajes *compilados*, como C o Rust, en los cuales una vez compilado el programa para una plataforma, tal programa es capaz de ejecutarse por sí mismo.

El Grupo PHP (PHP Group), la comunidad abierta detrás del desarrollo del lenguaje, ha definido la especificación del lenguaje de tal manera que las diferentes *implementaciones* interpreten el mismo código PHP, aunque muchas veces las implementaciones introducen particularidades propias cambiando algunos aspectos respecto a la especificación.

A los *intérpretes* también se les llama **máquinas virtuales** o **motores** (*engines*), que son el resultado del desarrollo de una implementación concreta. Los motores más populares para PHP son el *Zend Engine*, desarrollado en *C* por Zend Technologies y el Equipo de PHP; y el HHVM, desarrollado en *C++* por Facebook[1].

El más popular de ambos es por mucho el *Zend Engine*, que se considera la implementación de facto de PHP. En adelante en este texto, *PHP* referirá tanto al lenguaje (su especificación) como al *Zend Engine* (su implementación), y por tanto las versiones dadas, por ejemplo *PHP 7.2*, hace referencia a *la implementación de PHP 7.2 en Zend Engine*, que a todas luces resulta tremendamente largo e impráctico de escribir, decir y citar, cuando con «PHP 7.2» basta para que quede claro.

### 1.2.1 Usá la última versión estable (7.2)

Si estás comenzando con PHP, empezá utilizando la última versión estable disposible: PHP 7.2. PHP 7.2 es la versión estable más reciente; a partir de PHP 7.0 se agregaron fabulosas características nuevas que mejoraron tremendamente el lenguaje respecto de la viejas versiones 5.x. El motor fue ampliamente reescrito, y PHP es ahora mucho más rápido que las versiones anteriores.

Es muy probable que se encuentre con PHP 5.x en uso, la última versión de 5.x fue la 5.6. No es una mala opción, pero deberías tratar de actualizar a la última versión estable tan pronto como podás (PHP 5.6 no recibirá actualizaciones de seguridad después de 2018). Actualizar es realmente fácil dado que no hay muchos cambios que no sean retrocompatibles. Si no estás seguro(a) de a que versión corresponde una funcionalidad o característica, podés verificar en la documentación de PHP.

[1] Aunque Facebook ha anunciado que HHVM v3.30 es la última versión en soportar PHP.

### 1.2.2 Servidor web interno

A partir de PHP 5.3, podés empezar a aprender PHP sin necesidad de instalar y configurar un servidor web de regla. Para iniciar el servidor interno, ejecutá el siguiente comando desde la terminal en la raíz de su proyecto web:

```
php -S localhost:8000
```

- Aprendé más sobre el servidor interno.

### 1.2.3 Instalación en Linux

Las Distribuciones Linux más populares incluyen alguna versión de PHP en sus repositorios por defecto, aunque, por lo general, tales versiones están un poco atrasadas respecto a la última versión estable.

#### Instalar PHP en Ubuntu (16.04) desde los repositorios regulares

```
sudo apt-get install php7.0-cli
```

#### Instalar PHP en Ubuntu (14.04, 16.04 y 18.04) vía ppa

```
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
sudo apt-get install php7.1-cli
```

#### Instalar PHP en Fedora

#### Para fedora 25

```
rpm -Uvh http://rpms.famillecollet.com/fedora/remi-release-25.rpm
dnf --enablerepo=remi --enablerepo=remi-php71 install php-cli
```

#### Para fedora 26

```
rpm -Uvh http://rpms.famillecollet.com/fedora/remi-release-26.rpm
dnf --enablerepo=remi --enablerepo=remi-php71 install php-cli
```

#### Instalación en Mac

Los sistemas OS X incluyen una versión de PHP preempaquetada que, por lo regular, se encuentra un poco atrasada respecto a la última versión estable. Mavericks, por ejemplo, tiene la 5.4.17, Yosemite la 5.5.9, El Capitán la 5.5.29 y Sierra la 5.6.24, pero con PHP 7.2 publicada ninguna de las anteriores es suficiente.

Existen varias vías para instalar PHP en sistemas OS X.

### Instalar PHP vía *Homebrew*

Homebrew es un potente gestor de paquetes para OS X, y puede ayudarte a instalar PHP y diversas extensiones fácilmente. Homebrew PHP es un repositorio que contiene «*formulae*» relacionado con PHP para *Homebrew*, y te permitirá instalar PHP.

En este punto, podés instalar `php53`, `php54`, `php55`, `php56`, `php70`, `php71` o `php72` utilizando el comando `brew install` y cambiar entre ellas modificando su variable `PATH`. Alternativamente padés usar brew-php-switcher para que cambie la versión automáticamente por vos.

### Instalar PHP vía *Macports*

A la fecha, padés instalar `php54`, `php55`, `php56`, `php70`, `php71` o `php72` utilizando el comando `port install`, por ejemplo:

```
sudo port install php56
sudo port install php72
```

Y luego podés ejecutar el comando `select` para cambiar la versión activa de PHP:

```
sudo port select --set php php72
```

### Instalar PHP vía *phpbrew*

phpbrew es una herramienta para instalar y manejar varias versiones de PHP. esto puede ser realmente útil si dos aplicaciones/proyectos necesitan diferentes versiones de PHP y no utilizás máquinas virtuales.

### Instaladores todo en uno

Las soluciones previas funcionan para PHP per se y no proveen cosas como Apache, Nginx o PostreSQL. Las soluciones «todo en uno» como MAMP y XAMPP instalarán todos esos bits de software por vos y lo mantendrán todo junto, pero esas instalaciones sencillas implican menor flexibilidad.

### Instalación en Windows

Podés descargar los binarios directamente de windows.php.net/download. Tras la extracción de PHP, se recomienda establecer el PATH a la raíz de la carpeta de PHP (allí dónde `php.exe` se encuentra) y así podés ejecutar PHP desde cualquier parte.

Para aprendizaje y desarrollo local, podés usar el servidor interno que se incluye desde PHP 5.4+, así que no necesitás preocuparte por configurar nada. Si prefirieras una solución «todo en uno» que incluya un servidor web de regla y un gestor de base de datos, entonces herramientas como Web Platform Installer, XAMPP, EasyPHP, OpenServer y WAMP ayudarán a preparar rápidamente entornos de desarrollo sobre Windows. Dicho esto, esas herramientas serán ligeramente diferente de los entornos de producción, así que tené cuidado de las diferencias de entornos si estás desarrollando en Windows y desplegando en Linux.

Si necesitás ejecutar un entorno de predución en Windows, entonces IIS7 te dará el mejor rendimiento y estabilidad. Podés utilizar phpmanager (un *plugin* de IU para IIS7) para configurar y gestionar PHP de manera sencilla. IIS7 provee *FastCGI* por defecto y listo para usar, sólo necesitás configurar PHP como manejador. Para soporte y recursos adicionales está disponible una sección dedicada en iis.net para PHP.

Generalmente ejecutar una aplicación en diferentes entornos para desarrollo y producción pueden llevar a errores extraños apareciendo en la vida real. Si estás desarrollando en Windows y desplegando en Linux (o cualquier otro sistema no Windows) entonces deberías considerar utilizar una Máquina Virtual.

Chris Tankersley tiene un artículo muy útil publicado en su bitácora en el cual expone que herramientas usa.

## 1.3 Aspectos del lenguaje

PHP es un lenguaje de programación cuya sintaxis es heredera de C, pero a diferencia de este último, PHP es un lenguaje de tipado dinámico, multiparadigma e interpretado. Esto convierte a PHP en un lenguaje tremendamente flexible y potente.

PHP ha tenido una historia vibrante desde que comenzó como un proyecto personal de Rasmus Lerdorf hasta la implementación actual del Zend Engine 3, pasando de ser un lenguaje usado para potenciar la página personal de Rasmus a ser el lenguaje que hoy potencia gran parte del Internet, desde Wikipedia y Wordpress, hasta los millones de foros, bitácoras y aplicaciones que hoy por hoy dependen de PHP para funcionar.

En poco se parece el PHP de hoy al PHP de antaño, su evolución (no sin tropiezos) ha sido producto de un esfuerzo colectivo que ha fructificado en una especificación formal, varias implementaciones y, finalmente, en un lenguaje que cumple con las demandas que la industria y los profesionales de hoy día ejercen.

En este capítulo se abordará todos los aspectos generales necesarios para conocer el lenguaje y comenzar a programar apropiadamente con PHP.

### 1.3.1 Paradigmas de programación

Se ha dotado a PHP con un soporte amplio de paradigmas de programación. Durante su draatica evolución se le agregaron destacables mejoras en el soporte de los mismos, tales como ser la orientación a objetos y funcional (gracias al soporte de funciones anónimas [*closures* {clausuras}]) en PHP 5.0 (2004); los *namespaces* (espacios de nombres) en PHP 5.3 (2009), y los *traits* (rasgos) en PHP 5.4 (2012).

#### Programación imperativa

La programación imperativa no es más que el uso de *sentencias* que van cambiando los estados de un programa. Es decir, esas sentencias le van *instruyendo* a la computadora que valor tiene un elemento en un momento dado. Cuando los conjuntos de sentencias se *empacan* en funciones se le llama *programación procedural*, y cuando se emplean *estructuras* para controlar el flujo de los procedimientos, en efecto, se habla de *programación estructurada*.

Este paradigma es tremendamente útil para automatizar tareas, como crear pequeños *scripts* que realizan cálculos sencillos o resuelven tareas repetitivas.

El típico ejemplo de «Hola mundo» es en toda regla un ejemplo de programación imperativa para PHP.

```
echo "¡Hola mundo!";
```

#### Programación orientada a objetos

El principio básico de la orientación a objetos es que los programas se comporten como una secuencia de *transformaciones* en los *estados* de sus *objetos* por medio de *mensajes*, ahora, la implementación de este paradigma requiere el soporte de un conjunto amplio de características, como clases, interfaces, herencia, constructores, clonación, excepciones, etc. PHP soporta bastante bien este paradigma, proveyendo no sólo un conjunto amplio de sus carácterísticas, sino también un rico set de clases básicas de gran utilidad cuando se desarrolla empleando esta técnica de programación.

