PHP 7 - PROGRAMACIÓN ORIENTADA A OBJETOS

--- CONCEPTOS BÁSICOS ---

---

1. ***DEFINICIÓN DE CLASES (OBJETOS):***

   Las clases (que vienen a ser **objetos**) siempre se definen en **singular** y con las primeras letras de las palabras en **mayúscula.** Como buena práctica se aconseja que **cada clase se defina en su propio fichero**. **La idea es tener al final un directorio exclusivo para todos los ficheros de clases.**

```
class Person {

  // code for this class goes here

}

class ProductImage {

}
```

   Funciones que nos ayudan trabajando con clases:

   a) **get_declared_classes()** que devolverá un array con todas las clases.

   b) **class_exists($string)** que se le pasará un string como argumento y nos devolverá true or false si php encuentra ese string como una clase.

2. ***INSTANCIAS (también OBJETOS):***

   Una instancia es básicamente una variable a la que se le **asigna** como valor un objeto de una clase definida. Por lo que la variable pasará a ser también un **objeto**.

```
class Person {
}

$person1 = new Person;

$person2 = new Person;
```

   Funciones que nos ayudan trabajando con Instancias:

   a) **get_class($object)** a la que le pasamos una instancia como argumento y nos devuelve el nombre de la clase.

   b) **is_a($object, $string)** le pasamos una instancia y un string como argumentos y devolverá true or false si la instancia tiene el mismo nombre que el string.

3. ***PROPIEDADES ó ATRIBUTOS DE LA CLASE:***

   Vienen a ser las **variables** de una clase.

```
class Person {

  var $first_name;
  var $last_name;
  var $employed = false;
  var $country = 'None';

}
```

Si creamos una instancia $Customer = new Person. Y a esa instancia queremos asignarle un nombre, nos referiríamos de esta manera a dicho atributo:
$Customer->firstname = 'Amparo';

Funciones que nos ayudan trabajando con propiedades:
a) **get_class_vars($string)** Nos devuelve una lista con las propiedades de una **clase**.
b) **get_object_vars($object)** Nos devuelve una lista con las propiedades de la **instancia**
c) **property_exists($mixed, $string)** Devuelve true or false si una propiedad existe en una clase o en un objeto (instancia).

4. *MÉTODOS DE LA CLASE:*

Los métodos son **funciones** dentro de una clase que se declaran para que el objeto realice ciertas **acciones**.

```
class Person {

  var $first_name;
  var $last_name;

  function say_hello() {
    return "Hello world!";
  }

}
```

Si tenemos una instancia $Customer = new Person y queremos llamar a la función desde dicha instancia lo realizamos: **$customer->say_hello();**

Funciones que nos ayudan trabajando con métodos:
a) **get_class_methods($mixed)** Devuelve los métodos de una clase u objeto (instancia)
b) **method_exists($mixed, $string)** Devuelve true or false si un método pasado por argumento existe dentro de una clase u objeto (instancia)

5. *REFERIR A LOS ATRIBUTOS DE CLASE DESDE UN MÉTODO:* THIS-> .

Las variables definidas dentro de una clase **no llegan a ser definidas dentro de las funciónes de la clase** por lo que esta imagen abajo nos da "*undefined variable*".

```php
class Person {
  var $first_name;
  var $last_name;

  function full_name() {
    return $first_name . " " . $last_name;
  }
}

$p = new Person;
$p->first_name = 'Jack';
$p->last_name = 'Jackson';
echo $p->full_name();
// Notice: undefined variable: first_name, last_name
```

Para poder referirnos a los atributos de la clase **que están fuera del método** utilizamos la seudo variable **this->.** *"return $this->first_name . " " . $this->last_name;*

--- HERENCIA DE CLASES ---

---

6. *¿QUÉ ES HERENCIA?*

Herencia se da cuando una **nueva clase toma las propiedades y los métodos de una clase ya existente.** Esto lo hacemos para: Organizar el código, Prevenir repeticiones, simplificar el mantenimiento, Prevenir inconsistencias y bugs.
Las subclases se utilizan para anular o extender atributos y métodos de la clase padre.

