Guion para la Unidad 5: POO en PHP

Sitio: Centros - Granada
Curso: Desarrollo web en entorno servidor
Libro: Guion para la Unidad 5: POO en PHP

Imprimido por: Aguilera Aguilera, Javier
Día: miércoles, 22 de diciembre de 2021, 12:32

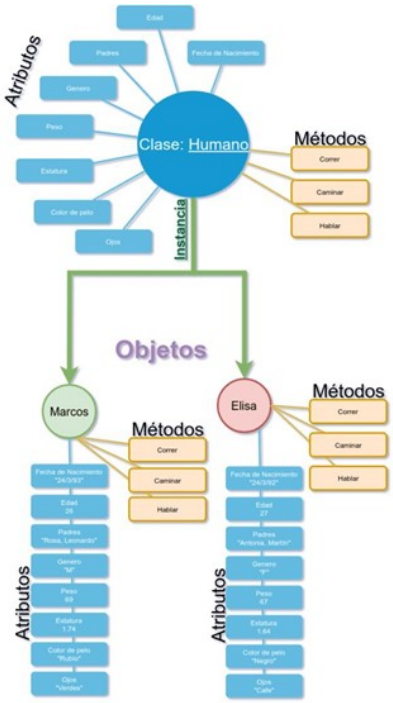
Tabla de contenidos

1. Introducción

PHP en su origen no es un lenguaje orientado a objetos puro.

Las primeras versiones PHP que incorporaban la POO presentaban ciertas carencias (como la ocultación, la herencia múltiple, el polimorfismo, …), pero a partir de la versión 7 fue completamente remodelado y actualmente tiene soporte completo para la POO

Ejemplo que ilustra los elementos de la POO:



Fuente:es.quora.com

2. Clases y objetos

Enel manual de referencia de PHP podemos encontrar toda la información que vamos a necesitar para trabajar con **clases y objetos**.

Recordemos que:

- Un objeto en POO representa alguna entidad de la vida real, es decir, alguno de los objetos únicos que pertenecen al problema con el que nos estamos enfrentando, y con el que podemos interactuar.
- Cada objeto, de igual modo que la entidad de la vida real a la que representa, tiene un estado (es decir, unos atributos con unos valores concretos) y un comportamiento (es decir, tiene funcionalidades o sabe hacer unas acciones concretas).
- Podemos definir informalmente una clase como una plantilla (o esqueleto o plano) a partir de la cual se crean los objetos.
- Cuando hablamos de objeto, hacemos referencia a una estructura que hay en un momento determinado de la ejecución del programa, mientras que cuando hablamos de clase hacemos referencia a una estructura que representa a un conjunto de objetos.

Definición de una clase:

Para declarar una clase se utiliza la palabra reservada ***class*.**

Los nombres de las clases suelen ir en mayúscula.

Ejemplo:

```
class Empleado
{
    private $nombre;
    private $apellido;
    private $edad;

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

Dentro de un método, la variable $this hace referencia al objeto sobre el que invocamos el método

Los **atributos** se declaran utilizando su nombre, los modificadores que pueda tener y opcionalmente un valor por defecto.

Ejemplo:

```
class Clase{
    private $att1 = 10; // con valor por defecto
    private $atr2; // sin valor por defecto
    private static $atr3 = 0; // estático
  ...
}
```

Las clases solamente son definiciones.

Crear un ejemplar de una clase, es decir instanciar un objeto de una clase se hace mediante el operador new seguido del nombre de la clase :*$obj = new Clase();*

Por ejemplo,
$emp = new Empleado;
Con esto hemos creado, o mejor dicho, instanciado, un objeto de la clase Empleado llamado $emp.

3. Encapsulación

Para los atributos y métodos se pueden utilizar los siguientes modificadores de visibilidad:

- public. Se pueden utilizar desde dentro y fuera de la clase.
- private. Pueden emplearse desde la propia clase.
- protected. Se pueden utilizar dentro de la propia clase, las derivadas y las antecesoras

Normalmente los atributos se declaran como privados y se crean métodos públicos para acceder a ellos.´

Para acceder a los métodos y atributos se utiliza:

- $objeto->propiedad;
- $objeto->método(argumentos);

Para prevenir que se produzcan cambios inesperados en el sistema y para ocultar la información para que no pueda ser modificada o vista por otras clases se debe de realizar un encapsulamiento de los datos. Es decir, sólo los métodos internos del objeto deberían acceder a su estado. Para modificar y acceder de forma segura a las propiedades o atributos se deben de usar los métodos conocidos como Getters y Setters.

Los métodos getters son aquellos que nos devuelven los valores de las propiedades.

Los métodos setters son aquellos que permiten modificar el valor de una propiedad.

El nombre de los métodos siguen la convención adoptada por PHP CamelCase, es decir que los métodos que tengan palabras compuestas hay que escribirlos con todas las palabras juntas y la primera letra de cada palabra en mayúscula, exceptuando la primera. Ejemplo: *copiarDatosPersonales*.

Recuerda que los métodos protegidos son menos restrictivos que los privados, Cuando declaramos un método como protegido, su visibilidad se extiende a la clase que lo declara y a todas las clases que lo heredan. Todos los métodos que no requieran acceso desde el exterior de la clase deberían ser definidos como protegidos.

Los métodos públicos serán considerados la interfaz del objeto, es decir, es la parte conocida de ese objeto para el resto de la aplicación.

4. Constructor y destructor

Un constructor es un método especial (mágico) que se suele usar para darle valor a los atributos de un objeto. Nunca devuelve un dato. Este método se llama automáticamente al crear el objeto. Puede recibir parámetros. Suele ser el primer método que definimos. Siempre debe ser público.

Se puede crear un constructor para la clase con el método __construct(). Este método forma parte de los métodos mágicos, nombres reservados a tareas concretas y que comienzan por dos guiones bajos, "__".

Ejemplo de cómo se hace en PHP 8:

PHP 7

```
class Point {
  public float $x;
  public float $y;
  public 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;
  }
}
```

PHP 8

```
class Point {
  public function __construct(
    public float $x = 0.0,
    public float $y = 0.0,
    public float $z = 0.0,
  ) {}
}
```

Destructor

Los destructores **__destruct**()son métodos que se encargan de realizar las tareas que se necesita ejecutar cuando un objeto deja de existir.

Cuando un objeto ya no está referenciado por ninguna variable, deja de tener sentido que esté almacenado en la memoria, por tanto, el objeto se debe destruir para liberar su espacio.

El destructor puede hacer labores de limpieza, como liberar una conexión a una base de datos o algún otro recurso que has reservado dentro de la clase. Puesto que has reservado el recurso dentro de la clase, tienes que liberarlo aquí o se quedará en ella indefinidamente. Muchos problemas en el conjunto del sistema los causan programas que ocupan recursos y se olvida liberarlos.

En el momento de su destrucción se llama a la función destructor, que puede realizar las tareas que el programador estime oportuno. La creación del destructor es opcional.

Ejemplo:

Clase típica con un constructor que se encarga de inicializar los datos a través de los parámetros y después inicializar los atributos:

```
<?php
class Usuario {
  private string $nombre;
  private string $apellido1;
  private string $apellido2;
  public function __construct(string $nombre, string $apellido1, string $apellido2 ) {
    $this->nombre = $nombre;
    $this->apellido1 = $apellido1;
    $this->apellido2 = $apellido2;

  }
}

$usuario = new Usuario("María", "González", "Pérez");
```

Como hacer más ligera la carga de estos datos en PHP 8:

```
<?php
class Usuario {

  public function __construct( private string $nombre,
                               private string $apellido1,
                               private string $apellido2) {
  }
}

$usuario = new Usuario("María", "González", "Pérez");

echo "<pre>";
var_dump($usuario);
echo "</pre>";
```

Otra novedad de PHP 8 son los argumentos con nombre. Antes las funciones y los métodos obligaban a añadir los parámetros en el orden en que están definidos. Gracias a los argumentos con nombre podemos añadir datos a los métodos sin tener en cuenta el orden, simplemente añadiendo el nombre y dos puntos antes del parámetro.

En el ejemplo anterior podemos poner:

```php
$usuario = new Usuario(apellido1: "González", apellido2: "Pérez", nombre:"María");
```

5. Herencia

La finalidad de la herencia es poder pasar todo el contenido de una clase, propiedades y métodos, a una subclase y modificar sólo las partes que sean diferentes.

Para crear una clase que herede de otra, se utiliza la palabra *clave_extends*. La clase derivada tendrá los mismos atributos y métodos que la clase base y podrá añadir nuevos o sobreescribirlos. Se llama redefinición de métodos a la creación de funciones en la clase hija, con el mismo nombre que en la clase padre.

Importante: el constructor de la clase madre NO se invoca automáticamente si existe constructor en la clase hija.

Como los constructores, los destructores padre no serán llamados implícitamente por el motor. Para ejecutar un destructor padre, se deberá llamar explícitamente a **parent::__destruct()** en el interior del destructor. También como los constructores, una clase child puede heredar el destructor de los padres si no implementa uno propio.

Ejemplo básico con uso de herencia:

```php
<?php
class Persona {
    private $DNI;
    private $nombre;
    private $apellido;
    function __construct($DNI, $nombre, $apellido) {
        $this->DNI = $DNI;
        $this->nombre = $nombre;
        $this->apellido = $apellido;
    }
    public function getNombre() {
        return $this->nombre;
    }
    public function getApellido() {
        return $this->apellido;
    }
    public function setNombre($nombre) {
        $this->nombre = $nombre;
    }

    public function setApellido($apellido) {
        $this->apellido = $apellido;
    }
    public function __toString() {
        return "Persona: " . $this->nombre . " ". $this->apellido;
    }
}
class Cliente extends Persona{
    private $saldo = 0;

    function __construct($DNI, $nombre, $apellido, $saldo){
        parent::__construct($DNI, $nombre, $apellido);
        $this->$saldo = $saldo;
    }
    public function getSaldo(){
        return $this->saldo;
    }
    public function setSaldo($saldo){
        $this->saldo = $saldo;
    }
    public function __toString(){
        return  "Cliente: ". $this->getNombre() ;
    }
}
// crear una persona
$per = new Persona("1111111A", "Ana", "Puertas");
// mostrarla, usa el método __toString()
echo $per . "<br>";
// cambiar el apellido
$per->setApellido("Montes");
// volver a mostrar
echo $per . "<br>";
// crea un cliente
$cli = new Cliente("22222245A", "Pedro", "Sales", 100);
// lo muestra
echo $cli . "<br>";
```

En PHP no es necesario definir el tipo de las variables que usamos. En POO es una buena práctica definir los tipos de los parámetros de entrada para evitar errores. De esta manera, si nos confundimos en algún dato, PHP nos alertará inmediatamente de que esperaba una variable con un tipo distinto.

El siguiente ejemplo muestra código con tipos:

```php
class Persona {
  protected string $nombre;
  protected string $apellido1;
  protected string $apellido2;

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

  function getNombre() {
    return $this->nombre;
  }

  function setApellidos(string $apellido1, string $apellido2
    $this->apellido1 = $apellido1;
    $this->apellido2 = $apellido2;
  }

  function getApellidos() {
    return $this->apellido1 . " " . $this->apellido2;
  }
}

class PersonaUSA extends Persona {
    protected $id;

   function setId(string $id) {
     $this->id = $id;
    }

    function getId() {
      return $this->id;
    }

    function getApellidos() {
      return $this->apellido2 . " " . $this->apellido1;
    }
}

function cambiaNombre(Persona $persona, string $nombre) {
  $persona->setNombre($nombre);
}

class PersonaEspaña extends Persona {

    private $dni;

    function setDni(string $dni) {
        $this->dni = $dni;
      }

      function getDni() {
        return $this->dni;
      }
```

Fuente: Curso de PHP 8 y MySQL 8 Editorial: Anaya

En otro fichero podemos hacer uso de la clase incluyendo, por ejemplo, el siguiente código:

```php
<?php
    require_once("clases2.php");
    $luis = new PersonaUSA();
    $luis->setNombre("Luis Miguel");
    $luis->setApellidos("Cabezas", "Granado");
    $luis->setId("66612345");
?>
<h1>
    Datos de <?= $luis->getNombre() . " " . $luis->getApellidos(); ?>
</h1>
<h2>
    ID <?= $luis->getId()  ?>
</h2>
<?php
  cambiaNombre($luis,"Pedro");
  echo $luis->getNombre();
?>
```

6. Propiedades y métodos estáticos

Los **atributos y métodos se pueden declarar como estáticos**, de manera que no habrá uno por objeto, sino uno por clase. Usamos la palabra reservada *static*.

Tanto los métodos como las propiedades estáticas son accesibles sin la necesidad de instanciar la clase en la que se encuentra.

Esto provoca que, si la clase se instancia, una propiedad estática no puede ser accedida desde ese objeto. No ocurre lo mismo con los métodos ya que, aunque sean estáticos si pueden ser accedidos desde objetos.

Los métodos estáticos nos permiten declarar funciones dentro de una clase que no utilicen propiedades o métodos de la misma. Estos métodos pueden calcular valores numéricos, hacer una conexión a una base de datos o comprobar que un correo electrónico esté bien definido.

Tanto las propiedades como los métodos pueden emplearse directamente, con el operador Paamayim Nekudotayim (los cuatro puntos ::),y tendremos un acceso rápido.

Ya que los métodos estáticos pueden ser llamados sin tener una instancia de la clase, la pseudovariable $this no está disponible en estos métodos. Veremos que en los métodos ahora aparecerá el operador self:: para devolver el valor.

A las propiedades estáticas no se puede acceder mediante el operador ->.

Invocar métodos no estáticos estáticamente genera una advertencia de nivel E_STRICT.

El siguiente ejemplo muestra cómo realizar una comprobación del DNI de forma estática en la clase PersonaEspaña del ejercicio que se mostró anteriormente:

```php
public static function comprobarDni($dni) {
    $letras = explode(",", "T,R,W,A,G,M,Y,F,P,D,X,B,N,J,Z,S,Q,V,H,L,C,K,E");
    if ( strlen( $dni ) < 9) {
      $dni = "0" . $dni;
    }
    $numero = intval( $dni );
    $letra = strtoupper( substr( $dni, -1 ) );

    if ( strlen( $dni ) == 9 && $letra == $letras[ $numero % 23 ] ) {
      return true;
    } else {
      return false;
    }
  }
```

El código que tendriamos que escribir para comprobar un DNI antes de insertarlo en el objeto podría ser algo parecido a esto:

```php
if ( $luis::comprobarDni( "04510533A" ) ) {
    $luis->setDni( "04510533A" );
}
```

Más información en: https://diego.com.es/late-static-binding-en-php

7. Rasgos (Traits)

Los traits son un **mecanismo de reutilización de código** en lenguajes que tienen **herencia simple,** como **PHP**.

El objetivo de los traits es reducir las limitaciones de la herencia simple permitiendo reutilizar métodos en varias clases independientes y de distintas jerarquías.

Un trait es similar a una clase, pero su objetivo es agrupar funcionalidades específicas.

Un trait, al igual que las clases abstractas, no se puede instanciar, simplemente facilita comportamientos a las clases sin necesidad de usar la herencia.

Con la palabra reservada *use* hacemos referencia a trait que queremos utilizar en nuestra clase.

Podemos **usar varios traits en la misma clase.**

También podemos **usar traits dentro de otros traits.** Eso hace que pueda haber métodos con el mismo nombre en diferentes traits o en la misma clase, por lo que tiene que haber un orden. Existe un **orden de precedencia** de los métodos disponibles en una clase respecto a los de los traits:

1. Métodos de un trait sobreescriben métodos heredados de una clase padre
2. Métodos definidos en la clase actual sobreescriben a los métodos de un trait

8. Constantes de clases

Las **constantes de clases** de clases permiten definir un valor en una clase y mantenerlo invariable durante la ejecución del script. Se escriben en mayúscula. Están asignadas una vez por clase. No usan el símbolo $ al declararlas o emplearlas.

Ejemplo:

```php
class Usuario{

    const URL_COMPLETA = "http://localhost"; //la constante se escribe en mayúsculas
    public $email;
    public $password;