Ahora, el paradigma orientado a objetos no se implementa igual en todos los lenguajes, por diversas razones, por ejemplo PHP no soporta la herencia múltiple, pero en cambio soporta la elegancia de los *traits* (rasgos).

- Leer más sobre PHP orientado a objetos
- Leer más sobre *traits* (rasgos)

#### Programación funcional

PHP soporta *funciones de orden superior*, esto significa que una función puede asignarse a una variable. Tanto las funciones definidas por el usuario como las funciones predefinidas por el lenguaje pueden referenciarse por medio de una variable y ser invocadas dinámicamente. Las funciones pueden pasarse como argumentos a otras funciones, y también pueden devolver otras funciones.

La recursividad, una característica que permite a una función llamarse a sí misma, está soportada por el lenguaje, pero la mayoría del código de PHP tiende a enfocarse en la iteración.

PHP 5.4 agregó la posibilidad de enlazar *closures* a el alcance de un objeto y también mejoró el soporte para invocables en tanto que pueden utilizarse intercambiablemente con funciones anónimas en la mayoría de los casos.

- Continuar la lectura en *Programación funcional con PHP*
- Leer sobre funciones anónimas
- Leer sobre la clase Closure
- Más detalles en el RFC de Closures
- Leer sobre invocables
- Leer sobre la invocación dinámica de funciones con call_user_func_array()

### 1.3.2 Metaprogramación

PHP soporta varias formas de *metaprogramación* a través de mecanismos como la API de reflexión y métodos mágicos. Existen muchos métodos mágicos disponibles como `__get()`, `__set()`, `__clone()`, `__toString()`, `__invoke()`, etc., que permiten a los desarrolladores inspeccionar el comportamiento de las clases. Algunos desarroladores de Ruby dicen que en PHP falta `method_missing` (ausencia de método), pero está disponible como `__call()` y `__callStatic()`.

- Leer sobre métodos mágicos
- Leer sobre reflexión
- Leer sobre sobrecarga

### 1.3.3 Interfaz de línea de comandos

PHP fue creado para escribir aplicaciones web, pero también es útil como lenguaje de *script* para programar aplicaciones de interfaz de línea de comandos (*command line interface [CLI]*). Los programas PHP para CLI pueden ayudar a automatizar tareas como pruebas, despliegue y administración de aplicaciones.

Los programas PHP para CLI son poderosos porque es posible utilizar directamente el código de una aplicación sin tener que crear y asegurar una GUI web para ella. Sólo hay que tener cuidado de *no* poner los *scripts* de PHP en el directorio raíz de acceso público.

Ejecútese desde la línea de comandos:

```
php -i
```

La opción `-i` mostrará en pantalla la configuración de PHP tal como una llamada a la función `phpinfo()`.

La opción `-a` provee una consola interactiva, similar a la IRM de Ruby o a la consola interactiva de Python. Existe un amplio conjunto de opciones de gran utilidad.

El siguiente ejemplo es un sencillo programa para CLI de «Hola, $name». Para hacerlo, crear un archivo y llamarle `hello.php`:

```
if ($argc !== 2) {
    echo "Ejecutado: php hello.php [nombre].\n";
    exit(1);
}
$name = $argv[1];
echo "Hola $name\n.";
```

PHP establece dos variables especiales basadas en los argumentos con los que se ejecuta el *script*: `$argc` es una variable de tipo entero que contiene el argumento de *conteo* y `$argv` es otra variable de tipo arreglo conteniendo cada *valor* del argumento. El primer argumento es siempre el nombre del archivo PHP, en este caso `hello.php`.

La expresión `exit()` se usa con número distinto de cero para indicarle a la consola que el comando ha fallado. Los códigos de salida comunmente usados pueden encontrarse aquí.

Para ejecutar el script, ejecutá desde la línea de comandos:

```
> php hello.php
Ejecutado: php hello.php [nombre]
> php hello.php mundo
Hola mundo.
```

- Leer sobre ejecutar PHP desde la línea de comandos
- Leer sobre la configuración en Windows para ejecutar PHP desde la línea de comandos

### 1.3.4 Sintaxis básica

Como se mencionó al principio de este capítulo, la sintaxis de PHP es principalmente heredera de la sintaxis de *C*, pero también bebe de PERL, Java y C++. No se asuste el(la) lector(a) por ello, la sintaxis de PHP es sencilla y de fácil aprendizaje.

#### Etiquetas

Todo código PHP se ubica entre etiquetas de apertura y cierre: `<?php ?>`. El código PHP puede embeberse junto al `html`, el intérprete ejecutará sólo aquel código que se encuentre entre las etiquetas PHP; no obstante es recomendable que el `html` y el código PHP no se mezclen en un mismo archivo, es preferible que los archivos PHP contengan sólo código PHP, la primer línea de estos archivos ha de ser la etiqueta de apertura `<?php` y la etiqueta de cierre se omite.

```
<?php
// La línea de arriba debe ser siempre la primera de todo archivo PHP
```

#### Comentarios

Los comentarios permiten escribir prosa explicativa que es útil a los programadores e irrelevante al intérprete, de modo que los ignora. Los comentarios cortos se prefijan con `//` y terminan con la línea.

Los comentarios largos o multilínea se encierran entre `\*` y `*/`, por ejemplo:

```
// Comentario corto
/*
Comentario largo,
este comentario puede contener mucho texto que explique el porqué
de las cosas que se están haciendo y puede componerse de muchas líneas.
 */
```

### Sentencias y bloques

Tal como en C, C++ o Java, en PHP cada sentencia debe terminar con un punto y coma «`;`», y cada bloque se delimita con llaves «`{}`», ejemplo:

```
// Las sentencias terminan en punto y coma:
echo "Las sentencias terminan en punto y coma";
$tres = 2 + 1;
echo $tres;

// Los bloques se delimitan en llaves
if(true) {
  // Cuepo del bloque
}

class LaClase {
  // Cuepo del bloque
}
```

Como puede observarse en el ejemplo anterior, tanto una estructura condicional como una clase representan bloques, muchas otras estructuras de PHP también son bloques y se verán más adelante.

### Tipos de dato

Un «tipo de dato» determina la semántica y límites de un valor cualesquiera que exista en el contexto de ejecución de todo programa. La especifiación de PHP categoriza los tipos soportados por PHP en dos grupos: **escalares** y **compuestos**.

### Tipos escalares

Los tipos escalares son tipos de *valor* o magnitud, su estructura es *atómica*[1] y todo escalar se comporta como contenedor exclusivo de su propio valor. PHP soporta cinco tipos escalares:

- `bool`
- `int` (números enteros)
- `float` (números de punto flotante)
- `string` (cadenas de texto)
- `null` (*valores* «nulos»)

Los tipos `integer` y `float` son también conocidos como «tipos aritméticos».

[1] Esto no es del todo cierto para las cadenas, las cadenas pueden *mutar* con las concatenaciones, no obstante su compartamiento es prácticamente atómico.

### El tipo *Boolean*

Abstrae los valores *booleanos*, se expresa como `bool` y `boolean` funge como sinónimo. Este tipo es capaz de contener exclusivamente uno de los dos valores booleanos: `true` y `false` (verdadero y falso).

### El tipo Entero

Abstrae la representación de los números enteros, se expresa como `int` y `integer` funge como sinónimo. Este tipo respeta sigo (positivo y negativo) y su capacidad de almacenamiento depende de la plataforma (32 o 64 bits); para 32 bits el rango posible ronda entre «**-2147483648** y **2147483647**» (unos dos mil millones) en tanto que para 64 bits el valor máximo es de alrededor de 9E18 (unos nueve quintillones).

Además de la notación decimal, este tipo también soporta la notación octal, hexadecimal y binaria.

### El tipo Flotante

Abstrae la representación de números reales, se expresa como `float`, en tanto que `double` y `real` fungen como sinónimos.

Algunas operaciones sobre enteros pueden producir resultados matemáticos que no se pueden representar como `int` y por tanto se representarán como `float`.

### El tipo Cadena

Una cadena es un conjunto continuo de *bytes* que representan una secuencia de cero o más caracteres.

Conceptualmente una cadena puede considerarse como un arreglo de bytes (de *elementos*) donde sus claves son valores enteros iniciando en el cero. No obstante una cadena **no** se considera una colección y no puede iterarse[2].

Se expresa como `string` y carece de sinónimos.

### El tipo Nulo

El tipo nulo puede almacenar exclusivamente un sólo valor: `NULL`. Este valor escalar representa la *nulidad*, es decir la ausencia de valor o magnitud. Este tipo es especial en tanto que no admite operaciones y su único fin es precisamente representar la nulidad. Se expresa como `null`.

### Tipos compuestos

Se denominan *compuestos* en tanto estos tipos *pueden contener* otras variables además de sí misma, por ejemplo un objeto contiene a sus propiedades y un arreglo contiene sus elementos.

Los objetos y *recursos* son tipos gestionados. El tipo contiene información (en un gestor) que encamina hacia el valor. PHP soporta los siguientes tipos compuestos:

- `array` (arreglos)
- `object` (objeto)
- `callable` (invocable)

[2] Mucho(a)s diseñadores de lenguajes definen las cadenas como escalares, entre otras razones, porque facilitan así su manipulación mediante los operadores y construcciones básicas del lenguaje, pero no es el caso para todos los lenguajes. En el caso de PHP, como se indica, se entienden las cadenas como un tipo escalar.

- `resource`

### El tipo Arreglo

Un arreglo se expresa como `array`, se trata de una estructura de datos que contiene una colección de cero o más elementos cuyos valores son accesibles mediante claves de tipo `int` o `string`.

### El tipo Objeto

Un objeto es una instancia de una clase. Cada declaración de clase define un nuevo tipo, y dicho tipo es también un *tipo objeto*.

### El tipo Invocable

Un invocable es un tipo que denota una función que puede invocarse dentro del contexto al que se provee como argumento.

### El tipo Recurso

Un recurso es un descriptor hacia una especie de entidad externa, por ejemplo archivos, bases de datos o *sockets* de red.

Los recursos son solamente creados y consumidos por la implementación, pero nunca son creados ni consumidos por código PHP.

- Leer más sobre los tipos de datos de PHP.

### Variables

Las variables son construcciones que almacenan datos y que son suceptibles de operaciones. En PHP las variables son de tipo dinámico, adoptan el tipo del valor que se les asigna, y si tras una asignación el nuevo valor es de un tipo diferente al anterior, la variable adopta ese nuevo tipo.