7. *DEFINIR SUBLCASES:*

Se definen igual que las clases pero añadiéndole la palabra ***extends*** seguido de la clase de la que hereda. Tener en cuenta que puede haber una subclase que se extienda de otra subclase.

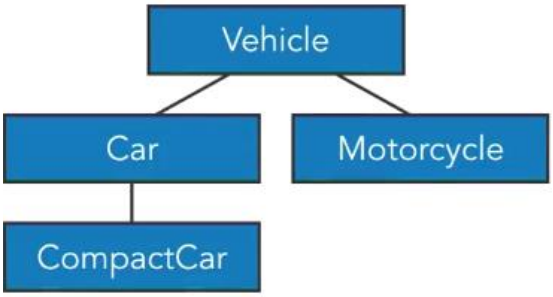
```php
class Vehicle {
  var $wheels;
  var $doors;

  function gasoline_type() {
    return 'Unleaded';
  }
}

class Car extends Vehicle {
}

class Motorcycle extends Vehicle {
}

class CompactCar extends Car {
}
```



Funciones que nos ayudan trabajando con herencia:

a) **get_parent_class($mixed)** Devuelve el padre de una clase o instancia.

b) **is_subclass_of($mixed, $string)** Devuelve true/ false si un string es subclase de una clase o instancia.

c) **class_parents($mixed)** Devuelve los padres de una clase o instancia.

8. *EXTENDER Y ANULAR ATRIBUTOS Y MÉTODOS EN SUBCLASES:*

El verdadero **poder de las subclases yace en poder extender o anular las propiedades de la clase padre**.

```php
class Vehicle {

  var $color;
  var $wheels;
  var $doors;

  function upload_image($file) {
    // ...
  }

  function is_good_for_rain() {
    return true;
  }

}
```

```php
class Car extends Vehicle {
  var $wheels = 4;
  var $doors = 4;
  var $convertible = false;

  function is_good_for_rain() {
    return !$this->convertible;
  }
}

class Motorcycle extends Vehicle {
  var $wheels = 2;
  var $doors = 0;

  function is_good_for_rain() {
    return false;
  }
}
```

--- CONTROL DE ACCESO A OBJETOS ---

---

9. *MODIFICADORES DE VISIBILIDAD:* PUBLIC, PRIVATE and PROTECTED.

Los modificadores de visibilidad se utilizan para **controlar el acceso a propiedades y métodos** de los objetos.

a. ***PUBLIC:*** Cualquier propiedad o método será **accesible desde cualquier lugar**, sin importar si estamos dentro de la clase, o dentro de una subclase o si creamos una instancia de un objeto y trabajamos con ella.
b. ***PRIVATE:*** Es el caso opuesto a Public. Si tenemos una propiedad o método privado, **solo será accesible desde dentro de la clase**. No es accesible por subclases ni a través de instancias haciendo referencia a la propiedad o método.
c. ***PROTECTED:*** Es **como Private, pero dando acceso también a subclases.**

Al definirse una variable con *"var"* la estamos haciendo automáticamente **pública. Pero es una mejor práctica usar public en vez de var.**

```
class Example {

  public $a;                          // default if undeclared (var)
  protected $b;
  private $c;

  public function hello_world() {     // default if undeclared
    return "Hello everyone!";
  }

  protected function hello_family() {
    return "Hello family!";
  }

  private function hello_me() {
    return "Hello me!";
  }
```

Practicas Recomendadas:

a) Usar **siempre** los modificadores de visibilidad, no los de por defecto.
b) **Considerar y recapacitar** sobre la visibilidad de nuestro código.
c) Declarar por defecto las propiedades y los métodos públicos **solo cuando sea necesario.**
d) Agrupar las propiedades y métodos con visibilidad similar.

10. *CUIDADO CON LA SOBRECARGA:*

La sobrecarga en PHP (o mejor llamarlo propiedad Dinámica), se da cuando PHP maneja las llamadas a propiedades y métodos que no existen o **no son visibles**.