// podemos mostrar el valor incluso sin crear antes un objeto. Se ha realizado a nivel de clase
echo Usuario::URL_COMPLETA;
```

9. Interfaces

A veces, es necesario que un equipo de varias personas trabajen juntas. En este caso se hace imprescindible definir pautas generales de trabajo para que el resultado final sea el esperado. Si el desarrollo consiste n programar varios objetos, el analista de la aplicación puede definir la estructura básica creando una plantilla con métodos que el objeto final debería tener obligatoriamente. Esta plantilla es una interfaz y permite establecer una clase con funciones definidas, pero sin desarrollar, que obliga a todas las clases que lo implemente a declarar estos métodos como mínimo.

Las interfaces aseguran que una clase cumple una serie de requisitos para que luego puedan utilizarse de una forma concreta. Puede ver las **Interfaces** como contratos que han de cumplir las clases que los implementan y donde se especifica qué debe cumplirse obligatoriamente.

En estos contratos definimos los métodos que habrá y en qué orden van a estar en una clase.

Son métodos vacíos que obligan a una clase a emplearlos, promoviendo así un estándar de desarrollo.

Si una clase implementa una interface, está obligada a usar todos los métodos de la misma (y los mismos tipos de argumentos de los métodos), de lo contrario dará un error fatal. Pueden emplearse más de una interface en cada clase, y pueden extenderse entre ellas mediante extends. Una interface puede extender una o más interfaces.

Todos los métodos declarados en una interface deben ser públicos.

Para definir una interface se utiliza la palabra interface, y para extenderla se utiliza implements.

Ejemplo:

```
interface Humano {
  public function setNombre(string $nombre);
  public function getNombre();
  public function setApellidos(string $apellido1, string $apellido2);
  public function getApellidos();
}
```

La interfaz del ejemplo define la estructura básica que queremos para las clases de humano que existen. Sólo se exponen qué métodos son obligatorios y después las clases que implementan esta interfaz los desarrollan.

Para obligar a una clase a implementar todos los métodos de la interfaz ponemos:

class Persona implements Humano {.....}

10. Clases abstractas

Hemos visto que una interfaz no permite crear el cuerpo de ninguna función , dejando esta tarea a las clases que lo implementan. Las clases abstractas permiten definir funciones que deben implementarse obligatoriamente en las clases que hereden y, además, permiten definir funciones completas que puedan heredarse.

Las clases abstractas son clases que no se instancian y sólo pueden ser heredadas. Mejoran la calidad del código y ayudan a reducir la cantidad de código duplicado.

Se suelen usar cuando debemos ser estrictos en los métodos que deben de existir en las clases hijas, aunque en ellas se hagan cosas diferentes.

Las clases abstractas pueden extenderse unas a otras, así como extender clases normales.

Si se define un método abstracto dentro de una clase, ésta ha de ser abstracta también. Un método abstracto define una función, pero no su implementación. Cuando una clase hereda de una abstracta, si ésta tiene un método abstracto, debe ser definido en la clase hija.

Además, estos métodos deben tener igual o mayor visibilidad que en la clase madre, Public > Protected > Private. Si en la clase madre se ha definido un método como protected, la clase hija deberá adoptarlo como protected o como public, nunca como private.

Ejemplo:

Retocamos el ejemplo anterior y en lugar de una interfaz usamos una clase abstracta para Humano, ya que tenemos claro cómo debe funcionar parte del código. También se define un método abstracto que obligatoriamente debe ser declarado en la clase que herede de esta.

```php
abstract class Humano {
  private $nombre;
  private $apellido1;
  private $apellido2;
  public function setNombre(string $nombre) {
    $this->nombre = $nombre;
  }
  public function getNombre() {
    return $this->nombre;
  }
 public function setApellidos(string $apellido1, string $apellido2) {
    $this->apellido1 = $apellido1;
    $this->apellido2 = $apellido2;
  }
  protected function getApellido1() {
    return $this->apellido1;
  }
  protected function getApellido2() {
    return $this->apellido2;
  }
  public function getApellidos() {
    return $this->getApellido1() . " " . $this->getApellido2();
}
  abstract public function getNombreCompleto();
}
```

Ahora la clase Persona debería quedar así:

```php
class Persona extends Humano{
  public function getNombreCompleto() {
    return $this->getNombre() ." ".  $this->getApellidos();
  }
}
```

11. Final

La palabra clave final afecta a métodos y clases.

Si se define una clase como final, no podrá ser heredada por ninguna clase.

Si un método se declara como final no podrá ser redefinido en ninguna clase que herede de la clase principal.

Por ejemplo podemos declarar estos dos métodos de la clase Persona como finales, para que ninguna clase que herede de esta pueda modificar su funcionamiento:

```
  final public function setDni(string $dni) {
    $this->dni = $dni;
  }
  final public function getDni() {
    return $this->dni;
  }
}
```

12. Métodos mágicos

Son métodos proporcionados por el intérprete de PHP.

- __tostring() permite devolver el objeto representado en forma de string. Es decir responde con un literar cuando se invoca a un objeto directamente, sin método asociado.

Esto hace que cuando se llame al objeto como si fuera un string, por ejemplo usando echo, el objeto devolverá lo que se defina en __toString().

- El método __call() se activa cuando se intenta llamar a un método que no es accesible públicamente.

El método __call es invocado en forma automática cuando se realiza una llamada a un método no definido explícitamente.

Lo primero que hará el intérprete será buscar un método específico con el nombre que nosotros hemos invocado, al no encontrarlo buscará una definición de un método __call. En caso de encontrarla lo invocará pasando como parámetros el nombre del método buscado y la lista de argumentos que nosotros hemos dado.

Este método se encargará de hacer una funcionalidad o incluso intentar ejecutar lo que necesitamos.

Es como un sustituto de la sobrecarga clásica en otros lenguajes

13. Clases anónimas

Las clases anónimas carecen de nombre y se definen en el mismo momento que se utilizan.

Se utilizan cuando necesitamos crear objetos sencillos que o es necesario tener disponible en todo el código, sino solo en ciertas ocasiones excepcionales (objetos desechables).

```php
interface Datos{
  public function detalle(string $nombre);
}
class Empresa {
  private $razonSocial;

  public function getRazonSocial(){
    return $this->razonSocial;
  }

  public function setRazonSocial(Datos $razonSocial){
    $this->razonSocial = $razonSocial;
  }
}
$empresa = new Empresa;
$empresa->setRazonSocial( new class implements Datos {
    public function detalle(string $nombre){
      echo $nombre;
    }
  }
);
$empresa->getRazonSocial()->detalle("Anaya Multimedia");
```

14. Organizando el código de las clases

- **Carga automática de clases** / Autoload

Para no tener que cargar las clases y librerías, de forma manual en nuestro proyecto, podemos escribir un fichero con una función que cargue los archivos automáticamente.

A medida que crece el código será necesario ir añadiendo require() en cada uno de los archivos. PHP incorpora una función que realiza el trabajo por nosotros y que permite cargar los archivos de clases a medida que se van necesitando.

Ejemplo:

```
k?php

use clases\Humano\Persona\PersonaUSA\PersonaUSA;

spl_autoload_register(function ($clase) {

    $directorio_clase = str_replace('\\', '/', $clase);
        if (file_exists($directorio_clase . '.php')) {
          require $directorio_clase . '.php';
        }
    });

$persona = new PersonaUSA();
```

- **Espacios de nombre** .

Es muy usado en los Frameworks de PHP.

Nos permite organizar el código, agrupando en paquetes distintas clases. O lo que es lo mismo, permite agrupar clases mediante un nombre descriptivo, de la misma manera que una carpeta agrupa un conjunto de ficheros. De esta manera pueden existir varias clases distintas con el mismo nombre.

Por ejemplo, podemos tener un espacio de nombres llamado "Loterias" con una clase llamada Loto y otro espacio de nombres llamado "Flores" con una clase también llamada Loto. Y además, las dos son totalmente diferentes en su declaración.

Para crear un espacio de nombres, hay que añadir al principio del fichero la palabra reservada namespace seguida de un nombre descriptivo. Todas las clases declaradas en el archivo pertenecerán al espacio de nombre.

Una buena práctica es que cada clase esté en un único archivo que a su vez estará contenida en un directorio con el nombre del namespace.

Resumiendo:

- Para hacer un paquete usare NameSpaces en la clase.
- La palabra reservada use nos permite cargar el espacio de nombres.
- Esto nos permite también usar clases de otros paquetes que tengan el mismo nombre.

Para que el código no se confunda renombraremos con un alias las clases.

- **Comprobar si existe una clase o un método**

Para comprobar si una clase está cargada o si existe existen métodos, como por ejemplo:

class_exists()

Para comprobar si existe un método get_class_methods()