Todas las variables en PHP se prefijan con `$`. Las variables deben nombrarse *siempre* con letras o con el guión bajo, pero **nunca** pueden iniciar con un número.

```
// Variables válidas nombradas de tal forma que reflejan su tipo
$unEntero = 6;
$unaCadena = "Texto",
$unBoolean = false;

// Variable válida que inicia con guión bajo
$_unFlotante = 3.1416;

// Variable inválida, PHP lanzará un error si se declara una variable así
$1VariableInvalida = null;
```

Debido a que las variables pueden cambiar de tipo, resulta útil poder saber que tipo tiene una variable en un momento concreto, para ello PHP dipone de la función `gettype()`.

```
$unEntero = "6";
echo gettype($unEntero); // Imprime 'string'
```

Hay ocasiones en las que se desea forzar a que una variable sea de un tipo determinado, PHP cuenta con la función `settype()` para establecer un tipo concreto a una variable.

```
// El tipo de
$unEntero = "6";
settype($unEntero, 'integer');
echo gettype($unEntero); // Imprime 'integer'
```

- Leer más sobre las variables.
- Leer más sobre gettype.
- Leer más sobre settype().

### Ámbito o alcance

**Ámbito** o **alcance** refieren a la accesibilidad de una variable dentro de un contexto. Si una variable se declara a nivel de un archivo será accesible para ese archivo y los archivo que incluya, incluyendo los bloques de estructuras de control y **exceptuando** para las **funciones** y **clases** dentro de dicho archivo.

```
<?php
$uno = 1;

include "extrafile.php";
```

Así, dentro del archivo `extrafile.php` la variable `$uno` es accesible. No obstante, los bloques implementan un *alcance local*, es decir, sus variables son accesibles únicamente dentro de su delimitación de bloque.

```
<?php
$uno = 1; // Variable con alcance de nivel de archivo

if(true) {
  $tres = 3;
}

function local() {
  $dos = 2; // Variable con alcance local
  echo $dos;
}

echo $dos; // Esto es un error porque $b sólo existe para el bloque de la función.
echo $tres; // Imprimirá 3
local(); // Imprimirá 2
```

Es necesario tener presente esas diferencias en la accesibilidad de las variables. Por ejemplo, el hecho de que *$tres* sea accesible fuera del bloque *if* tras su definición. No obstante lo anterior, por legibilidad, es conveniente «declarar» una variable previo a su redefinición dentro de un bloque.

```
<?php
$tres = null; // Declarada la variable con un valor nulo para su posterior uso.

if(true) {
  $tres = 3; // Redefinida la variable con un valor determinado
}

echo $tres; // Imprime 3
```

### Operadores

Los operadores son construcciones que permiten *transformar* el valor de variables y de expresiones. Son implementaciones de los operadores matemáticos, pero con ciertas particularidades.

Como en matemáticas, los operadores responden a una precedencia, así un operador de multiplicación (`*`) precede a un operador de adición (`+`). Asímismo es posible manipular la precedencia mediante paréntesis: `$nueve = (1 + 2) * 3;`

Los operadores también responden a una *asociatividad* que determina su orden de evaluación, de tal modo que cuando se presentan operadores de igual precedencia, su orden de evaluación se ejecuta en función de su asociatividad.

### Operadores aritméticos

Son operadores **binarios** correspondientes a la arimética básica:

- `+` Adición (o suma)
- `-` Sustracción (o resta)
- `*` Producto (o multiplicación)
- `/` Cociente (o división)

Los cuatro operadores anteriormente listados tienen una asociatividad por la **izquierda**. El operador `**` **se utiliza para la exponenciación y se asocia por la derecha**: `$a ** $n` significa elevar `$a` a la enésima potencia.

### Operadores de asignación

Todos los operadores de asignación son **binarios** (es decir, operan sobre dos elementos) y tienen una asociatividad por la **derecha**. Sirven para establecer en el elemento de la izquierda el valor del elemento de la derecha. El operador de asignación por antonomasia es `=`; pero no es el único. Otros operadores de asignación son: `+=`, `-=`, `*=`, `/=`, y sirven como abreviaciones de operaciones entre operadores aritméticos y de asignación. Un ejemplo lo dejará más claro:

```
<?php
$n = 6;
// Para sumar el propio $n a $n:
$n = $n + $n; // $n = 12

// Puede abreviarse como
$n += $n;
```

### Operadores de comparación

Son operadores binarios que permiten evaluar la semejanza de dos elementos. Estos operadores no responden a ninguna asociatividad.

- `$n == $m` Compara si el valor de `$n` es igual al de `$m`.
- `$n != $m` Compara si el valor de `$n` es diferente de `$m`.
- `$n > $m` Compara si el valor de `$n` es mayor que el `$m`.
- `$n >= $m` Compara si el valor de `$n` es mayor o igual que el `$m`.
- `$n < $m` Compara si el valor de `$n` es menor que el `$m`.
- `$n <= $m` Compara si el valor de `$n` es menor o igual que el `$m`.

Los primeros dos operadores de comparación (== y !=) no comparan el tipo, tal que `6 == '6'` devolverá `true`, y esto puede dar paso a potenciales errores, por ello es recomendable el uso de los operadores de comparación idéntica:

- `$n === $m` Compara si el valor de `$n` es idéntico al de `$m`.
- `$n !== $m` Compara si el valor de `$n` es no idéntico de `$m`.

El resultado de la comparación es siempre un valor *booleano*, y este puede utilizarse tanto en expresiones como asignarse a variables. Algunos ejemplos ayudarán a aclarar mejor estos conceptos:

```
<?php
$n = 6;
$m = '6'

// Comparación de igualdad
echo $n == $m; // true

// Cambiar el valor y comparar de nuevo
$m = 9;
echo $n == $m //  false

$m = '6';
echo $n != $m // $r = false
echo $n !== $m // $r = true

// Asignar el resultado de una comparación e imprimirlo
$m = 9;
$r = ($n > $m)// $r = false;
echo $r;
```

### Operadores lógicos

Los operadores lógicos son operadores **binarios** que se asocian por la **izquierda** y sirven para evaluar expresiones booleanas.

- `$a && $b` (*AND* [Y]) Devolverá `true` sí y solo sí `$a` y `$b` son `true`.
- `$a || $b` (*OR* [O]) Devolverá `true` sí cualquiera de `$a` o `$b` son `true`.
- `$a xor $b` (*XOR* [O exclusivo]) Devolverá `true` sí `$a` o `$b` son `true`, pero no ambos.

### Otros operadores

Se han visto la mayoría de operadores utilizados en PHP, y aunque no se han listado todos, los vistos hasta ahora son suficientes para la mayoría de operaciones en el día a día habitual del desarrollo con PHP.

- Leer más sobre los operadores de PHP.

### Estructuras de control

Las estructuras de control son fundamentales para encaminar el flujo de ejecución de las instrucciones de un programa. Básicamente, toda estructura de control *evalúa* una *expresión* y a partir del resultado de dicha evaluación, determina que se ejecuta a continuación.

Se distinguen dos tipos básicos de estructuras de control: *condicionales* y *ciclos*.

### Condiciones

Sirven para determinar, en función de una *condición*, que se ejecuta y que no. Las dos estructuras básicas de este tipo son `if (<expresión>)` y `switch (<expresión>)`.

A la estrucura `if (<expresión>)` puede agregarse las construcciones `else(<expresión>)` y/o `else if (<expresión>)`.

La condición `if (<expresión>)` evaluará la `<expresión>` y si esta resulta `true` ejecutará la sentencia o bloque que le sigue. Si se ha definido `else(<expresión>)`,y el resultado de la `<expresión>` es `false`, entonces se ejecuta la sentencia o bloque que le sigue. Si se ha definido `else if (<expresión>)`, y la evaluación del primer `if (<expresión>)` es `false` entonces se evaluará la nueva `<expresión>` correspondiente al `else if (<expresión>)` y su sentencia o bloque se ejecutará sólo si el resultado de dicha expresión es `true`.

Para la condición `switch (<expresión>)` se define una serie de `case` (casos), si la `<expresión>` concuerda con un caso, se comenzará a ejecutar a partir de dicha coincidencia hasta el fin de la estructura o cuando se encuentre un `brake` (quiebre). Al `switch (<expresión>)` se le puede anidar un `default` como un caso especial que se ejecuratá cuando ninguno de los otros casos coincida con la expresión.

Ejemplos a continuación:

```
<?php
$verdadero = true;

// La siguiente sentencia se ejecutará siempre
if($verdadero) echo 'verdadero';

// La siguiente sentencia no se ejecutará nunca
if($verdadero === false) echo 'false';

// Del siguiente ejemplo se ejecutará el bloque del else
if($verdadero === false) {
  echo 'nunca entra aquí';
  // Recordando que un bloque se delimita por llaves «{}»
}
else {
  echo 'entra en el else';
  // Cambiar el valor de la expresión
  $verdadero = false;
}

// A continuación se evalúan ambas expresiones,
// pero no entra en ninguna.
if($verdadero) {
  // nunca entra aquí
}
else if($verdadero === 6) {
  // tampoco entra aquí
}

switch ($verdadero) {
  case true:
    echo 'nunca entra aquí';
    break;
  case false:
    echo 'el valor de $verdadero es false\n';
    $verdaredo = true;
    echo 'ahora el valor de $verdadero es true';
```

(continué en la próxima página)

(proviene de la página anterior)

```
    break;
  case 6:
    echo 'nunca entra aquí';
    break;
  default:
    echo 'por defecto tampoco entrará aquí';
}
```

Comprender todas las combinaciones posibles para una estructura de control puede llevar algo de práctica, pero una vez conseguido es posible obtener un control eficiente y un código elegante.

### Ciclos

Los ciclos son estructuras que sirven para controlar la repetición de una sentencia o bloque. Mucha veces es necesario recorrer elementos y operar uno por uno, o bien, repetir una funcionalidad un determinado número de veces.

Los ciclos son semejantes a las condiciones en tanto que también necesitan de una expresión que se evalúa y en función de la cual se determina si el ciclo se ejecuta o no. Tal expresión, por lo general, se modifica con cada iteración del ciclo o bien, se altera una *bandera* para indicar al ciclo que debe dejar de ejecutarse.

