

Programación orientada a objetos en PHP 8

Declaraciones de tipo desde PHP 7.4.0

Las **declaraciones de tipo** se pueden añadir a:

- Los argumentos de función o método
- Valores de retorno de una función o método
- Propiedades o atributos de clase.

Estas declaraciones de tipo aseguran que el valor es del tipo especificado en el momento de la llamada, de no ser así se lanzará una excepción

Declaraciones de tipo desde PHP 7.4.0

- Esto nos permitirá que nuestro desarrollo sea más robusto, y que los métodos y funciones no acepten cualquier tipo de argumento.
- También se puede aplicar para el retorno, es decir, si un método tiene que devolver una cadena, se forzará a que siempre sea así.
- En caso contrario, obtendremos un `TypeError`, querrá decir que el método está devolviendo un valor que no queremos, o está recibiendo algún argumento de un tipo diferente al que esperamos.

Declaraciones de tipo desde PHP 7.4.0

A partir de la versión 7.4.0 de PHP disponemos de la posibilidad de indicar:

- Las **declaraciones de tipo** pueden ser marcadas como nullable(anulables) anteponiendo un interrogante **?** al nombre de tipo. Esto conlleva que el valor puede ser del tipo especificado o null.
- El **tipo de retorno de una función o método** es de un tipo concreto o devuelve null. Para ello solo habrá que añadir un interrogante **?** delante del nombre del tipo.

Declaraciones de tipo desde PHP 7.4.0

```
function dividirPorNumero(float $numero): ?float
{
    if ($numero != 0) {
        return 7 / $numero;
    } else {
        return NULL;
    }
}

echo "<br>Argumento 12.1 y el resultado de dividir es : <br>" .
dividirPorNumero(12.1);

echo "<br>Argumento 0 y el resultado de dividir es : <br>";
var_dump(dividirPorNumero(0));
```



Declaraciones de tipo desde PHP 7.4.0

```
function tipoDeMedalla(int $posicion): ?string
{
    switch ($posicion) {
        case 1:
            return 'Oro';
        case 2:
            return 'Plata';
        case 3:
            return 'Bronce';
        default:
            return null;
    }
}

echo tipoDeMedalla(2);
// Plata

echo tipoDeMedalla(34);
```

Declaraciones de tipo desde PHP 7.4.0



```
function printNullable(?string $str): void
{
    if ($str === null) {
        echo "El valor es nulo.";
    } else {
        echo "El valor es: " . $str;
    }
}
```

```
function concatenar(?string $str1, ?string $str2): ?string
{
    if ($str1 === null || $str2 === null) {
        return null;
    }
    return $str1 . $str2;
}
```

Declaraciones de tipo desde PHP 7.4

De forma automática PHP arreglará las incompatibilidades de tipos.

```
function incrementar(int $num): int
{
    return $num + 1;
}
echo incrementar(4.5); // 5
```

Incluir la línea **declare(strict_types=1)** en un archivo PHP significa que se activa el modo estricto para tipos.

Declarar strict types en nuestros ficheros PHP, forzará a que nuestros métodos y funciones acepten variables únicamente del tipo exacto que se declaren. En caso contrario lanzará un **TypeError**.

Esto significa que los tipos de datos de los argumentos y valores de retorno de las funciones y métodos se comprueban de forma estricta.

Declaraciones de tipo desde PHP 8

La versión PHP 8 permite indicar tipos diferentes para el retorno de una función.

```
function duplicarPositivo(float $numero): float|string|null
{
    if ($numero > 0) {
        return $numero * 2;
    }
    if ($numero < 0) {
        return NULL;
    }
    if ($numero == 0) {
        return "El valor es cero";
    }
}

echo "<br>Argumento 12.1 y el resultado es: <br>" . duplicarPositivo(12.1);
echo "<br>Argumento -45 y el resultado es: <br>";
var_dump(duplicarPositivo(-45));
echo "<br>Argumento 0 y el resultado es: <br>" . duplicarPositivo(0);
```

```
<?php declare(strict_types=1);
```

Strict_types

- Incluir la línea `declare(strict_types=1);` en un archivo PHP significa que se activa el modo estricto para tipos.
- La declaración `declare(strict_types=1);` debe colocarse en la parte superior del archivo PHP, antes de cualquier otra declaración o código.
- Declarar strict types en nuestros ficheros PHP, forzará a que nuestros métodos y funciones acepten variables únicamente del tipo exacto que se declaren. Se comprueban de forma estricta. En caso contrario lanzará un `TypeError`.
- Esto significa si una función espera un tipo de dato específico (por ejemplo, un entero), PHP generará un error si se le pasa un tipo diferente (como un string) en lugar de intentar convertirlo de manera implícita.
- El modo estricto solo se aplica al archivo en el que se declara y no a otros archivos incluidos o requeridos.
- Si queremos que nuestra aplicación sea estricta tendremos que declarar el modo estricto en todos los archivos PHP de nuestro desarrollo.



```
<?php declare(strict_types=1);
```

Strict_types

```
<?php
declare(strict_types=1);

function sum(int $a, int $b): int {
    return $a + $b;
}

$num1 = 5;
$num2 = "10"; // Esto es una cadena en lugar de un número entero

$result = sum($num1, $num2); // Esto generará un error debido a strict_types

echo $result;
?>
```

ARGUMENTOS CON NOMBRE

- PHP 8.0 introdujo los argumentos con nombre como una extensión de los parámetros posicionales existentes.
- Los argumentos con nombre permiten pasar argumentos a una función o métodos basándose en el nombre del parámetro, en lugar de la posición del mismo.
- Qué conseguimos:
 1. Que el significado del argumento sea autodocumentado(si se ha utilizado un nombre significativo para el argumento)
 2. Hace que los argumentos sean independientes del orden.
 3. Permite saltar los valores por defecto de forma arbitraria.
- Los argumentos con nombre se pasan **anteponiendo al valor el nombre del parámetro, seguido de dos puntos.**
- Se permite el uso de palabras clave reservadas como nombres de parámetros. Ej. (\$array sería un nombre de parámetro válido)
- El nombre del parámetro debe ser un identificador, no se permite especificarlo dinámicamente:
 - Ejemplo, **no** es posible utilizar **una variable** como nombre de parámetro.

- Los argumentos con nombre pueden combinarse con argumentos sin nombre, también llamados ordenados.
- Cuando combinas parámetros posicionales y por nombre, los parámetros por posición deben ir siempre primero y en el mismo orden en que se definieron en la función.
- Si se pasa un parámetro por nombre antes de los posicionales, PHP generará un error.

```
<?php
declare(strict_types=1);

// Definimos una función con varios parámetros.
function mostrarDatos(string $nombre, int $edad, string $correo, string $ciudad): void
{
    echo "Nombre: $nombre<br>";
    echo "Edad: $edad<br>";
    echo "Correo: $correo<br>";
    echo "Ciudad: $ciudad<br>";
}

// Creamos un array con los datos.
$datos = [
    'nombre' => 'Laura',
    'correo' => 'laura@ejemplo.com',
    'ciudad' => 'Madrid'
];
// Definimos una variable para la edad.
$edadUsuario = 25;

// Llamada a la función combinando paso por posición y por nombre
mostrarDatos(
    $datos['nombre'],           // Parámetro por posición (nombre).
    $edadUsuario,              // Parámetro por posición (edad).
    ciudad: $datos['ciudad'],  // Parámetro por nombre (ciudad desordenado).
    correo: $datos['correo']   // Parámetro por nombre (correo desordenado).
);
?>
```

LISTAS DE ARGUMENTOS DE LONGITUD VARIABLE (SPREAD) ...