La sobrecarga tiene algunos comportamientos por defecto:
a) Por defecto, obtendremos un *"**Notice: Undefined Property**"* cuando intentamos acceder a una **propiedad no definida**.
b) Por defecto, **se definirá una propiedad y su valor al asignar un valor a una propiedad que no está definida.**

```
class Product {

}

$p = new Product;
echo $p->name;
// Notice: Undefined property: Product::$name

$p->name = 'Guitar';
echo $p->name;
// Guitar
```

La sobrecarga se puede dar cuando trabajamos **con subclases que heredan atributos privados**, porque estas propiedades deberían de ser visibles en las subclases, pero en realidad **no son visibles.** Por lo que define una nueva propiedad en la subclase y por ese mismo hecho no se comporta como nosotros esperamos. **Ejemplo:**

```php
<?php

class Person{

    private $nombre = 'Pablo';

    public function hola_nombre(){

        echo "<br>$ " . $this->nombre;
    }

}

class Student extends Person
{

}

$student = new Student;

$student->nombre = 'Pedro';
// Define la variable y valor en la subclase
echo $student->hola_nombre();
// Devuelve 'Pablo' (Método en la clase)

?>
```

```php
<?php

class Person{

    private $nombre = 'Pablo';

}

class Student extends Person
{
    public function hola_nombre(){

        echo "<br>$ " . $this->nombre;
    }
}

$student = new Student;

$student->nombre = 'Pedro';
// Define la variable y valor en la subclase
echo $student->hola_nombre();
// Devuelve 'Pedro'

?>
```

11. *MÉTODOS SETTER and GETTER:*

Los métodos *setter* y *getter* se utilizan como métodos de clase intermediarios para poder establecer y cambiar valores de propiedades privadas de dicha clase.

```php
class Product {
  private $name;

  public function set_name($value) {
    $this->name = $value;
  }

  public function get_name() {
    return $this->name;
  }
}

$p = new Product;
$p->set_name('Birdhouse');
$p->get_name();
// Birdhouse
```

Los métodos setter y getter:

a) Nos permiten acceder a propiedades privadas que de otra forma serían inaccesibles.
b) Útil para regular el acceso.
c) Útil para propiedades de solo lectura o solo escritura.
d) Útil para pre-procesar valores. (Como tomar alguna información y procesarla antes de asignarla a la propiedad. O queremos sacar el valor de una propiedad y realizar algún proceso con ella antes de devolverla al usuario).
c) Evitar *setter* y *getter* que no sirvan de nada. Como en la imagen de arriba que podría hacer público el atributo y te ahorras los métodos.

*--- PROPIEDADES Y MÉTODOS **CONSTANTES** Y **ESTÁTICOS** ---*

12. *EL MODIFICADOR o PALABRA RESERVADA "CONST":*

No solo existen atributos y métodos dentro de las clases, también podemos declarar **constantes**, que almacenarán **valores fijos en una clase que de ninguna manera podemos alterar**. Los elementos constantes no dependen de un objeto instanciado.

Para declarar una constante dentro de una clase usamos *const* antes del nombre **sin el símbolo del $**

```
class MiClase { const MI_CONSTANTE = 'Una Constante'; }
```

Para acceder al valor de una constante debemos usar el nombre de la instancia o el nombre de la clase seguido de ***"dos puntos ::"*** **MiClase::MI_CONSTANTE;**
Utilizamos los :: en vez de $this debido a que los métodos constantes y estáticos se pueden invocar sin tener creada una instancia del objeto.

Buenas prácticas con la palabra reservada const:
a) Es recomendable poner el nombre de la constante en **Mayúsculas** para diferenciarlas rápidamente.
b) Las constantes se suelen utilizar para almacenar datos que utilizaremos para realizar una **conexión a una base de datos**.

13. *EL MODIFICADOR o PALABRA RESERVADA "STATIC":*

Al igual que las constantes, los elementos estáticos no dependen del objeto instanciado, **sino que son únicos para <u>todos los objetos de una misma clase</u>**.

```php
<?php
function sin_static() {
    $var = 1;
    echo $var . "<br />";
    $var++;
}
sin_static(); //Devuelve 1
sin_static(); //Devuelve 1
sin_static(); //Devuelve 1
?>
```

```php
<?php
function con_static() {
    static $var = 1;
    echo $var . "<br />";
    $var++;
}
con_static(); //Devuelve 1
con_static(); //Devuelve 2
con_static(); //Devuelve 3
?>
```

Para referirnos desde fuera de una clase a valores estáticos contenidos dentro de una clase **<u>sin instanciar un objeto y siempre que la variable estática sea pública</u>,** lo podemos realizar llamando a la clase seguido de :: y el nombre de la variable estática. **MiClase::Var_estatica;**
Sin embargo, para referirnos desde dentro de una clase a valores estáticos contenidos dentro de una clase lo podemos realizar mediante **self**.
self::Var_estática;.