PHP soporta cuastro estructuras para ciclos: `while`, `do while`, `for` y `foreach`.

#### while

En español significa «mientras», y es el tipo de ciclo máo básico. Este ciclo se itera *mientras* la condición de la expresión sea verdadera, por lo general la condición se controla con un *contador*:

```
<?php
$contador = 0;

while ($contador < 6) {
  echo $contador . "\n";
  $contador += 1;
}
```

#### do while

Esta estructura es similar a `while`, pero se ejecuta al menos una vez. Si la condición de `while` resulta `false` desde un principio, el ciclo **no** se ejecuta, en tanto que `do while` sí que lo hará al menos una.

```
<?php
$contador = 0;

// Nunca entrará debido a que el contador vale 0
while ($contador > 1 && $contador < 6) {
  echo $contador . "\n";
  $contador += 1;
}

// Se ejecutará al menos una vez
do {
  echo $contador . "\n";
```

(continué en la próxima página)

(proviene de la página anterior)

```
  $contador += 1;
} while ($contador < 6)
```

### for

Es el ciclo por excelencia para iterar sobre contadores. Es parecido a `while`, pero con una estructura que permite declarar el contador y la condición de iteración en una sóla línea, lo que favorece la construcción de cliclos elegates.

```
<?php
for($i = 0; $i < 6; $i++) {
  echo $contador . "\n";
}
```

La expresión `$i++` utiliza el operador unitario `++`, que sirve como una abreviatura de `$i += 1;`

Muchas veces el for se utiliza para recorrer arreglos u objetos y por tanto la condición de iteración está dada por el tamaño del arreglo:

```
<?php
$birras = ['rubia', 'mestiza', 'morena'];
$limite = count($birras);

for($i = 0; $i < $limite; $i++) {
  echo 'Que buena es la cerveza ' . $birras[$i] . "\n";
}
```

### foreach

Como se mencionó antes, los arreglos son tipos iterables, tipos que se comportan bien como vectores (arreglos indexados) y como diccionarios (arreglos de pares clave y valor). El ciclo `foreach` permite iterar sin necesidad de contadores sobre cualquier elemento que implemente la interfaz `Iterable`.

```
<?php
$birras = ['rubia', 'mestiza', 'morena'];
$cervezas = [
  'rubias' => ['artesanal', 'nacional', 'extranjera'],
  'mestizas' => ['artesanal', 'nacional'],
  'morena' => 'artesanal'
];

foreach($birras as $birra) {
  echo $birra;
}

foreach($cervezas as $clave => $alor) {
  echo $clave . ' ' . $valor;
}
```

## Funciones

Las funciones en PHP son herederas de las *funciones matemáticas*, son construcciones que esperan una *entrada*, realizan un *proceso* y devuelven una *salida*. Es decir, *empaquetan* una *tarea* o *rutina*.

Valga decir que *no todas* las funciones operan bajo el principio de *entrada → proceso → salida*; existen funciones que operan directamente sobre los argumentos de entrada y no devuelven nada, y otras que no esperan argumentos pero sí que retornan sus salidas. No obstante, la gran mayoría se ciñen al esquema básico descrito.

PHP dispone de un amplio número de funciones predefinidas y también permite la construcción de nuevas funciones por parte de lo(a)s programadores.

Para definir funciones en PHP se emplea la sentecia `function`, que antecede al *nombre de la función*, y tras el cual se definen (de ser necesario) entre paréntesis los parámetros que esperará como argumentos la función y, finalmente, el cuerpo de la función que siempre será un bloque entre llaves.

A partir de PHP 5.x se soporta declarar el tipo de los parámetros, y desde PHP 7.0 se soporta definir el tipo de retorno de la función mediante `: <tipo>` donde `<tipo>` puede ser cualquier tipo primitivo soportado por PHP o bien, cualquier tipo definido por el(la) desarrollar(a) como clase o interfaz.

Para invocar una función simplemente se emplea su nombre seguido de los argumentos correspondientes entre paréntesis.

```
<?php

function sumarEnteros(int $n1, int $n2) : int {
    return $n1 + $n2;
}

echo sumarEnteros(2, 6); // → 8
```

### Parámetros y argumentos

Como se indicó en el apartado anterior, a las funciones es posible asignarles una lista opcional de parámetros. Cada parámetro se separa por coma y para cada parámetro es posible declarar un tipo (primitivo o definido) a fin de que los argumentos proveídos estén restringidos a dicho tipo.

```
<?php

function mostrarTipos(int $n, string $str, bool $tof, $any) {
  echo gettype($n);
  echo gettype($str);
  echo gettype($tof);
  echo gettype($any);
}

mostrarTipos(6, "birra", false, new \stdClass());
mostrarTipos("6", "birra", false, new \stdClass()); // lanzará error
```

En ocasiones para una función se podría esperar una lista arbitraria de parámetros a fin de que durante la llamada se establezcan, bien una lista separada por comas, bien bien una lista en un arreglo, un conjunto cuantitativamente arbitrario de argumentos. A partir de PHP 5.6 es posible declarar que se espera dicho tipo de listas mediante el *token* `...` como prefijo al parámetro que contendrá el listado.

```
<?php

function sumar(int ...$sumandos) : int {
  $r = 0;
  foreach($sumandos as $s) {
      $r += $s;
  }
```

(continué en la próxima página)

(proviene de la página anterior)

```
  return $r;
}

echo sumar(1, 2, 3, 4); // → 10
```

### El depurador *Xdebug*

Una de las herramientas más útiles en el desarrollo de software es un buen depurador. Permite trazar el flujo de ejecución del código y monitorear los contenidos de la pila de ejecución. *Xdebug*, el depurador de PHP, puede ser utilizado por varios IDE's para proveer puntos de quiebre y pilas de inspección. También permite que herramientas como *PHPUnit* y *KCacheGrind* realicen un análisis y seguimiento del código.

Si un desarrollador se encuentra en un punto ciego, devolviendo cosas con `var_dump()`/`print_r()`, y aun así no puede encontrar la solución, quizá lo que necesite sea un depurador.

Instalar Xdebug puede ser engorroso, pero una de sus características más importantes es «la depuración remota», si tu desarrollo es local y probado en una máquina virtual o en otro serwidor, la depuración remota es la característica que querrás tener habilitada.

Tradicionalmente, en Apache es necesario modificar el *VHost* o el archivo `.htacces` con los siguientes valores:

```
php_value xdebug.remote_host 192.168.?.?
php_value xdebug.remote_port 9000
```

Las propiedades `remote_host` y `remote_port` corresponden con la computadora local y el puerto que se ha de configurar en un IDE para que lo escuche. Entonces se vuelve sólo cuestión de configurar el IDE en modo de «escucha de conexiones» y cargar la URL:

```
http://tu-proyecto.ejemplo.dev/index.php?XDEBUG_SESSION_START=1
```

El IDE entonces comenzará a interceptar el estado actual del *script* en ejecución, permitiendo a los desarrolladores establecer puntos de quiebre y evaluar los valores en memoria.

Los depuradores gráficos facilitan dar seguimiento al código, inspecionar variables, y evaluar el código contra la ejecución en vivo. Muchos IDEs tienen preempaquetados o soportado mediante extensiones la depuración gráfica con Xdebug.

- Leer más sobre Xdebug

## 1.4 Programación orientada a objetos

Desde que con PHP 5.0 se agregara el soporte inicial al paradigma orientado objetos, en las sucesivas versiones se ha ido enriqueciendo y mejorando este soporte. Hoy día este paradigma es el paradigma dominante a la hora de desarrollar con PHP. Muchísimas bibliotecas, *frameworks* y proyectos emplean este paradigma; por tanto, a todas luces, conocer la programación orientada a objetos con PHP es fundamental. Hoy por hoy, PHP soporta bastante bien este paradigma, proveyendo no sólo un conjunto amplio de sus características y principios, sino también un rico conjunto de clases básicas de gran utilidad cuando se desarrolla empleando esta técnica de programación.

### 1.4.1 Síntesis

El paradigma orientado a objetos se fundamenta en la abstracción semántica con el fin de *representar* un sistema de cosas (objetos), sean concretas o abstractas. La idea es permitir una ingeniería en el que lo(a)s desarrolladores puedan

programar sistemas de una forma *más natural* respecto de otros paradigmas.

En dicho sistema de objetos, estos sufren una secuencia de *transformaciones* en *sus estados* por medio de *mensajes* que se *envían* mediante *interfaces* unos a otros. Para poder implementar este paradigma los lenguajes deben soportar una serie de estructuras y relaciones que permitan la construcción de (básicamente) clases e instancias. No obstante, el paradigna orientado a objeto es más que sólo tales clases e instancias, también implica el soporte de interfaces, herencia, alcance, constructores, clonación, excepciones, etc.

El paradigma de la programación orientada a objetos se sustenta en cuatro principios fundamentales:

- Abstracción
- Encapsulamiento
- Composición
- Generalización

#### Abtracción

La *abstraciión* en una de las metodologías más usuales para abordar la complejidad. Se basa en la idea de *descomponer* o *simplificar* un concepto con el fin de enfatizar los elementos de interés y descartar el resto, por lo general, dentro de los límites establecidos por un *contexto*.

En la ingeniería informática se pretende seguir el *principio de mínima sorpresa*, que implica capturar aquellos atributos y comportamientos que no *deberían* generar sorpresas o ambigüedades dentro de un contexto dado. La idea es prevenir la implementación de características irrelevantes dentro de la abstracción y garantizar que la abstracción tenga sentido para el propósito del contexto.

El constructo central objeto de la abstracción es la **clase**. Cuanto producto de la abstracción se determinan los elementos escenciales de un concepto, dichos elementos son empleados para definir una clase. Luego cualquier objeto creado de dicha clase representa una instancia del concepto abstraído, así como (usualmente) ciertas características particulares.

So observará que los conceptos abstraídos en clases tienden a representar «*categorías*» y que plantean una úerte de «*declaración de intenciones*», una especie de *plantilla* o *estereotipo*, en tanto que *declara como* puede ser una entidad y *como* puede comportarse, pero **no** define entidad alguna.