PHP utiliza los tres puntos (...) para nombrar el operador de propagación (spread).

PHP tiene soporte para listas de argumentos de longitud variable en **funciones o métodos** definidas por el usuario.

Esto se implementa utilizando:

- El operador ... **convierte los argumentos variables en un array automáticamente.**
- Las listas de argumentos pueden incluir el operador ... para indicar que la función acepta un número variable de argumentos.
- Los argumentos serán pasados a la función dada **como un array**. (propagación de matrices).
- Se pueden especificar argumentos posicionales normales **antes** del operador ...
- En este caso, solamente los argumentos al final, que no coincidan con un argumento posicional, serán añadidos al array generado por ...
- También es posible añadir una declaración de tipo antes del símbolo ... Si está presente, todos los argumentos capturados por ... deben ser del tipo indicado.

LISTAS DE ARGUMENTOS DE LONGITUD VARIABLE Y OPERADOR SPREAD

```
<?php
declare(strict_types=1);

// Definimos la función con un argumento fijo y un número variable de argumentos
function sumarNumeros(string $mensaje, int ...$numeros): int
{
    // Inicializamos la suma
    $suma = 0;

    // Iteramos sobre el array de argumentos
    foreach ($numeros as $numero) {
        $suma += $numero;
    }

    // Mostramos el mensaje
    echo $mensaje . ": " . $suma . "\n";

    // Devolvemos la suma total
    return $suma;
}

// Llamamos a la función con mensaje y diferentes números de argumentos
sumarNumeros("La suma es", 1, 2, 3); // La suma es: 6
sumarNumeros("Total acumulado", 5, 10, 15, 20); // Total acumulado: 50
?>
```


OPERADOR SPREAD (...) PARA MEZCLAR O COMBINAR ARRAYS

```
<?php
```

```
$numbers = [2, 3, 4];
```

```
$scores = [1, ...$numbers, 5];
```

```
var_dump($scores);
```

```
array (size=5)
```

```
0 => int 1
```

```
1 => int 2
```

```
2 => int 3
```

```
3 => int 4
```

```
4 => int 5
```

```
<?php
```

```
$pares = [2, 4, 6];
```

```
$impares = [1, 3, 5];
```

```
$todos = [...$pares, ...$impares];
```

```
var_dump($todos);
```

```
array (size=6)
```

```
0 => int 2
```

```
1 => int 4
```

```
2 => int 6
```

```
3 => int 1
```

```
4 => int 3
```

```
5 => int 5
```

Expandir el contenido del array `$pares` y luego expandir el contenido del array `$impares`.

Cada elemento del array `$pares` se coloca en el nuevo array `$todos`, seguido por cada elemento del array `$impares`.

FUNCIONES CON NÚMERO DE ARGUMENTOS VARIABLES

ARGUMENTOS CON NOMBRE

OPERADOR SPREAD ...

- Es posible utilizar el operador spread en combinación con argumentos con nombre.
- Si faltan entradas requeridas en el array, o si hay una clave que no está listada como un argumento con nombre, se lanzará un error

```
<?php
declare(strict_types=1);

// Definimos una función que acepta argumentos con nombre
function registrarUsuario(string $nombre, string $email, int $edad): void
{
    echo "Nombre: $nombre<br>";
    echo "Email: $email<br>";
    echo "Edad: $edad<br>";
}

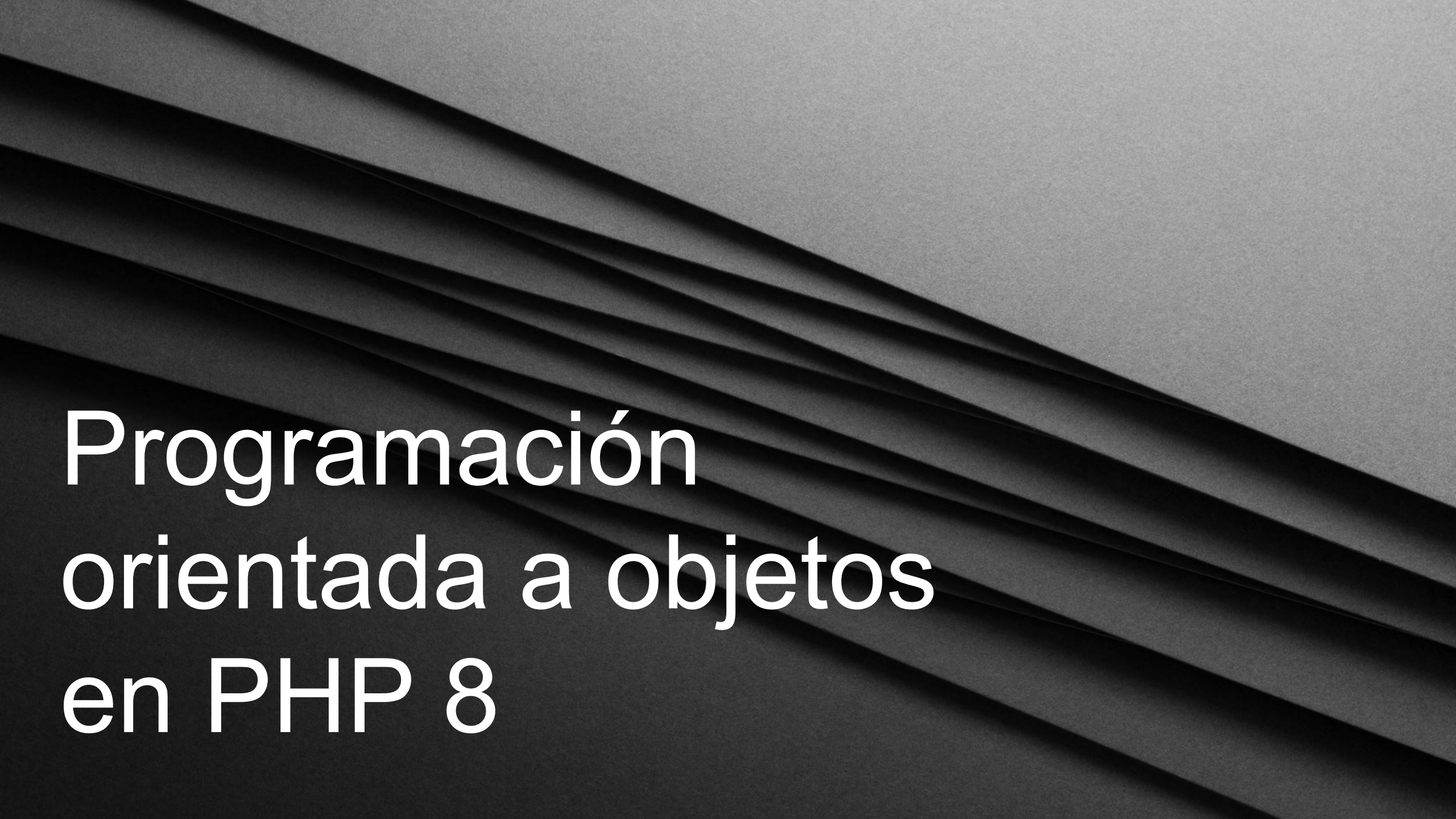
// Creamos un array con los datos del usuario
$datosUsuario = [
    'nombre' => 'María',
    'email' => 'maria@ejemplo.com',
    'edad' => 28 // Si este valor está comentado, se lanzará un error.
];

// Llamamos a la función usando propagación de arrays
registrarUsuario(...$datosUsuario);

//Ahora comentamos la edad para que se genere un error
$datosUsuario = [
    'nombre' => 'María',
    'email' => 'maria@ejemplo.com',
    // 'edad' => 28 // Comentamos este valor
];

// Llamamos a la función usando propagación de arrays y pasando la clave 'edad' por nombre: CORRECTO
registrarUsuario(
    ...$datosUsuario,
    edad: 28 // La edad se pasa por nombre
);

// Si se omite la clave 'edad' en el array, lanzará un ERROR al intentar acceder.
registrarUsuario(
    ...$datosUsuario // Esto generará un error, ya que falta 'edad'
);
```