Importante!:
a) **Las propiedades estáticas no son accesibles a través de una instancia.**
b) **Los métodos estáticos si lo son, pero NO debes hacerlo.**

14. *COMPORTAMIENTO ESTÁTICO HEREDADO:*

Las variables estáticas son variables **compartidas**. Por lo que su valor fluctuará por la clase, subclases e instancias comportándose digamos como una variable global.

```
class Student {
  public static $total_students = 0;
}

class PartTimeStudent extends Student {
}

Student::$total_students++;
Student::$total_students++;
Student::$total_students++;
PartTimeStudent::$total_students++;

echo Student::$total_students;
// 4
echo PartTimeStudent::$total_students;
// 4
```

15. *REFERENCIA A LAS CLASES PATERNAS:*

Hemos visto que podemos hacer referencia a propiedades estáticas **dentro de la clase** mediante ***self::*** y **fuera de la clase** mediante ***nombre_clase::***
Pero también podemos **referenciar a las propiedades estáticas de la clase paterna de dos formas**:

a. Mediante el nombre de la clase paterna. ***Nombre_clase_padre::***
b. Mediante la palabra reservada *"parent"*. ***parent::var_estatica.***

Es mejor práctica que nos refiramos con ***"parent"*** para hacer referencia a una propiedad o método estático que esté en la clase padre. **Esto solo sirve para propiedades y métodos estáticos, no para instancias.** (Técnicamente puedes acceder a métodos pero se recomienda NO hacerlo)

Otro dato importante es que usaremos *parent* seguido del nombre de la propiedad o método. Pero **no es necesario referenciar con *parent* cuando trabajamos con propiedades estáticas**. Ya que las propiedades estáticas **ya están compartidas con todas las subclases.** Es una variable común que ambos utilizan. Por lo que no utilizaremos *parent::var_estática* si no ***self::var_estática.***

Lo que **sí es práctico y recomendable** es llamar al método estático padre, **especialmente a la hora de sobrescribir/extender o recurrir a los métodos de la clase padre desde la subclase:**

```
class Chef {

  public static function make_dinner() {
    echo "Cook food.";
  }

}

class AmateurChef extends Chef {

  public static function make_dinner() {
    echo "Read recipe.";
    parent::make_dinner();
    echo "Clean up mess.";
  }
```

```
class Image {

  public static $resizing_enabled = true;

  public static function geometry() {
    echo "800x600";
  }
}

class ProfileImage extends Image {

  public static function geometry() {
    if(self::$resizing_enabled) {
      echo "100x100";
    } else {
      parent::geometry();
```

16. *ENLACE ESTÁTICO EN TIEMPO DE EJECUCIÓN ("LATE STATIC BINDINGS"):*

1º Entendamos que la palabra reservada ***"self"* siempre hace referencia a la clase donde se escribe:**

```
class Bicycle {
  protected static $wheels = 2;

  public static function wheel_details() {
    $wheel_string = self::$wheels == 1 ? "1 wheel" : self::$wheels ." wheels";
    return "It has " . $wheel_string . ".";
  }
}

class Unicycle extends Bicycle {
  protected static $wheels = 1;
}

echo Bicycle::wheel_details();
// It has 2 wheels.
echo Unicycle::wheel_details();
// It has 2 wheels.
```

En este ejemplo vemos que el resultado será el mismo sin importar si invocamos al método desde la clase o la subclase. Porque ***self* siempre va a hacer referencia a Bicycle. Y es lo mismo que poner Bicycle::$wheels.**

