**CONCEPTS POO PHP**

## Class

The basic definition of a class begins with the keyword class, followed by a class name, and continuing with a pair of curly braces that enclose the definitions of the properties and methods belonging to that class.

The class name can be any valid tag, as long as it is not a PHP reserved word. A valid class name begins with a letter or an underscore, followed by an arbitrary number of letters, numbers, or underscores.

```php
<?php
class ClaseSencilla
{
    // Declaración de una propiedad
    public $var = 'un valor predeterminado';

    // Declaración de un método
    public function mostrarVar() {
        echo $this->var;
    }
}
```

## Exceptions:

It has an exception model similar to that of other programming languages. An exception can be thrown ("thrown"), and caught ("catched") within PHP. The code can be inside a try block to make it easier to catch potential exceptions. Every try block must have at least one corresponding catch or finally block.

The syntaxes can be seen in the example below.

```php
<?php
function inverso($x) {
    if (!$x) {
        throw new Exception('División por cero.');
    }
    return 1/$x;
}

try {
    echo inverso(5) . "\n";
    echo inverso(0) . "\n";
} catch (Exception $e) {
    echo 'Excepción capturada: ',  $e->getMessage(), "\n";
}
```

**Data persistence:**

Php has several functions for data persistence.

fopen and fclose: fopen saves in a variable the name of the file where the data will be saved along with the way in which the file is opened whenever it is opened, it must end with an fclose to avoid overwriting errors.

```
fopen(
    string $filename,
    string $mode,
    bool $use_include_path = false,
    resource $context = ?
): resource
```

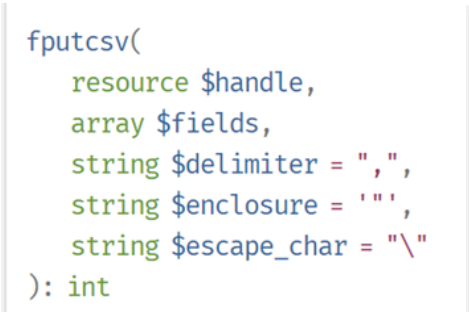
```
fclose(resource $handle): bool
```

**Una lista de los modos posibles de fopen() usando `mode`**

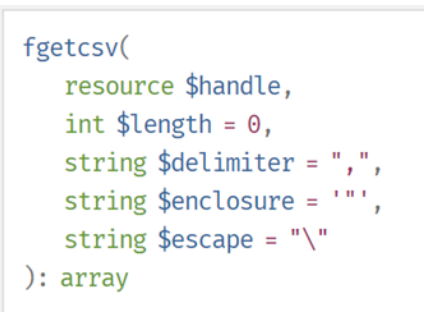
| `mode` | Descripción |
|---|---|
| `'r'` | Apertura para sólo lectura; coloca el puntero al fichero al principio del fichero. |
| `'r+'` | Apertura para lectura y escritura; coloca el puntero al fichero al principio del fichero. |
| `'w'` | Apertura para sólo escritura; coloca el puntero al fichero al principio del fichero y trunca el fichero a longitud cero. Si el fichero no existe se intenta crear. |
| `'w+'` | Apertura para lectura y escritura; coloca el puntero al fichero al principio del fichero y trunca el fichero a longitud cero. Si el fichero no existe se intenta crear. |
| `'a'` | Apertura para sólo escritura; coloca el puntero del fichero al final del mismo. Si el fichero no existe, se intenta crear. En este modo, fseek() solamente afecta a la posición de lectura; las lecturas siempre son pospuestas. |
| `'a+'` | Apertura para lectura y escritura; coloca el puntero del fichero al final del mismo. Si el fichero no existe, se intenta crear. En este modo, fseek() no tiene efecto, las escrituras siempre son pospuestas. |
| `'x'` | Creación y apertura para sólo escritura; coloca el puntero del fichero al principio del mismo. Si el fichero ya existe, la llamada a **fopen()** fallará devolviendo **`false`** y generando un error de nivel **`E_WARNING`**. Si el fichero no existe se intenta crear. Esto es equivalente a especificar las banderas `O_EXCL\|O_CREAT` para la llamada al sistema de `open(2)` subyacente. |
| `'x+'` | Creación y apertura para lectura y escritura; de otro modo tiene el mismo comportamiento que `'x'`. |
| `'c'` | Abrir el fichero para sólo escritura. Si el fichero no existe, se crea. Si existe no es truncado (a diferencia de `'w'`), ni la llamada a esta función falla (como en el caso con `'x'`). El puntero al fichero se posiciona en el principio del fichero. Esto puede ser útil si se desea obtener un bloqueo asistido (véase flock()) antes de intentar modificar el fichero, ya que al usar `'w'` se podría truncar el fichero antes de haber obtenido el bloqueo (si se desea truncar el fichero, se puede usar ftruncate() después de solicitar el bloqueo). |
| `'c+'` | Abrir el fichero para lectura y escritura; de otro modo tiene el mismo comportamiento que `'c'`. |
| `'e'` | Establecer la bandera 'close-on-exec' en el descriptor de fichero abierto. Disponible solamente en PHP compilado en sistemas que se ajustan a POSIX.1-2008. |