Programación orientada a objetos en PHP 8

Declarar una clase y crear un objeto

Para declarar una clase, se utiliza la palabra clave **class** seguido del nombre de la clase. Para instanciar un objeto a partir de la clase, se utiliza **new**:

```
<?php
class NombreClase
{
    // propiedades
    // y métodos
}

$ob = new NombreClase();
```

Clases con mayúscula: Todas las clases empiezan por letra mayúscula.

Declarar una clase y crear un objeto

Una vez que hemos creado un objeto, se utiliza el operador `->` para acceder a una propiedad o un método:

```
$objeto->propiedad;  
$objeto->método(parámetros);
```

Si desde dentro de la clase, queremos acceder a una propiedad o método de la misma clase, utilizaremos la referencia `$this`

```
$this->propiedad;  
$this->método(parámetros);
```

Declarar una clase y crear un objeto

Aunque se pueden declarar varias clases en el mismo archivo, es una mala práctica. Así pues, cada fichero contendrá una sola clase, y se nombrará con el nombre de la clase.

Como ejemplo, codificaríamos una persona en el fichero **Persona.php** como:

```
<?php
class Persona {
    private string $nombre;

    public function setNombre(string $nom) {
        $this->nombre=$nom;
    }

    public function imprimir(){
        echo $this->nombre;
        echo '<br>';
    }
}

$bruno = new Persona(); // creamos un objeto
$bruno->setNombre("Bruno Díaz");
$bruno->imprimir();
```

Encapsulación.

Visibilidad de métodos y atributos

Accesos que permiten las palabras reservadas `public`, `protected` y `private` en métodos y propiedades, llamados modificadores de acceso (o ámbito):

- **Private:**
 - Desde la misma clase que declara
- **Protected:**
 - Desde la misma clase que declara
 - Desde las clases que heredan esta clase
- **Public:**
 - Desde la misma clase que declara
 - Desde las clases que heredan esta clase
 - Desde cualquier elemento fuera de la clase (desde los objetos)

Encapsulación

- Las propiedades se definen privadas o protegidas (si queremos que las clases heredadas puedan acceder).
- Para cada propiedad, se añaden métodos públicos (getter/setter):

```
public setPropiedad(tipo $param)

public getPropiedad() : tipo
```
- Las constantes se suelen definir como públicas para que sean accesibles por todos los recursos.
- Instalar en **Visual Studio Code** la extensión:

PHP 8 Getter and Setter



```
<?php
class MayorMenor
{
    private int $mayor;
    private int $menor;

    /**
     * Get the value of mayor
     * @return int
     */
    public function getMayor(): int
    {
        return $this->mayor;
    }

    /**
     * Set the value of mayor
     * @param int $mayor
     * @return self
     */
    public function setMayor(int $mayor): self
    {
        $this->mayor = $mayor;
        return $this;
    }

    /**
     * Get the value of menor
     * @return int
     */
    public function getMenor(): int
    {
        return $this->menor;
    }

    /**
     * Set the value of menor
     * @param int $menor
     * @return self
     */
    public function setMenor(int $menor): self
    {
        $this->menor = $menor;
        return $this;
    }
}
```

Constructor

El constructor de los objetos se define mediante el método mágico `__construct`:

- Es un método **opcional**
- Puede o no tener parámetros
- **Solo puede haber un único constructor.**
- El constructor se ejecuta inmediatamente después de crear un objeto y no puede ser llamado nuevamente.
- Un constructor **no puede devolver datos.**
- Un constructor puede recibir parámetros que se utilizan normalmente para inicializar atributos.
- Para definir el constructor `__construct` (utilizamos dos caracteres de subrayado y la palabra `construct`).
- El constructor debe ser un método público (**public function**).

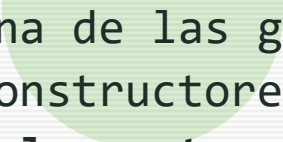
Constructor



```
<?php
class Persona
{
    private string $nombre;

    public function __construct(string $nom)
    {
        $this->nombre = $nom;
    }
    public function imprimir()
    {
        echo $this->nombre;
        echo '<br>';
    }
}
$bruno = new Persona("Bruno Díaz");
$bruno->imprimir();
?>
```

Constructores en PHP 8




Una de las grandes novedades que ofrece PHP 8 es la simplificación de los constructores con parámetros, lo que se conoce como **promoción de las propiedades del constructor**.

Para ello, en vez de tener que declarar las propiedades como privadas o protegidas, y luego dentro del constructor tener que asignar los parámetros a estas propiedades, el propio constructor promociona las propiedades.

Imaginemos una clase Punto donde queramos almacenar sus coordenadas:

Constructor en PHP 8



```
//Anterior a PHP 8
<?php
class Punto
{
    protected float $x;
    protected float $y;
    protected float $z;

    public function __construct(
        float $x = 0.0,
        float $y = 0.0,
        float $z = 0.0
    ) {
        $this->x = $x;
        $this->y = $y;
        $this->z = $z;
    }
}
```

```
//A partir de PHP 8
<?php
class Punto
{
    public function __construct(
        protected float $x = 0.0,
        protected float $y = 0.0,
        protected float $z = 0.0,
    ) {
    }
}
```

El orden importa



A la hora de codificar el orden de los elementos debe ser:

```
<?php
declare(strict_types=1);

class NombreClase {
    // propiedades

    // constructor

    // getters - setters

    // resto de métodos
}
?>
```



Llamada de métodos y propiedades no estáticas: dentro y fuera de la clase

Si llamamos a los métodos desde un objeto de la clase. La sintaxis es la siguiente:

nombre del objeto->nombre del método

Antecedemos al nombre del método el nombre del objeto y el operador ->

Si queremos llamar dentro de la clase a un método que pertenece a la misma clase, la sintaxis es la siguiente:

\$this->nombre del método

Es importante tener en cuenta que esto solo se puede hacer cuando estamos dentro de la misma clase.