Para resolver este problema, la versión de PHP 5.3 incorporó una nueva funcionalidad *("late static bindings")* **que permite hacer referencia a la clase en uso** dentro de un contexto de herencia con la palabra reservada ***static*** (ya que no querían crear una nueva palabra reservada).

```
class Bicycle {
  protected static $wheels = 2;

  public static function wheel_details() {
    $wheel_string = static::$wheels == 1 ? "1 wheel" : static::$wheels ." wheels";
    return "It has " . $wheel_string . ".";
  }
}

class Unicycle extends Bicycle {
  protected static $wheels = 1;
}

echo Bicycle::wheel_details();
// It has 2 wheels.
echo Unicycle::wheel_details();
// It has 1 wheel.
```

Resumen!:

a) Las referencias estáticas con ***STATIC* se resuelven usando la última clase desde la que se llama.**
b) Las referencias estáticas con ***SELF* se resuelven usando la clase donde se escribió el código self.**
c) ***Self y Static* solo pueden hacer referencia a propiedades estáticas.**
d) ***get_called_class()*** Nos devuelve la última clase a la que se ha llamado.

--- MÉTODOS MÁGICOS ---

17. *MÉTODO CONSTRUCTOR*: __construct()

El constructor es un **método especial dentro de una clase**, se suele utilizar para **darle valor a los atributos del objeto al instanciarlo**.

Es el primer método que se ejecuta al crear el objeto y se llama automáticamente al crearlo. Este método puede recibir parámetros como cualquier otro método y para pasárselos tenemos que pasarle los parámetros a la instancia. **El constructor NO DEVUELVE ningún dato y debe ser un método público.** Se define: *public function __construct([parámetros]){"algoritmo"}*

```php
class Coche{

    // Atributos
    public $modelo;
    public $color;
    public $velocidad;

    // Constructor
    public function __construct($modelo, $color, $velocidad
        $this->modelo = $modelo;
        $this->color = $color;
        $this->velocidad = $velocidad;
    }
```

Para hacer uso del constructor instanciamos un objeto ***$coche = new Coche('BMV','Verde','100')***

Resumen Importante!:

a) Los constructores se utilizan para darle valores a los atributos del objeto al instanciarlo.

b) Es el primer método que se ejecuta al crear el objeto y se llama automáticamente al crearlo.

c) El constructor debe ser un método **público y NO devuelve** ningún dato. **No se puede llamar.**

18. *PARÁMETROS DEL CONSTRUCTOR CON UN ARRAY DE ARGUMENTOS*

Hemos visto como un constructor puede recibir parámetros para dar valores a los atributos de clase. Pero cuando un constructor recibe muchos parámetros, al instanciar el objeto nos puede resultar confuso introducir dichos valores y tendremos que ver el orden de los parámetros en el constructor. Incluso es posible que queramos que dicho objeto contenga parámetros con valores por defecto que hayamos definido en la clase. Para ello lo que podemos realizar es pasarle un array asociativo de argumentos ***$args['name_arg1' = 'x' , 'name _arg2' = 'y' , 'name_arg3 = 'z' …]***

```php
class Product {
  public $name;
  public $color;
  public $price;

  public function __construct($args=[]) {
    $this->name = $args['name'] ?? NULL;
    $this->color = $args['color'] ?? NULL;
    $this->price = $args['price'] ?? NULL;
  }

}

$shirt = new Product(['name'=>'T-shirt', 'color'=>'blue', 'price'=>9.99]);
echo $shirt->color;
// blue
```

*Nota: **?? NULL** hará que por defecto ponga el valor a NULO, si no se introduce ningún valor.*

19. *MÉTODO DESTRUCTOR:* __destruct()

Los destructores son **métodos públicos de clase** que se suelen utilizar cuando se trabaja con loggings o para realizar tareas de limpieza, o junto con un constructor como un conjunto de métodos de "preparación" y "desmontaje".