También puede entenderse una clase como una declaración particular de tipo de datos. Ya se han visto los tipos primitivos de PHP, por ejemplo los enteros (`int`). Dicho tipo declara como es un entero cualesquiera, como se opera y cuales son las limitaciones que le atañen, pero no define ningún entero en particular. Luego, al observar las limitaciones de los tipos primitivos se manifiesta evidente que difícilmente pueden representar por sí mismos abstracciones más complejas, por ejemplo un *Gato* o un *Planeta*, y por una buena razón: no todos los sistemas necesitan representar gatos o planetas, o cualquier otro sustantivo que al(a la) lector(a) le venga a la mente. Y esto precisamente es lo que permiten las clases, declarar esas estructuras específicas, esos tipos de datos particulares para el sistema que se está atendiendo.

Las clases *tienden* a representar **sustantivos**, cuando se realiza un análisis de requerimientos, los sustantivos y/o categorías identificadas son potenciales clases del sistema. A partir de allí, declarar una clase no es algo difícil.

### 1.4.2 Clases y objetos

#### Declaración

Para declarar una clase, se utiliza la sentencia «`final`» `class NombreClase {}`, donde «`final`» es un identificador opcional que determina que la clase **no** es *heredable* (más adelante se ahonda en este tema); luego el identificador `class` que es necesario para declarar que la estructura a continuación **es** una clase; y finalmente `NombreClase` representa un nombre arbitrario con el que se designa la clase en cuestión. Es usual que los nombres de las clases sean sustantivos, ya que su valor semántico brinda información inferible sobre la misma. También nótese como el nombre

de la clase inicia con mayúscula, esto es una convención, pero una convención ampliamente extendida: los nombres de clases siempre deben comenzar con mayúscula[1].

```
// declaración de una clase
class NombreClase {
    // Propiedades
    // ...
    // Métodos
}
```

Se ha mencionado a *Gato* y *Planeta* como abstracciones complejas que pueden representarse como clases. Como puede verse, ambos nombres son sustantivos y cada uno expone una categoría, así como ninguno define ningún gato ni planeta en concreto.

```
// declarar la clase Gato
class Gato {
    // Propiedades
    // ...
    // Métodos
}

// declarar la clase Planeta
class Planeta {
    // Propiedades
    // ...
    // Métodos
}
```

#### Objetos

Por **objeto** ha de entenderse la instancia concreta de una clase. Luego, de una clase pueden existir muchos objetos. Es decir, estos se componen como entidades particulares de una clase; por ejemplo (es sencillo observar que) «Félix» es un gato específico del tipo `Gato`, y que «Marte» y «Venus» son planetas particulares del tipo `Planeta`.

Para instar un objeto se utiliza el operador `new` y una sentencia de asignación:

```
$objetoUno = new NombreClase();
$objetoDos = new NombreClase();

// Siguiendo los ejemplos de Gato y Planeta:
$felix = new Gato();
$venus = new Planeta();
$marte = new Planeta();
```

### 1.4.3 Miembros

Los elementos que caracterizan a una clase y determinan ese *como puede ser* Y *como se comportan* radican en dos construcciones: **propiedades** y **métodos**. A estos dos elementos se les conoce como **miembros**.

Toda clase debe componerse por propiedades y métodos; estos son los que le dan verdadero significado a una clase, los nombres de clase son arbitrarios y tienen poco más que valor semántico (fundamental, sea dicho, para una buena estructuración), pero son sus miembros los que le aportan identidad. Es fácil intuir que un gato carece momento angular y movimiento de traslación, y que un planeta no tiene patas ni maúlla; la identidad de estas instancias emerge de sus atributos y comportamientos: los gatos tienen patas y maúllan, los planetas tienen un diámetro y rotan.

[1] para más detalles sobre esta y otras convenciones, consultar el capítulo de «Guía de estilo»

Al conjunto de miembros *expuestos* por una clase se le llama *interfaz de la clase*; se ha utilizado el adjetivo «expuesto» en tanto que cada miembro es suceptible de un determinado «*ámbito*» o «*alcance*».

### Ámbito o alcance

Se declara un ámbito o alcance por miembro, tal alcance determina la restricción de *acceso* o *visibilidad* del miembro respecto de otras clases. Naturalmente, todos los miembros de una clase son accesibles para sí misma, empero el cumplimiendo del principio de encapsulamiento demanda que determinados miembros de la interfaz **no** sean accesibles para otras clases, y otros miembros sí. A la interfaz accesible por otras clases se le llama *interfaz pública*. Para restringir el acceso a los miembros se dispone de tres niveles de alcance:

- Público (`public`)
- Protegido (`protected`)
- Privado (`private`)

---

**Nota:** Si no se declara alcance para un miembro, entonces tal se asume como públicos.

---

### Alcance público

Establece que el miembro es accesible, además de la propia clase, por cualquier otra clase tenga acceso a la clase en cuestión.

### Alcance protegido

Establece que el miembro es accesible solo para la propia clase y clases derivadas, pero **no** de otras clases e instancias aunque tengan acceso a la clase en cuestión.

### Alcance privado

Establece que el miembro es accesible exclusivamente por la propia clase.

### Propiedades

Las **propiedades** son los elementos que *describen* **como** *es* o como se *compone* una clase, es decir, representan un **atributo** de sus objetos.

Las propiedades, al final, no son más que variables asociadas al alcance de una clase, y por tanto comparten las características de las variables; dicho de otra forma: son de tipo dinámico (adoptan el tipo de dato del valor) y se nombran con las mismas reglas que una variable.

Para declarar propiedades se emplea la sentencia «`alcance` `$nombrePropiedad` « `= defecto`»`;` listadas a nivel de estructura de una clase, donde «`alcance`» corresponde a uno de los tres niveles de alcance anteriormente expuestos (recordando que de omitirlo, por defecto se declara como `public`), `$nombrePropiedad` representa un nombre arbitrario que identifique a la propiedad y, finalmente, « `= defecto`» una asignación opcional para inicializar la propiedad con un valor determinado:

```
class NombreClase {
    public $nombrePrapiedadA = null;
    protected $_nombrePropiedadB = 'B';
    private $__nombrePropiedadC = ['C'];
    // ...
    // Métodos
}
```

El siguiente ejemplo ilustra mejor las propiedades de Gatos y Planetas:

```
class Gato {
    private $__colorDeOjos;
    private $__nombre;
    private $__nacimiento;
}

class Planeta {
    private $__nombre;
    private $__diametro;
}
```

### Métodos

Los métodos son los elementos que definen el **comportamiento** de un objeto, es decir, que *hacen*, como *operan* y como *funcionan*. Semejante a las propiedades, en el caso de los métodos estos no son más que **funciones** asociadas al alcance de una clase, y de igual manera comparten las reglas de declaración de cualquier función de PHP.

Para declarar un método se utiliza la sentencia `«alcance» nombreMetodo(«parámetros») « : tipoRetorno» {}` donde el alcance aplica igual que para las propiedades y el nombre del método también es un nombre arbitrario que identifique al método, luego, entre paréntesis se puede declarar una lista opcional de parámetros y, finalmente, un tipo de retorno opcional.

```
class NombreClase {
    public $nombrePrapiedadA = null;
    protected $_nombrePropiedadB = 'B';

    // ...
    public function doSomething() : bool {
      // ...
    }
}
```

Un *comportamiento* típico de los gatos, por ejemplo, es maullar; y de los planetas, rotar:

```
class Gato {
    // Propiedades
    // ...

    public function maullar() {
        echo "¡Miau!";
    }
}

class Planeta {
    // Propiedades
    // ...
```

(continué en la próxima página)

(proviene de la página anterior)

```
    public function rotar($grados) {
      echo "Rotando $grados grados";
    }
}
```

### 1.4.4 Encapsulamiento

Los niveles de alcance permiten el cumplimiento del principio de encapsulamiento. Este principio pretende aislar el estado o datos de un objeto y ocultar los *detalles de la implementación*, proveyendo el acceso mediante *interfaces*. La interfaz de una clase no es más que el conjunto de miembros accesibles de dicha clase. Así, existe una *interfaz pública* compuesta por todos los miembros cuyo alcance es público, y lo mismo para los miembros protegidos y privados.

Al *envolver* (y con ello, «*aislar*» de cierta forma) la implementación se procura mantener el principio de responsabilidad mínima, es decir, que una pieza de funcionalidad sea responsable de gestionar nada más que su propio estado, o de ejecutar una sola tarea. Si se cambia el mecanismo de una funcionalidad (por ejemplo, por una versión más eficiente) pero se mantiene la misma interfaz, como consecuencia se obtiene una mayor *transparencia*, en tanto que los usuarios de dicha interfaz no deben realizar ningún cambio y aun así se ven beneficiados por la mejora en la funcionalidad que consumen.

Un ejemplo típico de la implementación del encapsulamiento son los *establecedores* (*setters*) y *recuperadores* (*getters*), mediante los cuales se pueden garantizar ciertos comportamientos, como el tipo de un dato, la estructura o patrón de una cadena, límites numéricos, etc. Ejemplos:

```
class Gato {
  // Propiedades
  // ...

  /**
   * Garantiza que solo se establecezca uno de los tres colores posibles
   */
  public function setColorDeOjos(string $color) {
    if(in_array($color, ['negro', 'verde', 'amarillo'])) $this->__colorDeOjos =
↪$color;
    else echo "Color de ojos inválido";
  }

  public function getColorDeOjos() : string {
    return $this->__colorDeOjos;
  }

  /**
   * Comprueba el tamaño de la cadena, y si es mayor a 65 lo trunca.
   */
  public function setNombre(string $nombre) {
    if(strlen($nombre) > 65) $nombre = substr($nombre, 0, 65);
    $this->__nombre = $nombre;
  }

  public function getNombre() : string {
    return $this->__nombre;
  }

  /**
   * Garantiza que la fecha se establezca como un tipo \DateTime
```

(continué en la próxima página)

(proviene de la página anterior)

```
   */
  public function setFechaNacimiento(\DateTime $nacimiento) {
    $this->__nacimiento = $nacimiento;
  }

  public function getFechaNcimiento() : \DateTime {
    return $this->__nacimiento;
  }
}

class Planeta {
    // Propiedades
    // ...

    /**
     * Comprueba el tamaño de la cadena, y si es mayor a 255 caracteres lo trunca.
     */
    public function setNombre(string $nombre) {
      if(strlen($nombre) > 255) $nombre = substr($nombre, 0, 255);
      $this->__nombre = $nombre;
    }

    public function getNombre() : string {
      return $this->__nombre;
    }

    /**
     * Establece el diámetro a partir del radio.
     */
    public function setDiametro(float $radio) {
      $this->__radio = $radio;
      $this->__diametro = 2 * $this->__radio;
    }

    public function getDiametro() : float {
      return $this->__diametro;
    }
}
```

El código anterior puede parecer al principio un poco extenso, pero es bastante sencillo. El método *setColorDeOjos()* de la clase *Gato* garantiza no sólo que el parámetro proveido sea una cadena, sino que además verifica que sea una de tres posibles. Luego, si se necesita, por ejemmplo, soportar más colores de ojos, se podrían agregrar a la sentencia de verificación, manteniendo la interfaz pero ampliando la funcionalidad.