Declaración de una clase y creación de un objeto

```
<?php

class Coche
{
    public $ruedas;
    private $motor;
    //Poner como modificador var está obsoleto.
    //var $color;
    function __construct()
    {
        $this->ruedas = 4;
        $this->motor = 1600;
        $this->color = "";
    }
    function arrancar(){
        echo "estoy arrancando<br>";
    }
    static function girar() {
        echo "estoy en girar <br>";
    }
    function frenar(){
        echo "estoy frenando <br>";
    }
    function poner_color($pintar){
        $this->color = $pintar;
    }
    function elegir_motor($motor)
    {
        $this->motor= $motor;
    }
}
```

```
$micoche = new Coche;
$mazda = new Coche();

$mazda->girar();
$mazda->frenar();

echo "el coche tiene " . $mazda->ruedas . " ruedas<br>";
$micoche->poner_color("rojo");

//$mazda->motor=8; Esta línea daría un error de acceso
//El atributo motor es private y sólo se puede acceder a
él desde dentro de la clase.

$micoche->elegir_motor(2000);

var_dump($micoche);
?>
```

Clases estáticas

Son aquellas clases que contienen propiedades y/o métodos estáticos (también se conocen como métodos o propiedades de clase, porque su valor se comparte entre todas las instancias de la misma clase).

Se declaran con **static** y se referencian con el operador **::**

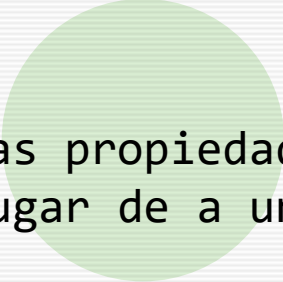
- Si queremos acceder a un método estático se antepone el **nombre de la clase**:

`Producto::nuevoProducto()`

- Si desde un método queremos acceder a una propiedad estática de la misma clase, se utiliza la referencia **self**

`self::$numProductos`

Propiedades y clases estáticas



Las propiedades y métodos estáticos en PHP pertenecen a la clase en sí misma en lugar de a una instancia específica de la clase.

Puedes acceder a ellos sin crear una instancia de la clase.

Las propiedades y métodos estáticos son útiles para almacenar datos o comportamientos que son comunes a todas las instancias de una clase.

Propiedades Estáticas

- **Definición:** Se declaran con la palabra clave `static`.
- **Acceso:** Se accede a ellas usando el operador de resolución de ámbito `::`
- **Comportamiento:**
 - Todas las instancias de la clase comparten el mismo valor para una propiedad estática.
 - Su valor puede ser modificado a diferencia de las constantes cuyo valor es inmutable.

Propiedades y clases estáticas

En PHP, tanto `const` como `static` se utilizan para definir miembros de clase que no están asociados con instancias específicas de la clase.

Ambos tipos de miembros se comparten entre todas las instancias de la clase y se accede a ellos de manera similar, pero su comportamiento y uso son diferentes.

`const`

- **Inmutabilidad:** Una constante no puede cambiar su valor una vez definida.
- **Acceso:** Se accede a las constantes de clase usando el operador de resolución de ámbito `::`
- **Visibilidad:** Las constantes son siempre públicas y no pueden tener modificadores de visibilidad (`public`, `protected`, `private`).

`static`

- **Mutabilidad:** Una propiedad estática puede cambiar su valor durante la ejecución del programa.
- **Acceso:** Se accede a las propiedades y métodos estáticos usando el operador de resolución de ámbito `::`
- **Visibilidad:** Las propiedades y métodos estáticos pueden tener modificadores de visibilidad (`public`, `protected`, `private`).

Inicialización de propiedades estáticas y constantes

En PHP, las propiedades estáticas solo pueden ser inicializadas utilizando un valor constante como:

- un string literal
- un número
- una constante definida

No se pueden utilizar expresiones o valores calculados para inicializar propiedades estáticas.

```
class Ejemplo {  
    public static $propiedadEstatica = 'Valor inicial'; // Correcto  
    public static $numero = 42; // Correcto  
    public static $constante = self::CONSTANTE; // Correcto si CONSTANTE es una constante de clase  
    const CONSTANTE = 'Valor constante';  
}
```

```
class Ejemplo {  
    public static $propiedadEstatica = 'Valor inicial' . ' concatenado'; // Incorrecto  
    public static $numero = 21 * 2; // Incorrecto  
}
```

Inicialización de propiedades estáticas y constantes



Si se necesita inicializar una propiedad estática **con un valor calculado**, se puede hacer dentro de un método estático.

```
<?php
class Ejemplo
{
    // Propiedad estática
    public static string $propiedadEstatica;

    // Constructor
    public function __construct()
    {
        // Llamar al método estático de inicialización
        self::inicializar();
    }

    // Método estático de inicialización
    public static function inicializar(): void
    {
        self::$propiedadEstatica = 'Valor inicial' . ' concatenado';
    }

    // Método estático para obtener el valor de la propiedad estática
    public static function obtenerPropiedadEstatica(): string
    {
        return self::$propiedadEstatica;
    }
}

// Crear una instancia de la clase Ejemplo
$ejemplo = new Ejemplo();

// La propiedad estática se inicializa en el constructor
echo Ejemplo::obtenerPropiedadEstatica(); // Salida: Valor inicial concatenado
?>
```

Inicialización de propiedades estáticas y constantes

Las constantes en PHP pueden ser inicializadas con cualquier valor que sea una expresión constante, lo que incluye:


- valores literales
- constantes definidas
- expresiones que se evalúan en tiempo de compilación no en la ejecución.

Las constantes no pueden ser inicializadas con valores que dependen de la ejecución del programa, como el resultado de una función.

```
class Ejemplo
{
    const CONSTANTE1 = 'Valor constante'; // Correcto
    const CONSTANTE2 = 42; // Correcto
    const CONSTANTE3 = self::CONSTANTE1 . ' concatenado'; // Correcto
    const CONSTANTE4 = 21 * 2; // Correcto
}
```

```
class Ejemplo {
    const CONSTANTE1 = time(); // Incorrecto
    const CONSTANTE2 = rand(1, 10); // Incorrecto
}
```

Clases estáticas



```
<?php
class Producto
{
    const IVA = 0.23;
    private static $numProductos = 0;

    public static function nuevoProducto()
    {
        self::$numProductos++;
    }
}

Producto::nuevoProducto();
$impuesto = Producto::IVA;
```



```
<?php
class Producto
{
    // Constante para el IVA (valor fijo)
    const IVA = 0.23;

    // Propiedad estática para contar el número total de productos
    private static int $numProductos = 0;

    // Propiedad estática para contar cuántos productos hay de cada tipo
    private static array $productosPorTipo;

    // Propiedad para el nombre del producto (cada instancia tendrá su propio nombre)
    private string $nombre;

    // Constructor de la clase Producto
    public function __construct(string $nombre)
    {
        $this->nombre = $nombre;

        // Incrementamos el número total de productos
        self::$numProductos++;

        // Incrementamos el número de productos de este tipo
        if (isset(self::$productosPorTipo[$nombre])) {
            self::$productosPorTipo[$nombre]++;
        } else {
            self::$productosPorTipo[$nombre] = 1;
        }
        echo "Producto '$this->nombre' creado. Número total de productos: " . self::$numProductos . "<br>";
    }
}
```

```
//Método estático de inicialización array de productos por tipo
public static function inicializarTipo(): void
{
    self::$productosPorTipo = [];
}
//Método estático para inicializar el número de productos
public static function inicializarProductos(): void
{
    self::$numProductos = 0;
}

// Método estático para obtener el número total de productos
public static function getNumProductos(): int
{
    return self::$numProductos;
}

// Método estático para obtener cuántos productos hay de cada tipo
public static function getProductosPorTipo(): array
{
    return self::$productosPorTipo;
}

// Método para obtener el nombre del producto (esto es para cada objeto individual)
public function getNombre(): string
{
    return $this->nombre;
}
}
```

```
// Mostramos el valor de la constante IVA
echo "El IVA es: " . Producto::IVA . "<br><br>";

// Creamos varios objetos de la clase Producto
$producto1 = new Producto("Portátil"); // Producto 1
$producto2 = new Producto("Teléfono"); // Producto 2
$producto3 = new Producto("Portátil"); // Producto 3 (otro del mismo tipo)
$producto4 = new Producto("Tablet"); // Producto 4
$producto5 = new Producto("Teléfono"); // Producto 5 (otro del mismo tipo)

// Mostramos el número total de productos
echo "<br>Número total de productos creados: " . Producto::getNumProductos() . "<br>";
// Mostramos el número total de productos desde una instancia
echo "Valor número total de productos en un objeto: " . $producto1->getNumProductos() . "<br>";

// Mostramos cuántos productos hay de cada tipo
echo "<br>Productos por tipo:<br>";
foreach (Producto::getProductosPorTipo() as $tipo => $cantidad) {
    echo "Tipo: $tipo, Cantidad: $cantidad<br>";
}
//Inicicalizar el número de productos de la propiedad estática
Producto::inicializarProductos();
Producto::inicializarTipo();
echo "<br>Número total de productos creados: " . Producto::getNumProductos() . "<br>";
echo "<br>Productos por tipo:<br>";
foreach (Producto::getProductosPorTipo() as $tipo => $cantidad) {
    echo "Tipo: $tipo, Cantidad: $cantidad<br>";
}
```