Al contrario que los constructores, los destructores **no toman ningún tipo de argumento** y se ejecutarán automáticamente cuando PHP **haya ejecutado el script y/o cuando observe que una variable instanciada no contenga ningún tipo de valor y no sirva para nada**. Para esto utilizaremos ***"unset"***. Con *"unset"* dejaremos a la variable instanciada sin ningún valor, PHP verá que es innecesaria y automáticamente ejecutará el método destructor.

```php
class Product {
  public static $instance_count = 0;

  public function __construct() {
    Logger::log('Creating a product');
    self::$instance_count++;
  }

  public function __destruct() {
    Logger::log('Deleting a product');
    self::$instance_count--;
 }

}

$shirt = new Product;
echo Product::$instance_count;
// 1

unset($shirt);
echo Product::$instance_count;
// 0
```

```php
<?php
class Session
{
    protected $data = array();

    public function __construct()
    {
        // load session data from database or file
    }

    // get and set functions

    public function __destruct()
    {
        // store session data in database or file
    }
};
```

Vamos a trabajar más con constructores que con destructores, debido a que muchas veces instanciamos un objeto y trabajamos con él para finalmente mostrarle los resultados al usuario.

Resumen Importante!:

a) Son métodos **públicos** de clase que no toman **ningún** tipo de **argumento**.
b) Se ejecutan **automáticamente** una vez ejecutado el script y/o cuando PHP observe una variable instanciada sin valor (inservible).
c) Se utiliza ***"unset"*** para dejar a una variable instanciada sin valor.

20. *MÉTODO DE CLONACIÓN:* __clone()

Cuando trabajamos con variables y objetos podemos llegar a pensar que para copiar un objeto basta con asignarlo a otra variable: $persona2 = $persona1. Pero **esto no es correcto**, debido a que los objetos trabajan por **referencias**, por lo que, aunque tengamos ahora dos variables diferentes como son *$persona1* y *$persona2*, **si cambiamos un valor en una de ellas se reflejara en ambas porque estamos trabajando sobre el mismo objeto.**

Para poder clonar un objeto correctamente debemos de usar la instrucción ***clone*** en la asignación de la variable, tal como si fuésemos a usar *new*: **$persona3 = *clone* $persona1.**

Pero puede darse el caso de que **necesitemos provocar algún cambio en el objeto al clonarlo**, por ejemplo asignarle un nombre especial, limpiar algunas variables que contentan parámetros que ya no necesitemos, etcétera. **Para este caso existe el método __clone() que se dispara en el momento**

de clonar el objeto mediante la instrucción *clone*, y al igual que muchos de los métodos mágicos ya vistos, nos da la posibilidad de ejecutar sentencias o generar cambios al dispararse.

```php
class Persona{
    private $nombre;

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

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

    public function __clone(){
        $this->nombre='Clon de: '.$this->nombre;
    }

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

```php
$persona = new Persona('Juan');
$persona2 = clone $persona;
$persona3 = new Persona('');

$persona3->setNombre('Pedro');

echo $persona->show_name();
// Juan
echo '<br>';
echo $persona2->show_name();
//Clon de Juan
echo '<br>';
echo $persona3->show_name();
// Pedro
echo '<br>';
```

Importante!:
a) Siempre método Público.
b) No se le pasan parámetros.

21. *COMPARANDO OBJETOS:*

Podemos comparar dos objetos al igual que comparamos dos variables, con los operadores **==** o **===**

Al utilizar el operador de comparación **(==)**, se comparan de una forma sencilla **las variables de cada objeto**, es decir: **dos instancias de un objeto son iguales si tienen los mismos atributos y valores** (los valores se comparan con ==), **y son instancias de la misma clase**.

Cuando se utiliza el operador identidad **(===)**, **las variables de un objeto son idénticas sí y sólo sí hacen referencia a la misma instancia de la misma clase**.

22. *AUTOLOAD, CARGAR AUTOMÁTICAMENTE LAS CLASES SIN REQUIRE O INCLUDE:* __autoload()

Autoload **se define fuera de la clase** y se va **a llamar automáticamente cuando PHP encuentre el nombre de una clase que no reconoce**, justo antes de que PHP nos dé un error. Esto nos proporcionará una oportunidad de localizar la clase.

Autoload **recibe como parámetro una clase** con la que va a realizar la búsqueda:

```php
function __autoload($class){echo "Definition for {$class} is missing!";}
```

Pero qué ocurre cuando tenemos 50 clases de las cuales solo hemos programado 24 y no conocemos sus dependencia? Es más, normalmente cuando las clases pasan de la decena se nos hace algo confuso qué es lo que hacen y/o como éstas se relacionan. Podríamos directamente cargar todas las clases pero es mala praxis ya que se consume mucha memoria solo para, en algunos casos, utilizar solamente algunas de ellas en el script que esté ejecutando PHP en ese momento.

En esos casos utilizaremos **spl_autoload_register()** *"**S**tandard **P**ublic **L**ibrary"*. En esencia, lo que hace es, cada vez que se necesita —por ejemplo, al sacar la instancia de una clase (new tal) o al extender una clase—, detectar si la clase ha sido cargada y, en caso contrario, cargarla. Así, por ejemplo, si hemos guardado las clases en una carpeta llamada class, para despreocuparnos de este jaleo basta con incluir al principio de cada archivo nuestro código:

```php
<?php
spl_autoload_register(function ($clase) {
    require_once('../class/'.$clase.'.php');
});
?>
```

Para ser más cuidadosos (y por ejemplo no cargar un archivo distinto al que no queremos) podemos realizar una función que nos realice la búsqueda y después **pasársela como parámetro <u>String</u>** a la función **spl_autoload_register():**

```php
function my_autoload($class) {
  if(preg_match('/\A\w+\Z/', $class)) {
    include 'classes/' . $class . '.class.php';
  }
}

spl_autoload_register('my_autoload');

$dog = new Pet;
```

Nota: preg_match chequeará que el nombre de la clase contiene caracteres legales (mayus, minus, numéricos o barrabaja). Es un chequeo de seguridad. Si es así, entonces incluirá el archivo "$class" en la carpeta "clases" *más* ".class.php" *(que podría ser solo ".php")*

Importante!:

a) Para realizar mejor uso de autoload debemos de ser organizados con nuestros ficheros, nombres de estos y de sus clases y almacenarlos en una sola carpeta. Cada clase se guarda en un fichero con el mismo nombre: clases/My_Clase_Uno.php -> function My_Clase_Uno().

b) Autoload se utiliza para cargar tus ficheros y clases de forma segura, como una red.

c) Se conserva memoria.

23. *CONSTANTES MÁGICAS:*

Al igual que PHP provee de unos métodos mágicos predefinidos para realizar diferentes acciones, también provee de unas **constantes predefinidas para obtener información** del código contenido en el script que estamos ejecutando.

Las constantes mágicas al igual que los métodos mágicos comienzan con dos barras inferiores (__), pero a diferencia de estos últimos las constantes **también acaban con las dos barras inferiores**, además son sensibles a si se escriben con mayúsculas o minúsculas, **siendo la única forma correcta las mayúsculas**. Con estas características lo tenemos muy fácil para detectarlas al leer un código, incluso si no las conocemos sabremos por su forma de escribirlas de que se trata.

- **__LINE__**: Esta constante nos devuelve la línea exacta donde está situada en el fichero, puede ser muy útil para trazar errores o crear logs en nuestras aplicaciones.
- **__DIR__:** Esta constante nos devuelve el nombre del directorio donde está incluido el script, y al igual que LINE puede ser muy útil para trazar bugs.
- **__FILE__:** Nos devuelve la Ruta completa donde se encuentra el script, además del nombre de este con su extensión, al igual que en las constantes anteriores puede ser muy útil para localizar bugs.
- **__FUNCTION__:** Si la usamos dentro de una función nos devolverá el nombre de la función que la contiene.
- **__CLASS__:** Al igual que FUNCTION nos devuelve el nombre de la clase donde esta declara.
- **__METHOD__:** Es la hermana gemela de FUNCTION solo que enfocada a los métodos, su función es exactamente la misma, devolver el nombre del método que la contiene.
- **__NAMESPACE__:** Aunque aún no hemos llegado a explicar los namespaces de PHP, no está de más conocer esta constante, la cual nos devuelve el namespace donde está declarada.
- **__TRAIT__:** Tampoco hemos llegado a hablar sobre los Traits, pero al igual que con los namespaces no está de más conocerla. Esta constante devuelve el nombre del Trait donde fue declarada, y en caso de haber sido declarado un namespace también lo devuelve.

Info sacada de:
PHP – Object Oriented Programing