Por otra parte, el método *setDiametro()* espera un valor de radio como parámetro y en su implementación calcula el diámetro; y similar a lo dicho anteriormente, es posible ampliar la funcionalidad del método sin romper la compatibilidad mientras se mantenga la interfaz.

Puede notarse que para los métodos *get* se ha indicado un tipo de retorno. En PHP, como ya se dijo en el apartado de funciones, los tipos de retorno son opcionales, pero su uso permite mantener un código mucho más cohesionado y unas interfaces más estables, lo que facilita el mantenimiento del propio código.

Dicho lo anterior, no todos los método tienen que ser del tipo *set* o *get*, por ejemplo, los planetas rotan:

```
  class Planeta {
    // Propiedades
    // ...
```

(continué en la próxima página)

(proviene de la página anterior)

```
    public function rotar(int $grados) {
        if($grados < 1 || $grados > 360) echo "Grados de rotación fuera de rango.";
        else echo "Rotando $grados"."° grados";
    }

    // otros métodos...
}
```

El método *rotar()* no cambia ningún estado, sólo imprime el dato en pantalla, pero verifica que los grados proveidos correspondan a un valor entre 1° y 360°, que puede considerarse válido.

Es importante que exista un análisis y diseño previo a la redacción de las clases, a fin de que el diseño provea de una abstración lo bastante amplia y permita un desarrollo más sencillo del código. Esto, por supuesto, no es siempre posible, pero del diseño se pueden obtener al menos unas interfaces lo más cercanas posibles a la implementación necesaria, y gracias al encapsulamiento, si se respetan las interfaces, los cambios dentro de los métodos pueden ser incrementales y mantener la funcionalidad del código a lo largo del desarrollo.

### 1.4.5 Interfaces

Una clase es una definición de tipo, que se compone de un conjunto de miembros y que expone una *interfaz* en tanto *oculta* la implementación de su funcionalidad. El principio de ocultación plantea que

Existen escenarios en los que diferentes clases exponen una misma interfaz, pero en efecto se tratan de **tipos diferentes** y por eso son **clases distintas**, no obstante es posible deducir que pertecen a un sólo *tipo abstracto*. Interesa muchas veces poder garantizar un *contrato* de que una clase implemente una interfaz concreta, tal contrato es posible gracias a las Interfaces. En PHP las Interfaces se declaran mediante la sentencia `interface NombreInterfaz {}` donde `interface` es el identificador de declaración y `NombreInterfaz` es un nombre arbitrario que la identifica.

> **Advertencia:** Es importante **no** confundir *interfaz* como el concepto de los «miembros expuestos de una clase» con *Interfaz*, el concepto de la construcción del lenguaje que se expone mediante el identificador `interface` y permite la declaración de estas construcciones.

Una Interfaz es una suerte de «declaración de intenciones». Se compone de definiciones de constantes y declaraciones abstractas de **métodos públicos**; se le llama «abstracta» porque todos los métodos de una Interfaz carecen de implementación, como «declaración de intenciones» tiene por finalidad servir como ese *contrato* que debe cumplir una clase.

Por ejemplo, un Gato es un ser vivo, e interesa abstraer ciertos comportamientos de cualquier ser vivo, como caminar, comer o emitir sonidos; o bien un planeta es una forma muy concreta de *esfera* que tiene la capacidad de rotar o de devolver las cantidades de sus magnitudes, como el diámetro o es radio. Así, podrían definirse las siguientes interfaces:

```
interface SerVivo {
    public function caminar($distancia);
    public function emitirSonido();
}

interface Esfera {
    public function rotar(int $grados);
    public function getDiametro();
}
```

Posteriormente, en la sentencia de declaración de clase, mediante el identificador `implements`, se *establece el contrato*, y con ello la *obligación* de definir todas las implementaciones que la Interfaz declara.

```php
class Gato implements SerVivo {
    // propiedades
    // ...

    public function caminar($distancia) {
        echo "Avanza $distancia";
    }

    public function emitirSonido() {
        echo "¡Miau!";
    }
}

class Planeta implements Esfera {
    // propiedades
    // ...

    public function rotar() {
        echo "Rotando $grados"."° grados";
    }

    public function getDiametro() : float {
        return $this->__diametro;
    }
}
```

### 1.4.6 Reutilización

La reutilización es uno de los principios fundamentales y más útiles de la programación orientada a objetos. Una funcionalidad tiende a ser útil en más de un contexto, y resulta provechoso poder utilizar un código que ya existe en lugar de repetir la misma funcionalidad por cada contexto que se aborda. En el paradigma orientado a objetos la reutilización se presta en dos conceptos relacionados: herencia y polimorfismo.

#### Herencia

Muchos elementos que se abstraen mediante el paradigma orientado a objetos obedecen a una jerarquía de categorías. Por ejemplo, en biología un gato es también un mamífero, e inmediatamente tenemos conciencia de que existen muchos *tipos* de mamíferos, como los perros, las focas y un enorme etcétera. Lo mismo puede aplicar al resto de la realidad, a las matemáticas, la geometría. etc.

Recordando que el paradigma de la POO plantea el modelado de la realidad por medio de la abstración, mediante la herencia es posible modelar esa jerarquía de categorías. Naturalmente, dependiende del sistema que se modela puede interesar una mayor o menor profundidad en el modelado de dichas categorías, por ejemplo si entre «Mamífero» y «Gato» interesa abstraer «Felino» o no.

La herencia es también un mecanismo de **especialización**. Si se tiene una funcionalidad general en una clase y se necesita una funcionalidad específica al tiempo que se debe mantener el resto de funcionalidad general, entonces la herencia permite implementar clases con esas funcionalidades especializadas.

La herencia plantea a las clases *superiores* o *generales* como **clases base**, y las *menores* o *especializadas* como **clases derivadas**. También se entiende que las clases derivadas **extienden** de las clases bases, y que una clase derivada **es un tipo** de la clase base. Para definir una clase que hereda de otra se emplea el identificador **extends** en la sentencia de declaración de clase: `class ClaseDerivada extends ClaseBase {}`.

```php
class Mamifero {
    private $__colorDeOjos;
    private $__nombre;
    private $__nacimiento;

    // Métodos
    // ...

    public function emitirSonido() {
        echo "«Un mamífero cualquiera»";
    }
}

class Gato extends Mamifero {
    public function emitirSonido() {
        echo "¡Miau!";
    }
}

class Perro extends Mamifero {
    public function emitirSonido() {
        echo "¡Guau!";
    }
}

$gato = new Gato();
$perro = new Perro();

$gato->emitirSonido(); // ¡Miau!
$perro->emitirSonido(); // ¡Guau!
```

En el ejemplo se puede observar como en la clase `Mamifero` se define e implementa todo aquello (que interesa y) que es común a un mamífero cualesquiera, y luego las clases `Gato` y `Perro` extienden de `Mamifero` *reimplementado* o *especializando* el método `emitirSonido()`.

---

**Nota:** Existen dos tipos de herencia: **simple** y **múltiple**. La mayoría de lenguajes que soportan POO, como PHP, suelen implementar únicamente la herencia simple. Otros lenguajes, como C++ soportan la herencia múltiple. El criterio suele ser el factor de complejidad, en tanto los conflictos tienden a ser mayores y más complejos en la herencia múltiple que en la herencia simple.

---

### Traits

La herencia simple es la única seportada por PHP, pero existen también ciertas ventajas en la herencia múltiple. Los *Traits* (*Rasgos*) proveen un mecanismo que pretende proveer lo mejor de la herencia múltiple evitando sus problemas y complejidades. Los *Traits* son construcciones semejantes a las clases en tanto son una composiciones de propiedades y métodos. Se asemejan a una Interfaz en tanto exponen una interfaz y un *contrato* de composición y funcionalidad, pero un Rasgo **no es** un tipo, sino que se limita a componer el modo y forma de un tipo. Luego, las clases pueden incluir los Rasgos en su construcción. De esta forma es posible definir Rasgos de funcionalidad más *atomizada* en archivos específicos, utilizar tales Rasgos en tantas clases como haga falta y, en igual medida, redefinir en las clases lo que los Rasgos definen.

Por ejemplo, los planetas tienen un eje y rotan en torno al mismo, pero no solo los planetas rotan, también lo pueden hacer los círculos, los cilindros, los conos, etc. No obstante no sería la mejor práctica definir una clase base que implemente únicamente un método `rotar()` y luego toda la jerarquía de clases desde dicha clase base. Es mucho

más eficiente definir un Rasgo `Rotar` y agregarlo a toda clase que haga falta.

```
trait RotarAware {
  public function rotar(int $grados) {
      if($grados < 1 || $grados > 360) echo "Grados de rotación fuera de rango.";
      else echo "Rotando $grados"."° grados";
  }
}

class Planeta {
    use RotarAware;
}

class Cilindro {
    use RotarAware;
}

$marte = new Planeta();
$unCilindro = new Cilindro();

$marte->rotar(120);
$unCilindro->rotar(180);
```

En el ejemplo puede advertirse otra ventaja, y es que mienrtas la funcionalidad sea exactamente la misma aun en clases diferentes, al ser un sólo lugar dónde se implementa tal funcionalidad se maximisa la reutilización de códiogo.

### Polimorfismo

El polimorfismo se relaciona estrechamente con la herencia, este principio del paradigma plantea precisamente lo que en el apartado de Herencia se expuso como especialización. Pero el polimorfismo se refiere a la idea de que un método de igual firma (en decir, nombre, retorno, número y tipo de parámetros) en diferentes clases implementen diferentes funcionalidades. En el ejemplo de herencia las clases `Gato` y `Perro` heredan de `Mamífero`, pero el método `emitirSonido()` en la clase `Gato` imprime un `¡Miau!` y en la clase `Perro` imprime `¡Guau!`.