El IVA es: 0.23

Producto 'Portátil' creado. Número total de productos: 1
 Producto 'Teléfono' creado. Número total de productos: 2
 Producto 'Portátil' creado. Número total de productos: 3
 Producto 'Tablet' creado. Número total de productos: 4
 Producto 'Teléfono' creado. Número total de productos: 5

Número total de productos creados: 5
 Valor número total de productos en un objeto: 5

Productos por tipo:
 Tipo: Portátil, Cantidad: 2
 Tipo: Teléfono, Cantidad: 2
 Tipo: Tablet, Cantidad: 1

Número total de productos creados: 0

Productos por tipo:

Creación de clases

En PHP, es una buena práctica colocar cada clase en un archivo separado. Esto facilita la organización del código y permite utilizar la carga automática de clases (autoloader).

Ventajas de colocar cada clase en un archivo separado

Organización: Mantener cada clase en su propio archivo hace que el código sea más fácil de mantener y entender.

Reutilización: Facilita la reutilización de clases en diferentes proyectos.

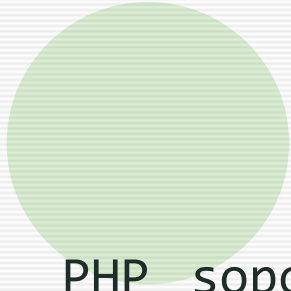
Carga Automática: Permite utilizar la carga automática de clases, lo que simplifica la inclusión de archivos.

Carga Automática de Clases (autoloader)

La carga automática de clases es una característica que permite cargar automáticamente las clases cuando se necesitan, sin necesidad de usar `include` o `require` manualmente.

Nombres de archivos y clases: Deben ser idénticos para permitir la carga automática.

Herencia



PHP soporta **herencia simple**, de manera que una clase solo puede heredar de otra, no de dos clases a la vez. PHP no permite herencia múltiple

Para ello se utiliza la palabra clave **extends**.

Si queremos que la clase A herede de la clase B haremos:

```
class A extends B
```

El hijo hereda los atributos y métodos públicos y protegidos.

Herencia

PRUEBACLASES

Producto.php

Tv.php

```
<?php

class Producto
{
    public $codigo;
    public $nombre;
    public $nombreCorto;
    public $PVP;

    public function mostrarResumen()
    {
        echo "<p>Producto: " . $this->codigo . "</p>";
    }
}
```

```
<?php

require_once "Producto.php";

class Tv extends Producto
{
    public $pulgadas;
    public $tecnologia;
}
```

Constructor en hijos

1. Constructores en clases hijas no se crean automáticamente

Cuando una clase hija hereda de una clase padre, PHP no crea automáticamente un **constructor** para la clase hija. Esto significa que si no defines un constructor en la clase hija, la clase hija no tendrá su propio constructor.

2. Si la clase hija no tiene constructor, se ejecuta el del padre

Si no defines un constructor en la clase hija, PHP ejecutará el **constructor de la clase padre**, **si la clase padre tiene uno**. Este comportamiento asegura que, si heredas de una clase que tiene un constructor, ese constructor se ejecutará, lo que es útil cuando la clase padre tiene inicializaciones importantes que la clase hija debe respetar.

3. Si defines un constructor en la clase hija

Cuando defines un constructor en la clase hija, PHP ejecuta ese constructor y no el del padre de manera automática. En este caso, el constructor de la clase padre no se ejecuta a menos que lo hagas explícitamente.

4. Para invocar el constructor del padre desde la clase hija, debes usar el operador **parent::**

```
public function __construct() {  
    parent::__construct();  
}
```

Constructor en hijos

```
<?php
class Producto
{
    public string $codigo;

    public function __construct(string $codigo)
    {
        $this->codigo = $codigo;
    }

    public function mostrarResumen()
    {
        echo "<p>Prod:" . $this->codigo . "</p>";
    }
}
```

```
<?php
include_once "Producto.php";

class Tv extends Producto
{
    public $pulgadas;
    public $tecnologia;

    public function __construct(string $codigo, int $pulgadas,
    string $tecnologia)
    {
        parent::__construct($codigo);
        $this->pulgadas = $pulgadas;
        $this->tecnologia = $tecnologia;
    }

    public function mostrarResumen()
    {
        parent::mostrarResumen();
        echo "<p>TV " . $this->tecnologia . " de " . $this->pulgadas . "</p>";
    }
}
```


Sobreescritura de métodos



- En una clase hija se puede redefinir un método, es decir que podemos crear un método con el mismo nombre que el método de la clase padre. Ahora cuando creamos un objeto de la subclase, el método que se llamará es el de dicha subclase.
- Lo más conveniente es sobrescribir métodos para completar el algoritmo del método de la clase padre. No es bueno sobrescribir un método y cambiar completamente su comportamiento.

Palabra reservada parent::

- Cuando queramos acceder a una constante o método de una clase padre, la palabra reservada **parent** nos sirve para llamarla desde una clase hija.
- Cuando en una clase hija se reescriba el mismo método de una clase padre eliminando las definiciones y cambiando su visibilidad del método de la clase padre, para acceder al método original de la clase padre se debe utilizar la palabra reservada **parent::**

Palabra reservada parent y operador de resolución de ámbito de clases (doble ::)

```
<?php
class ClasePadre
{
    public $titulo = "Programación en PHP 8";
    protected function miFuncion()
    {
        echo 'Método Padre <br>';
    }
}
class ClaseExtendida extends ClasePadre
{
    public function miFuncion()
    {
        echo '<h1>Método hijo</h1><br>';
        echo $this->$titulo;
        parent::miFuncion();
    }
}
$hiijo=new ClaseExtendida();
$hiijo->miFuncion();
?>
```

Parámetros opcionales



Tanto para programación funcional como para programación orientada a objetos.