If you work with CSV format files, php offers specific functions for it, such as

fgetcsv and fputcsv: fputcsv converts the data given in an array variable into a line in csv format and fgetcsv does the reverse, making it easier to write and read. The syntaxes are as follows.



```
fputcsv(
    resource $handle,
    array $fields,
    string $delimiter = ",",
    string $enclosure = '"',
    string $escape_char = "\"
): int
```



```
fgetcsv(
    resource $handle,
    int $length = 0,
    string $delimiter = ",",
    string $enclosure = '"',
    string $escape = "\"
): array
```

The last 3 parameters of the function are optional and by default they will have the values shown in the images.

### Wrapping:

Isolation protects the properties of an object from being modified by those who do not have the right to access them, only the object's own internal methods can access its state. This ensures that other objects cannot change an object's internal state unexpectedly.

Php offers 3 encapsulation attributes:

- **Public**. The attribute or method preceded by public may be read or called from any part of the code, whether from the same class or outside of it. This means that it may also be altered without restrictions. Therefore, it is recommended that the attributes be declared as private unless exceptional cases.
- **Private**. The attribute or method set to private can only be read from inside the class. Trying to access from outside the class would get an error.
- **Protected.** Protected attributes or methods are similar to private ones, they can only be accessed from inside the class, but with a very useful difference, classes inherited from it will also be able to read and alter these attributes.

### Constructors:

PHP allows developers to declare constructor methods for classes. Those that have a constructor method will call it on each new object created, which makes it ideal for any initialization the object might need before it is used.

```
__construct(mixed ...$values = ""): void
```

Constructors are ordinary methods that are called during the instantiation of their corresponding object. As such, they can define an arbitrary number of arguments, which can be required, can have a type, and can have a default value. Constructor arguments are called by placing the arguments in parentheses after the class name.

```php
<?php
class Point {
    protected int $x;
    protected int $y;

    public function __construct(int $x, int $y = 0) {
        $this->x = $x;
        $this->y = $y;
    }
}

// Pass both parameters.
$p1 = new Point(4, 5);
// Pass only the required parameter. $y will take its default value of 0.
$p2 = new Point(4);
// With named parameters (as of PHP 8.0):
$p3 = new Point(y: 5, x: 4);
?>
```

**GETTERS AND SETTERS IN PHP OOP**

Getter methods are those that return property values. In general, these methods do not receive parameters and must return something, that is, they must always be public.

To have a getter I must have its properties and constructor where they are defined.

This would be an example of what the syntax would be like to make a getter in PHP:

```php
public function get_nombre(){
    return $this->nombre;
}
```

Setter methods are those that allow you to modify the value of a property. These usually receive a parameter with the new value of the property and do not return anything.

These methods typically create one for each property whose value can be changed.

An example of the syntax in PHP would be:

```php
public function set_nombre($nombre){
    $this->nombre = $nombre;
}
```

**ARRANGEMENTS IN PHP**

In PHP, an array is a data structure that allows you to store multiple elements in a single variable. These elements are stored as key-value pairs. In fact, you can use an array whenever you need to store a list of elements.

Because there are a few ways to initialize it, most of the time it's the array() construct.

An example would be:

```php
<?php
$array = array();
?>
```

In this example the variable $array is initialized as an empty array

Since PHP 5.4 you can initialize it like this:

```php
<?php
$array = [];
?>
```

Now to add elements it could be as follows

```php
<?php
$array = [];
$array[] = 'One';
$array[] = 'Two';
$array[] = 'Three';
echo '<pre>';
print_r($array);
?>
```

To access the elements and be able to see them in the console, it could be as follows

```php
<?php
$array = ['One', 'Two', 'Three'];

// get the first element of the $array array
echo $array[0];
echo "<br>";

// get the second element of the $array array
echo $array[1];
echo "<br>";
```

Which would show us the following:

```
One
Two
Three
```

To go through the elements of an array, the cleanest way to do it in code would be through a loop called foreach:

```php
<?php
$array = ['One', 'Two', 'Three'];

foreach ($array as $element) {
  echo $element;
  echo '<br>';
}
?>
```

**RELATIONSHIP BETWEEN CLASSES IN PHP**

**AGGREGATION**

You can specialize the relationship "association"

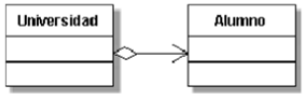
in two more types (note that the arrows are still continuous, but add a rhombus at their start

## COMPOSITION

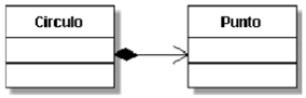
It is similar to aggregation, it only brings the semantics to the relation by saying that in addition to aggregation, it exists without the relation with A.