### Espacios de nombres (*Namespaces*)

Como se mencionó antes, la comunidad de PHP tiene un montón de desarrolladores creando un montón de código. Esto significa que una biblioteca de PHP puede utilazr el mismo nombre de clase que otra. Cuando ambas bibliotecas son usadas en el mismo espacio de nombre, colisionan y causan problemas.

Los *espacios de nombres* solucionan este problema. Tal como se describe en el Manual de referencia de PHP, los espacios de nombres pueden compararse con los directorios de los sistemas operativos que dan *espacios de nombres* a los archivos; dos archivos con el mismo nombre coexisten en directorios separados. Semejante, dos clases de PHP con el mismo nombre pueden coexistir en distintos *espacios de nombres*. Tan sencillo como eso.

Es importante establecer el código dentro de espacios de nombres de tal manera que puedan ser utilizados por otros desarrolladores sin que tengan que preocuparse por colisiones con otras bibliotecas.

Una recomendación dada para el uso de espacios do nombres se encuentra en la PRS-4, que sugiere proveer una convención a los archivos, clases y espacios de nombres para permitir un código de conectar y ejecutar.

En octubre de 2014 el PHP-FIG marcó obsolescente la anterior convención de autocarga: PSR-0. Tanto PSR-0 como PSR-4 permanecen perfectamente utilizables. Los últimos requerimientos con PHP5.3 y muchos proyectos exclusivos en PHP 5.2 implementan la PSR-0.

Si vas a utilizar una convención de autocarga para un nuevo proyecto o paquete, considerá PSR-4.

- Leer sobre espacios de nombres
- Leer sobre PSR-0
- Leer sobre PSR-4

### 1.4.7 Estructurando un proyecto

### 1.4.8 La biblioteca estándar de PHP (*Standard PHP Library [SPL]*)

La biblioteca estándar de PHP está empaquetada con el propio motor de PHP y provee una colección de clases e interfaces. Está construida principalmente para atender las típicas necesidades de clases de estructuras de datos (*stack* [lista], *queue* [cola], *heap* [pila], etc), e iteradores que pueder recorrer sobre esas estructuras de datos o sobre tus propias clases que implementen interfaces de la SPL.

- Leer sobre la SPL

## 1.5 Gestión de dependencias

la comunidad de PHP es basta y cuenta con miles de bibliotecas,*frameworks* y componentes para escoger. Seguramente tu proyecto terminará utilizando muchas de esas bibliotecas, y por tanto tu preyecto *dependerá* de ellas, por eso las llamamos *dependencias*. Hasta hace no mucho, PHP no contaba con una buena forma de gestionar esas dependencias. Incluso cuando se gestionan manualmente, toca preocuparse por los autocargadores. Pero hoy por hoy eso ya no es un ningún problema.

Actualmente existen dos principales gestores de dependencias para PHP: Composer y PEAR. Composer es por mucho el gestor de dependencias más popular para PHP, aunque por mucho tiempo PEAR fue el favorito de la comunidad. Conocer la historia de PEAR es buena idea, podrías encontrar referencias a PEAR incluso aunque no lo hayas usado nunca.

### 1.5.1 Composer y *Packagist*

Composer es un gestor de dependencias *brillante*. Basta con listar las dependencias de un proyecto en el archivo `composer.json` y, con unos sencillos comandos, Composer automáticamnte descargará y configurará la autocarga de dependencias. Composer es el análogo de NPM en el mundo de `node.js`, o de *Bundler* en el mundo Ruby.

Existe un montón de bibliotecas para PHP que son compatibles con Composer, listas para ser utilizadas en cualquier proyecto PHP. Esos «paquetes» se listan en Packagist, el repositorio oficial de las bibliotecas PHP compatibles con Composer.

#### Cómo instalar Composer

La mejor manera para descargar Composer es siguiendo las instrucciones oficiales. Esto verificará que el instalador no esté corrupto o amañado. El instalador instalará Composer *localmente* en el directorio actual de trabajo.

Se recominda instalarlo *globalmente* (por ejemplo, una copia única en `/usr/local/bin`), para hacerlo (dependiendo de tu sistema) podrías ejecutar:

```
mv composer.phar /usr/local/bin/composer
```

**Nota:** Si lo anterior falla debido a permisos, anteponé el comando `sudo`.

Para ejecutar Composer instalado localmente se debe invocar utilizando la llamada de PHP: `php composer.phar`; y cuando está instaldo globalmente basta `composer`.

#### Instarar en Windows

Para lo usuarios de Windows la manera más sencilla para instalar y ejecutar Composer es utilizar el instalador ComposerSetup, que prepara una instalación global y configura el `$PATH` de Windows de tal manera que basta invocar el comando `composer` desde cualquier directorio en la CLI.

#### Cómo instalar composer manualmente

La instalación manual de composer en una técnica avanzada. . .

### 1.5.2 PEAR

El gestor de dependencias veterano que muchos desarrolladores de PHP disfrutan es PEAR.

#### Cómo instalar PEAR

Para instalar PEAR

#### Cómo instalar un paquete

Si un paquete

#### Manejando dependencias de PEAR con Composer

Si se tiene

## 1.6 Guía de estilo y las *PSR*

La comunidad de PHP es amplia y diversa, compuesta por una innumerable cantidad de bibliotecas, *frameworks* y componentes. Es común para los desarrolladores de PHP escoger varios de esos elementos y combinarlos en un sólo proyecto. Es importante que el código PHP se adhiera (tanto como sea posible) a un estilo de código común para facilitar a los desarrolladores el mezclar y empalmar varias bibliotecas en sus proyectos.

El Framework Interop Group (Grupo de Iteroperatividad de *Frameworks*) ha propuesto y aprobado una serie de recomendaciones de estilo. No todas están relacionadas con *estilo de código*, pero aquellas que sí lo hacen son las PSR-0, PSR-1, PSR-2 y PSR-4. Dichas recomentaciones no son más que una serie de reglas que muchos proyectos como Drupal, Zend, Symfony, Laravel, CakePHP, phpBB, AWS SDK, FuelPHP, Lithium, etc,. han adoptado. Todo(a) desarrollador(a) puede utilizarlas en sus propios proyectos, pero no son ninguna imposición, también cada cual puede continuar con su propio estilo.

No obstante, idealmente, se debería escribir código PHP que se adhiera a convenciones conocidas (porque en ello radica la *interoperatividad* ). Esto puede ser cualquier combinación de PSR, o una de las convenciones definidas por PEAR o Zend. Esto significa que otros desarrolladores pueden leer y trabajar fácilmente con código de terceros, y aquellas aplicaciones que implementen los componentes podrán mantenerse consistentes incluso cuando trabajen con un amplio rango de código de terceros.

- Leé sobre PSR-0
- Leé sobre PSR-1
- Leé sobre PSR-2
- Leé sobre PSR-4
- Leé sobre la convención de estilo de PEAR
- Leé sobre la convención de estilo de Symfony

Podés utilizar PHP_CodeSniffer para verificar el código contra alguna de las recomendaciones listadas, y extensiones para editores de texto como Atom para obtener retroalimentación en tiempo real.

Es posible corregir un esquema de código automáticamente utilizando una de las siguientes herramientas:

- PHP Coding Standards Fixer (Corrector de convenciones de código para PHP) que posee usa nabe de código muy bien probada.
- PHP Code Beautifier and Fixer (Corrector y embellecedor de código para PHP) herramienta que se incluye con *PHP_CodeSniffer* y que puede utilizarse para ajustar el código como corresponde.

Y puede ejecutarse el comando *phpcs* manualmente desde consola:

```
phpcs -sw --standard=PSR2 file.php
```

La herramienta mostrará los errores y como corregirlos. También puede ser útil incluir este comando en un *hook* de git. así, ramas que contengan violaciones contra una convención particular no podrán entrar en el repositorio hasta que las misma sean corregidas.

Si se tiene *PHP_CodeSniffer*, entonces se puede corregir los problemas de esquema de código reportados, automáticamente, con PHP Code Beautifier and Fixer.

```
php-cs-fixer fix -v --level=psr2 file.php
```

El inglés está predefinido para todos los nombres simbólicos y código de infraestructura. Los comentarios pueden estar escritos en cualquier idioma fácilmente legible por todas las personas actuales y futuras que puedan trabajar con el código.

## 1.7 Buenas prácticas

### 1.7.1 Los principios

PHP es un lenguaje basto que permite a desarrolladores con cualquier nivel de habilidad producir código, no sólo rápidamente, sino también eficientemente. No obstante, mientras se avanza en el aprendizaje del lenguaje, tiende a olvidarse (o pasar por alto) las buenas prácticas en favor de atajos y malos hábitos. Para ayudar a prevenir esta inicidencia, esta sección se enfoca en recordar a los desarrolladores de las buenas prácticas a aplicar con PHP.

### 1.7.2 Fecha y tiempo

PHP provee la clase `DateTime` que facilita el trabajo para leer, escribir, comparar o calcular fecha y tiempo. Además de la citada clase, en PHP existen muchas funciones relacionadas con tiempo y hora, pero `DateTime` provee una cómoda interfaz orientada a objetos para la mayoría de los usos posibles.

Para comenzar, considérsse el siguiente ejemplo:

```php
$raw = '06/09/1987';
$date = \DateTime::createFromFormat('d/m/Y', $raw);
$today = new \DateTime();

echo 'Fecha: ' . $date->format('Y-m-d') . "\n"; // 1987-09-06
echo 'Hoy: ' . $today->format('Y-m-d') . "\n";
```

A partir del método fábrica `createFromFortmat()` se crea una instancia de `DateTime` en función de los argumentos proveidos. Luego, indicando al método `format()` un formato específico se emite una nueva cadena de fecha con dicho formato. Cuando en lugar del método fábrica se instancia mediante el operador `new` entonces el objeto se crea con la fecha actual.