Un parámetro es opcional si en la declaración del método le asignamos un valor por defecto. Si después, llamamos al método sin enviarle dicho valor tomará el que tiene por defecto.

Podemos definir parámetros opcionales tanto para el constructor como para cualquier otro método de la clase.

Los parámetros opcionales nos permiten desarrollar clases que sean más flexibles en el momento que definimos objetos de las mismas.

```
<?php
class Persona
{
    private $nombre;
    private $edad;
    public function __construct($nombre, $edad = null)
    {
        $this->nombre = $nombre;
        $this->edad = $edad;
    }
    public function presentarse()
    {
        if ($this->edad !== null) {
            echo "Hola, soy $this->nombre y tengo {$this->edad} años.";
        } else {
            echo "Hola, soy {$this->nombre}.";
        }
    }
}

// Crear una instancia de la clase
$persona1 = new Persona("Juan");
$persona2 = new Persona("Maria", 30);
// Llamar al método presentarse
$persona1->presentarse(); // Salida: Hola, soy Juan.
$persona2->presentarse(); // Salida: Hola, soy Maria y tengo 30 años.
```

Métodos encadenados

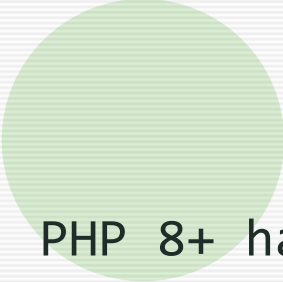


- Métodos encadenados también se conoce como *method chaining*.
- Los métodos encadenados en PHP permiten llamar múltiples métodos de un objeto en una sola línea de código.
- Esto se hace retornando el propio objeto (`$this`) dentro de los métodos de una clase, lo que permite seguir llamando otros métodos de la misma instancia de forma consecutiva.

Métodos encadenados. Antes de PHP 8

- **Uso de self**
 - Retorno de self: Siempre retorna una instancia de la clase donde se definió el método. Si el método está en Coche, retornará una instancia de Coche, incluso si lo llamas desde una subclase.
 - Encadenamiento en Subclases: No permite el encadenamiento fluido en subclases si el método está definido en la clase base.
- **Uso del Nombre de la Clase (por ejemplo, Coche)**
 - Retorno del Nombre de la Clase: Retorna una instancia de la clase base (Coche) o de cualquier subclase que herede de esa clase (como CocheDeportivo). Esto hace que sea más flexible cuando se trabaja con herencia, ya que puede devolver una instancia de la subclase si es llamada desde allí.
 - Encadenamiento en Subclases: Permite el encadenamiento fluido en subclases.

Métodos encadenados

- 
- PHP 8+ ha mejorado el soporte para retornos en métodos encadenados.
 - Usar `self` en métodos de la clase base no causará problemas cuando se trabaja con subclases, ya que PHP entiende que debe retornar una **instancia de la clase que llama al método**, no necesariamente de la clase base.


```
<?php
class Coche
{
    private $marca;
    private $modelo;
    private $color;
    public function setMarca(string $marca): Coche
    {
        $this->marca = $marca;
        return $this;
    }
    public function setModelo(string $modelo): Coche
    {
        $this->modelo = $modelo;
        return $this;
    }
    public function setColor(string $color): self
    {
        $this->color = $color;
        return $this;
    }
    public function getMarca(): string
    {
        return $this->marca;
    }
    public function getModelo(): string
    {
        return $this->modelo;
    }
    public function getColor(): string
    {
        return $this->color;
    }
}
$coche = new Coche();
$coche->setMarca('Ford')->setModelo('Mustang')->setColor('Rojo');

echo 'Marca: ' . $coche->getMarca() . '<br>';
echo 'Modelo: ' . $coche->getModelo() . '<br>';
echo 'Color: ' . $coche->getColor() . '<br>';
```

```
class CocheDeportivo extends Coche
{
    private int $velocidadMaxima;

    public function setVelocidadMaxima(int $velocidad): self
    {
        $this->velocidadMaxima = $velocidad;
        return $this;
    }

    public function getVelocidadMaxima(): int
    {
        return $this->velocidadMaxima;
    }
}

$cocheDeportivo = new CocheDeportivo();
$cocheDeportivo->setMarca('Ferrari')->setModelo('488')->setColor('Rojo')->setVelocidadMaxima(330);


echo 'Marca: ' . $cocheDeportivo->getMarca() . '<br>';
echo 'Modelo: ' . $cocheDeportivo->getModelo() . '<br>';
echo 'Color: ' . $cocheDeportivo->getColor() . '<br>';
echo 'Velocidad Máxima: ' . $cocheDeportivo->getVelocidadMaxima() . ' km/h<br>';
echo $cocheDeportivo instanceof CocheDeportivo ? 'Es un coche deportivo' : 'No es un coche deportivo';
```

Marca: Ferrari Modelo: 488 Color: Rojo Velocidad Máxima: 330 km/h
--

Métodos encadenados



- Si no indicamos qué devuelve el método (self o Clase)
 - El encadenamiento de métodos funcionará porque los métodos devuelven \$this.
- No habrá verificación de tipo en tiempo de compilación, lo que puede llevar a errores difíciles de detectar si los métodos no devuelven lo que se espera.



```
<?php
class Libro {
    private string $nombre;
    private string $autor;

    public function getNombre() : string {
        return $this->nombre;
    }
    public function setNombre(string $nombre) : self {
        $this->nombre = $nombre;
        return $this;
    }

    public function getAutor() : string {
        return $this->autor;
    }
    public function setAutor(string $autor) : Libro {
        $this->autor = $autor;
        return $this;
    }

    public function __toString() : string {
        return $this->nombre." de ".$this->autor;
    }
}

$p1 = new Libro();
$p1->setNombre("Harry Potter");
$p1->setAutor("JK Rowling");
echo $p1; //Usando método mágico __toString()
// Method chaining
$p2 = new Libro();
$p2->setNombre("Patria")->setAutor("Aramburu");
echo $p2; //Usando método mágico __toString()
```

Ejercicio

Crear una **clase Persona** que tenga como atributos el **nombre** y la **edad**. El constructor recibe los datos para inicializar dichos atributos. Y un método **mostrar()** que imprima muestre nombre y edad.

Crear una segunda **clase Empleado** que herede de la clase Persona y añadir un atributo **sueldo**. El constructor recibe los tres atributos de la **clase Empleado**. Llamar al constructor de la clase padre para inicializar los atributos nombre y edad del Empleado. Sobre escribir el método **mostrar()** de la clase padre para que muestre también el sueldo.

Definir un objeto de la clase Persona y llamar a sus métodos. También crear un objeto de la clase Empleado y llamar a sus métodos.

Cuando definamos clases debemos guardarlas en un archivo que tenga el mismo nombre de la clase en mayúsculas y extensión php. Ejemplo **Persona.php**

El objeto persona mostrará

Nombre: Rodríguez Pablo

Edad: 24

El objeto empleado mostrará

Nombre: González Ana

Edad: 32

Sueldo: 2400€

Acceder a miembros de Clases y Objetos

Las tres palabras reservadas *\$this*, *self* y *parent* y son utilizadas para acceder a propiedades y métodos desde el interior de la definición de la clase.