From the point of view of many languages (such as Java or PHP) this does not necessarily result in any changes to the code, but can be taken as part of the conceptual documentation of what the system design should look like.

Ejemplos de Agregación y de Composición




*"La Universidad agrupa muchos Alumnos"*




## Inheritance

Inheritance is a mechanism for creating classes by extending other base classes. The new class inherits the methods and properties of the base class it extends, so it already has out-of-the-box functionality..

The new class can have new methods and properties, or you can overwrite existing ones, as needed. As a result we have a new class, which in addition to its own type, can be identified by the same type as its "mother class" even if it behaves differently.

The Media class could provide some common methods and properties, and each descendant class does some things a little differently.

To apply it, you must add an extends and the name of the class to which you want to inherit after the class name.

```
class Estudiante extends Persona
{
```

**Polimorfism**

Polymorphism suggests multiple forms. It is the ability to access multiple functions through the same interface. Making the same identifier, or function can have different behaviors depending on the context in which it is executed.

To establish a polyformic relationship in PHP it is necessary to establish a hierarchy of classes so that each of the objects involved share the same root.

To do this we are going to create a simple class hierarchy where we will have a base class called "Polygon" and its respective extended classes: "Triangle", "Square", "Rectangle". Each of these classes will have a common method called "calculo" and whose function will be to show the mathematical formula for calculating the area of the geometric figure in question.

```php
<?php
class Poligono
{
  function calculo()
  {
  echo 'El area depende del tipo de poligono';
  }
}

class Cuadrado extends Poligono
{
  function calculo()
  {
  echo 'area de un cuadrado : a=l*l<br>';
  }
}

class Rectangulo extends Poligono
{
  function calculo()
  {
  echo 'area de un rectangulo : a=b*h<br>';
  }
}

class Triangulo extends Poligono
{
  function calculo()
  {
  echo 'area de un triangulo : a=(b*h)/2<br>';
  }
}
```

Once we have defined our classes, we will create the function that will be in charge of making the polymorphic call to the "calculo" method whose execution will vary depending on the object that implements it.

```php
function area(Poligono $obj)
{
  $obj->calculo();
}
/*
We create the necessary objects
*/
$cuadrado = new Cuadrado;
$rectangulo = new Rectangulo;
$triangulo = new Triangulo;
/*
Polymorphic call
*/
area($cuadrado);
area($rectangulo);
area($triangulo);
?>
```

The "area" function shows us the formula in each of its executions for each type of geometric figure, despite the fact that in its initial definition we have specified that the object is of the "Polygon" type, being the base class of each object.

**Abstraction**

An abstract class has the same structure as a normal class, you just need to add the keyword “abstract” at the beginning of its declaration.

```php
<?php

abstract class Poligono
{
  abstract function calculo();
}
```

An abstract class serves as a starting point for an object definition by creating a class hierarchy as a first step.

The abstract classes are similar to the normal ones in their construction and concept, with differences that they cannot create objects from them and they can incorporate abstract methods in which only their declaration exists, their implementation will go in future extended classes.

All methods declared as abstract must necessarily belong to an abstract class.

Encapsulation is possible in abstract classes (public, private, protected, final, etc).

```php
class Cuadrado extends Poligono
{
  function calculo()
  {
  echo 'area de un cuadrado : a = l*l<br><br>';
  }
}

class Rectangulo extends Poligono
{
  function calculo()
  {
  echo 'area de un rectangulo : a = b*h<br><br>';
  }
}

class Triangulo extends Poligono
{
  function calculo()
  {
  echo 'area de un triangulo : a = (b*h)/2<br>';
  }
}
```

**Interfaces**

An interface is a set of declarations to codify functions or methods in the different classes that implement them, used as a point of union between objects of different nature.

It cannot contain implemented methods or attributes, only method and constant declarations.

The same class can implement several interfaces and given its nature, all the methods are declared as public.

A class hierarchy is not necessary since it implements a connection between the different objects with the base class with the "implements" operator, achieving the necessary polymorphism in a certain function.

```php
<?php

interface Poligono
{
  public function calculo();
}

class Cuadrado implements Poligono
{
  public function calculo()
  {
  echo 'area de un cuadrado : a = l*l<br><br>';
  }
}

class Rectangulo implements Poligono
{
  public function calculo()
  {
  echo 'area de un rectangulo : a = b*h<br><br>';
  }
}

class Triangulo implements Poligono
{
  public function calculo()
  {
  echo 'area de un triangulo : a = (b*h)/2<br>';
  }
}
```

```php
function area(Poligono $obj)
{
  $obj->calculo();
}
/*
We create the necessary objects
*/
$cuadrado = new Cuadrado;
$rectangulo = new Rectangulo;
$triangulo = new Triangulo;
/*
Polymorphic call
*/
area($cuadrado);
area($rectangulo);
area($triangulo);
?>
```