Es posible realizar cálculo sobre objetos `DateTime` con la clase `DateInterval`. `DateTime` provee métodos como `add()` y `sub()` que toman un argumento tipo `DateInterval`.

```php
<?php
$start = \DateTime();
// crear un copia de $start y adicionar un mes y 6 días
$end = clone $start;
$end->add(new \DateInterval('P1M6D'));

$diff = $end->diff($start);
echo 'Diferencia: ' . $diff->format('%m mes, %d días (total: %a días)') . "\n";
// Diferencia: 1 mes, 6 días (total: 37 días)
```

En objetos tipo `DateTime` es posible realizar una comparación regular:

```php
<?php
if($start < $end) {
    echo "¡El inicio es anterior al final!\n";
}
```

En el caso de iterar sobre eventos recurrente es posible hacer uso de la clase `DatePeriod`. Toma dos valores de tipo `DateTime`, inicio y fin, y el intérvalo para el que deolverá todos los eventos de por medio.

```php
<?php
// muestra todos los jueves entre start y $end
$periodInterval = \DateInterval::createFromDateString('first thursday');
$periodIterator = new \DatePeriod($start, $periodInterval, $end, \DatePeriod::EXCLUDE_
↪START_DATE);

foreach ($periodIterator as $date) {
    // mostrar cada fecha en el periodo
    echo $date->format('Y-m-d') . ' ';
}
```

Carbon es una extensión popular para PHP. Hereda toda la funcionalidad existente en la claase `DateTime`, por lo que implica mínimas adaptaciones en el código, no obstante provee características extra como el soporte de Localización, así como diversos mecanismos para agregar, restar, y formatear objetos `DateTime`, además de una forma de probar el código emulando fechas y tiempo arbitrarios.

- Leer sobre DateTime
- Leer sobre el formateado de fechas

### 1.7.3 Patrones de diseño

Al construir un proyecto es útil utilizar patrones comunes a lo largo del código y estructuras. Utilizar patrones es una gran ayuda en tanto que una mayor facilidad para gestionar el código y permitir a otros desarrolladores entender rápidamente como todo encaja y se estructura.

Si se utiliza un *framework* entonces la mayoría del código de alto nivel y la estructura del proyecto estarán basados en ese *framework*, así que muchas decisiones sobre patrones ya se encuentran tomadas. Pero sigue siendo una decisión de cada desarrollador(a) escoger los mejores patrones a seguir en el código que escribe aún sobre el *framework*. Si, por otra parte, no se está utilizando un *framework* para construir un proyecto, entonces se debe encontrar los mejores patrones acordes al tipo y tamaño de la aplicación que se construye.

- Leer más sobre patrones de dinseño (y ejemplo útiles).

### 1.7.4 Trabajando con UTF-8

Hasta ahora, PHP no soporta Unicode a bajo nivel. Existen mecanismos para asegurar un correcto procesamiento de cadenas UTF-8, pero no son sencillos, hay que ser cuidadosos, detallistas, consistentes, y se requiere lidiar con ellos por toda la estructura de la aplicación, desde el html, pasando por el SQL hasta el propio PHP.

#### UTF-8 a nivel de PHP

Las operaciones básicas con cadenas, como la concatenación o la asinación de cadenas a variables, realmente no requiere de ningún tratamiento especial para UTF-8. No obstante, la mayoría de funciones sobre cadenas, como `strpos()` y `strlen`, necesitan de consideraciones especiales. Esas funciones tienen un `mb_*` equivalente, por ejemplo: `mb_strpos()` y `mb_strlen()`. Tales funciones están especialmente diseñadas para operar con cadenas Unicode y se habilitan vía la extensión Multibyte String.

Se ha de tener el cuidado de utilizas las funciones `mb_*` siempre que se opere con cadenas Unicode. Por ejemplo, si se utiliza la función `substr()` en una cadena UTF-8, existe un buena probabilidad de que el resultado incluya algunos caracteres cortados. La función correcta a utilizar debe ser su correspondiente alternativa de *multibyute*, `mb_substr()`.

Lo difícil puede ser recordar utilizar siempre las funciones `mb_*`. Con solo una que se olvide, las cadenas Unicode tratadas por dicha función podrían corromperse durante su procesamiento.

Ahora, no todas las funciones tienen una contraparte `mb_*`, si no existe la que se necesita para un caso dado, entonces se tiene un lío.

Además se debería utilizar `mb_internal_encoding()` al inicio de cada archivo PHP (o al principio de un archivo global de inclusión o autocarga), y `mb_http_output()` justo después de que se emita una salida al navegador. La definición explícita de codificación de cadenas en cada archivo puede salvar a más de alguno(a) de un dolor de cabeza.

También, muchas funciones de PHP que operan sobre cadenas tienen un parámetro opcional que permite especificar la codificación de caracteres. Siempre debería indicarse explícitamente `'UTF-8'` en dicho parámetro. Por ejemplo, `htmlentities()` soporta un tercer parámetro para indicar la codificación de caracteres, y siempre se debería indicar `'UTF-8'`. Nótese que en PHP 5.6 y posterior, la opción de configuración default_charset se emplea como valor predeterminado. PHP 5.4 y 5.5 utilizan UTF-8 como valor predeterminado. Las versiones anteriores de PHP emplean ISO-8859-1.

Si se esstá construyendo una aplicación distribuída y no se puede segurar que la extensión `mbstring` esté disponible, entonces considerar el paquete patchwork/utf8. Este paquete utiliza la extensión `mbstring` cuando está disponible, y de lo contrario provee de un soporte hacia funciones no UTF-8.

Finalmente, si se tiene la extensión `mbstring` instalada, se puede hacer uso de ocelote, que provee una clase `StringHelper` con métodos estáticos que envuelven y agregan funcionalidades a muchas de las funciones `mb_*` (incluyendo algunas funcionalidades no existentes en la extensión de `mbstring`). De esta manera se disminuye el

riesgo de olvidar utilizar la correspondiente alternativa `mb_*` y se gana legibilidad, por su puesto, el pago es un pequeño sobre esfuerzo (*overhead*) que implica hacer estas llamadas.

### UTF-8 a nivel de base de datos

Si un programa de PHP se conecta a MySQL (o SQL Server), existe una buena probabilidad de que las cadenas no se almacenen codificadas como UTF-8 incluso si se consideran las precauciones citadas en el apartado anterior.

Para garantizar que las cadenas desde PHP a MySQL se almacenen como UTF-8, hay que asegurarse de que la base de datos y sus tablas estén configuradas con la opción `utf8mb4`, y que en la cadena de conexión se indique la codificación de caracteres con dicho valor. Nótese que se debe indicar **‘‘utf8mb4‘‘** para soportar el conjunto completo de UTF-8, y **no** solo el conjunto `utf8`.

En el caso de SQLServer, la extensión `sybase` **no** soporta UTF-8, por lo que si se utiliza esta extensión para la conexión, antes de realizar cualquier escritura, se deben convertir las cadenas a ISO-8859-1. No obstante lo mejor ees prescindir de la extensión `sybase` y emplear el controlador **‘‘sqlsrv‘‘** proveido por Microsoft, dicho controlador por defecto funciona con UTF-8.

- Leer más sobre las constantes del controlador sqlsrv (inglés)
- Leer cómo enviar y obtener datos en UTF-8 usando el soporte nativo UTF-8 (inglés)

### UTF-8 a nivel del navegador

Para asegurar que las salidas de PHP al navegador sean en UTF-8 se debe utilizar la función `mb_http_output()`.

Se le debe indicar al navegador mediante la respuesta http que la página debe considerarse como UTF-8. Actualmente, es común establecer la codificación de caracteres en una cabecera de la Respuesta:

```
header('Content-Type: text/html; charset=UTF-8');
```

Un mecanismo histórico que se ha utilizado ha sido establecer una etiqueta `meta` en la sección `head` del html.

```
<?php
// Indicar a PHP que las cadenas de todo el archivo son en UTF-8
mb_internal_encoding('UTF-8');

// Indicar que las salidas también serán en UTF-8
mb_http_output('UTF-8');

// He aquí una cadena en UTF-8
$string = 'Êl síla erin lû e-govaned vîn.';

// Realizar una operación sobre la cadena con nua función *multibyte*
// Nótese como se hace una substracción a un caracter no-Ascii para fines␣
↪demostrativos
$string = mb_substr($string, 0, 15);

// Conectar a una base dadtos para almacenar la nueva cadena
// Nótsee el `charset=utf8mb4` en la cadena de conexión (DSN)
$link = new PDO(
    'mysql:host=your-hostname;dbname=your-db;charset=utf8mb4',
    'your-username',
    'your-password',
    [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_PERSISTENT => false
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ]
);

// Almacenar la cadena resultante en la base de datos
// La base de datos y sus tablas están en codificación utf8mb4
$handle = $link->prepare('INSERT INTO ElvishSentences (Id, Body) VALUES (?, ?)');
$handle->bindValue(1, 1, PDO::PARAM_INT);
$handle->bindValue(2, $string);
$handle->execute();

// Obtener la cadena almacenada para comprobar que está correcta stored correctly
$handle = $link->prepare('select * from ElvishSentences where Id = ?');
$handle->bindValue(1, 1, PDO::PARAM_INT);
$handle->execute();

// Almacenar el resultado en un objeto que será enviado en el HTML
$result = $handle->fetchAll(\PDO::FETCH_OBJ);

header('Content-Type: text/html; charset=UTF-8');
?><!doctype html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Página de prueba de UTF-8</title>
    </head>
    <body>
        <?php
        foreach($result as $row) {
            print($row->Body);  // Esto debería mostrar correctamente en el navegador
↪la cadena en UTF-8
        }
        ?>
    </body>
</html>
```

### Lecturas recomendadas (en español e inglés):

- Manual de PHP: Operadores de cadenas
- **Manual de PHP: Funciones de cadenas**
    - strpos()
    - strlen()
    - substr()
- **Manual de PHP: Funciones de cadenas de caracteres multibyte**
    - mb_strpos()
    - mb_strlen()
    - mb_substr()
    - mb_internal_encoding()
    - mb_http_output()
    - htmlentities()

        - htmlspecialchars()
- Stack Overflow: What factors make PHP Unicode-incompatible?
- Stack Overflow: Best practices in PHP and MySQL with international strings
- Stack Overflow: DOMDocument loadHTML does not encode UTF-8 correctly
- How to support full Unicode in MySQL databases
- Bringing Unicode to PHP with Portable UTF-8

### 1.7.5 Internacinalización (i18n) y localización (l10n)