Para acceder a los miembros de una Clase usaremos las siguientes **palabras reservadas**:

\$this-> En PHP \$this es usada como una *pseudo-variable* para hacer referencia al contexto actual del objeto. Se usa únicamente para acceder a las propiedades y métodos **no estáticos** en la propia Clase.

self:: acceder a las propiedades y métodos definidas como const o static en la propia Clase.

parent:: acceder a las propiedades definidas como const o static de la Clase padre, así como a sus métodos (aunque no sean de tipo static).

:: El Operador de Resolución de Ámbito (también denominado Paamayim Nekudotayim, significa doble dos-puntos en hebreo), el doble dos-puntos, permite acceder a elementos estáticos, constantes, y sobrescribir propiedades o métodos de una clase.

Parse error: syntax error, unexpected T_PAAMAYIM_NEKUDOTAYIM in file on line ...

Resumen: Métodos y propiedades estáticos static

Las propiedades y métodos estáticos son accesibles sin la necesidad de instanciar la clase.

Las propiedades estáticas sólo pueden ser inicializadas utilizando un string literal o una constante; las expresiones no están permitidas.

Por tanto, se puede inicializar una propiedad estática con enteros o arrays (por ejemplo), pero no se puede inicializar con el contenido de otra variable, con el valor de devolución de una función, o con un objeto.

NO se puede acceder a una propiedad static con un objeto instanciado, aunque sí a un método static.

No es posible usar la pseudo-variable \$this, ya que los métodos estáticos pueden ejecutarse sin tener una instancia del objeto, y \$this hace referencia al objeto creado.

Debemos utilizar `self::` acceder a las propiedades y métodos definidas como const o static en la propia Clase.

Antepondremos el nombre de la clase seguido de :: para referirnos a constantes y propiedades y métodos estáticos fuera de la clase.

Métodos y propiedades static

Advertencia

En PHP 8, llamar a métodos **NO** estáticos de forma estática está obsoleto y generará un ERROR.

Funciones para obtener información de los objetos

Al trabajar con clases y objetos, existen un conjunto de funciones ya definidas por el lenguaje que permiten obtener información sobre los objetos:

- **instanceof**: permite comprobar si un objeto es de una determinada clase. Devuelve **true** si el objeto es una instancia de la clase o subclase especificada, y **false** en caso contrario
- **get_class**: devuelve el nombre de la clase
- **get_declared_class**: devuelve un array con los nombres de las clases definidas
- **class_alias**: crea un alias
- **class_exists** / **method_exists** / **property_exists**: devuelve **true** si la clase / método / propiedad está definida
- **get_class_methods** / **get_class_vars** / **get_object_vars**: Devuelve un array con los nombres de los métodos / propiedades de una clase / propiedades no estáticas de un objeto que son accesibles desde dónde se hace la llamada.
- **get_parent_class(object): string** devuelve el nombre de la clase padre de un objeto o clase. Si el objeto no tiene una clase padre o la clase dada no existe, se devuelve **false**
- **is_subclass_of(object, string): bool** Devuelve **true** si el objeto pertenece a una clase que es una subclase de la clase dada, y **false** en caso contrario

```
<?php
class Artículo
{
    const IVA = 0.23;
    public static $numProductos = 0;
    public $codigo;

    public function __construct(string $cod)
    {
        self::$numProductos++;
        $this->codigo = $cod;
    }

    public function mostrarResumen(): string
    {
        return "El producto " . $this->codigo . " es el número " . self::$numProductos;
    }
}

$p = new Artículo("PS5");

//Compruebo si un objeto es instancia de una clase
if ($p instanceof Artículo) {
    echo "<br>Es un objeto y es de la clase ". get_class($p)."<br>";

    //Creo un alias de la clase Artículo
    class_alias("Artículo", "AliasArtículo");
    //Creo un objeto usando el alias
    $c = new AliasArtículo("Nintendo Switch");
    echo "<br>Un artículo es un " . get_class($c);

    print_r(get_class_methods("Artículo"));
    print_r(get_class_vars("Artículo"));
    print_r(get_object_vars($p));

    if (method_exists($p, "mostrarResumen")) {
        echo $p->mostrarResumen();
    }
}
```

Clases Abstractas

Se define mediante `abstract class NombreClase {}`

Características:

- Una clase abstracta no puede ser instanciada, no podremos crear objetos a partir de ellas. Pero sí llamar a sus métodos estáticos, si los tuviese.
- Una clase abstracta es una clase que cuenta con al menos un método abstracto.
- Una clase abstracta puede incorporar uno o varios métodos abstractos y puede contener métodos implementados.
- Los métodos abstractos son aquellos que solo existe su declaración, dejando su implementación a las futuras clases extendidas o derivadas.
- Los métodos abstractos se definen: `abstract function nombreMetodo();`
- Cuando se hereda de una clase abstracta, todos los métodos definidos como abstractos en la declaración de la clase padre deben ser implementados en la clase hija.
- Los métodos deben ser definidos con la misma o con una visibilidad menos restrictiva.
- Por ejemplo, si el método abstracto está definido como `protected`, la implementación de la función debe ser definida como `protected` o `public`, pero nunca como `privada`.

Clases Abstractas



Los **prototipos de los métodos tienen que coincidir**: la declaración de tipos y el número de argumentos requeridos deben ser los mismos.

El método abstracto sólo necesita definir los argumentos requeridos.

Si la clase derivada define un argumento opcional y el prototipo del método abstracto no lo hace, no habría conflicto.

La clase derivada puede definir parámetros opcionales que no estén en la estructura del prototipo

```
<?php
// Definición de la clase abstracta FiguraGeometrica
abstract class FiguraGeometrica {
    // Método abstracto para calcular el área
    abstract public function calcularArea();
    // Método abstracto para calcular el perímetro
    abstract public function calcularPerimetro();
}

// Definición de la clase hija Circulo que hereda de FiguraGeometrica
class Circulo extends FiguraGeometrica {
    // Propiedad para almacenar el radio del círculo
    private $radio;
    // Constructor de la clase Circulo
    public function __construct($radio) {
        $this->radio = $radio;
    }
    // Implementación del método para calcular el área de un círculo
    public function calcularArea() {
        return pi() * pow($this->radio, 2);
    }
    // Implementación del método para calcular el perímetro de un círculo
    public function calcularPerimetro() {
        return 2 * pi() * $this->radio;
    }
}
```



```
// Crear una instancia de la clase Circulo
$miCirculo = new Circulo(5);
// Calcular el área y el perímetro del círculo
$area = $miCirculo->calcularArea();
$perimetro = $miCirculo->calcularPerimetro();
// Imprimir los resultados
echo "Área del círculo: $area\n";
echo "Perímetro del círculo: $perimetro\n";
```

Clases Finales

Clases finales

Son clases opuestas a las abstractas, ya que evitan:

1. Que se pueda heredar una clase.
2. O sobrescribir un método.

```
<?php
class Producto
{
    private $codigo;

    public function getCodigo(): string
    {
        return $this->codigo;
    }

    final public function mostrarResumen(): string
    {
        return "Producto " . $this->codigo;
    }
}

// No podremos heredar de Microondas
final class Microondas extends Producto
{
    private $potencia;

    public function getPotencia(): int
    {
        return $this->potencia;
    }

    // No podemos implementar mostrarResumen()
}
```

Interfaces



Una interface es un conjunto de declaraciones de métodos sin incluir su codificación, dejando a la clase que implementa la interfaz esta tarea. Una interfaz provee una buena forma de asegurarse que objeto contiene métodos determinados.

Las interfaces se definen de la misma manera que una clase, aunque reemplazando la palabra reservada **class** por la palabra reservada **interface** y sin que ninguno de sus métodos tenga su contenido definido.

```
interface NombreInterfaz{}
```


Interfaces



- Una interfaz no se puede instanciar y todos sus métodos son públicos dada la propia naturaleza de la interfaz.
- Una interfaz no puede contener ni atributos, ni métodos implementados, solo declaraciones de métodos y **constantes**.

Interfaces

Las interfaces pueden heredar de una o varias interfaces mediante el operador **extends**.

```
interface c extends a, b {}
```

Una clase puede implementar uno o varios interfaces mediante el operador **implements**. Soporta herencia múltiple pero solo para interfaces.

```
class d implements a,b {}
```

Interfaces

```
<?php
// Definición de la interfaz Mostrable
interface Mostrable
{
    public function mostrarResumen(): string;
}

// Definición de la interfaz MostrableTodo que extiende la interfaz Mostrable
interface MostrableTodo extends Mostrable
{
    public function mostrarTodo(): string;
}

// Definición de la interfaz Facturable
interface Facturable
{
    public function generarFactura(): string;
}
```

Interfaces

```
class Producto implements MostrableTodo, Facturable
{
    private $nombre;
    private $precio;

    public function __construct($nombre, $precio)
    {
        $this->nombre = $nombre;
        $this->precio = $precio;
    }

    public function mostrarResumen(): string
    {
        return "Producto: {$this->nombre}, Precio: {$this->precio}";
    }

    public function mostrarTodo(): string
    {
        return $this->mostrarResumen();
    }

    public function generarFactura(): string
    {
        return "Factura para el producto {$this->nombre}: {$this->precio} €";
    }
}
```

Interfaces

```
// Crear una instancia de Producto

$miProducto = new Producto("Ejemplo de Producto", 50);

// Llamar a los métodos de las interfaces

echo $miProducto->mostrarResumen() . "\n";

echo $miProducto->mostrarTodo() . "\n";

echo $miProducto->generarFactura() . "\n";

?>
```

Trait

- Los traits (rasgos) son un mecanismo de reutilización de código en lenguajes de herencia simple como PHP.
- El objetivo de un trait es el de reducir las limitaciones propias de la herencia simple.
- Los traits permiten definir métodos e implementar métodos que pueden ser utilizados por múltiples clases sin requerir herencia.
- Los traits son “*similares*” a las clases abstractas, pero con los métodos implementados.
- Se declara utilizando:

```
trait NombreTrait {}
```

- Hay que utilizar la palabra reservada `use` para incorporar uno o varios trait a una clase.

```
class UnaClase
{
    use UnTrait, OtroTrait;
    //Código de la clase
}
```

Trait

```
<?php
trait Registrable
{
    public function registrar()
    {
        echo "Usuario registrado: {$this->nombre}<br>";
    }
}
class Usuario
{
    private $nombre;

    public function __construct($nombre)
    {
        $this->nombre = $nombre;
    }

    public function saludar()
    {
        echo "¡Hola, soy {$this->nombre}!<br>";
    }
}

// La clase Usuario utiliza el trait Registrable
class UsuarioRegistrado extends Usuario
{
    use Registrable;
}

// Crear una instancia de UsuarioRegistrado
$usuario = new UsuarioRegistrado("EjemploUsuario");
$usuario->saludar(); // Método propio de la clase Usuario
$usuario->registrar(); // Método proporcionado por el trait Registrable
```

Trait : Resolución de conflictos

- ¿Qué pasa si la clase padre, el Trait y la clase actual tienen un método con el mismo nombre?, ejemplo:

Clase User método getName()

Clase padre Person método getName()

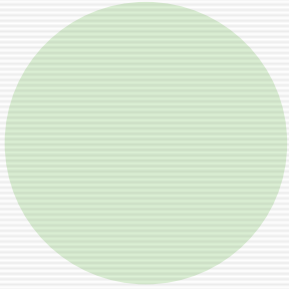
Trait UnTrait método getName()

- Prevalece el método de la clase actual;
- Se revisa de esta manera:
 - (1) clase,
 - (2) trait,
 - (3) clase padre.

Trait: Resolución de conflictos

- Cuando se usan múltiples traits es posible que haya diferentes traits que usen los mismos nombres de métodos.
- Si dos Traits insertan un método con el mismo nombre, se produce un error fatal, siempre y cuando no se haya resuelto explícitamente el conflicto.
- Para resolver los conflictos de nombres entre Traits en una misma clase, se debe usar el operador **insteadof** para elegir unívocamente uno de los métodos conflictivos.
- Como esto solamente permite excluir métodos, se puede utilizar el operador **as** para añadir un alias a uno de los métodos.
- El operador **as** no renombra el método el método original.

Métodos Mágicos



`__construct()` y `__destruct()`

`__get()` y `__set()`

`__toString()`

Métodos Mágicos

`__construct()` y `__destruct()`

- El método mágico más utilizado en PHP es `__construct()`, un método que es llamado automáticamente cuando se crea el objeto. Permite inicializar parámetros para construir el objeto.
- A partir de PHP 5 se introduce un concepto de destructor similar al de otros lenguajes orientados a objetos. El método destructor será llamado tan pronto como no haya referencias a un objeto determinado, o en cualquier otra circunstancia de finalización, como por ejemplo un `exit()`;
- `__destruct()` no se suele implementar ya que el compilador de PHP se encarga de eliminar el objeto al finalizar el script.

Métodos Mágicos

`__toString()`

Representación del objeto como cadena. Se ejecuta automáticamente este método cuando intentamos imprimir un objeto, es decir, cuando hacemos `echo $objeto` .

`__get(propiedad), __set(propiedad, valor)`

Permitirían acceder a las propiedades privadas de un objeto.

Siempre es más legible/mantenible codificar los getter/setter.

Se lanzará automáticamente, si están implementados y se intenta acceder a propiedades privadas desde un objeto.

Métodos Mágicos

```
<?php
class MiClase
{
    private $datos = [];
    // Método mágico __set para asignar valores a propiedades inaccesibles
    public function __set($nombre, $valor)
    {
        $this->datos[$nombre] = $valor;
    }

    // Método mágico __get para obtener valores de propiedades inaccesibles
    public function __get($nombre)
    {
        if (isset($this->datos[$nombre])) {
            return $this->datos[$nombre];
        }
        return null;
    }

    // Método mágico __toString para representar la clase como una cadena
    public function __toString()
    {
        //print_r con true no imprime en pantalla sino devuelve un string
        return "MiClase: " . print_r($this->datos, true);
    }
}
```

Métodos Mágicos

```
$miObjeto = new MiClase();

// Utilizamos el método mágico __set para asignar valores a propiedades inaccesibles

$miObjeto->nombre = "Ejemplo";

$miObjeto->edad = 30;


// Utilizamos el método mágico __get para obtener los valores de propiedades
inaccesibles

echo "Nombre: " . $miObjeto->nombre . "<br>"; // Imprime "Nombre: Ejemplo"

echo "Edad: " . $miObjeto->edad . "<br>"; // Imprime "Edad: 30"

// Utilizamos el método mágico __toString para representar el objeto como una cadena

echo $miObjeto; // Imprime "MiClase: Array ( [nombre] => Ejemplo [edad] => 30 )"
```