PHP Orientado a objetos

Constantes de clase

- Se definen usando la palabra clave const
- Pueden ser públicas, privadas o protegidas, a partir de PHP 7.1.0

```
class MiClase{
    const MICONSTANTE= 123;
    private const USERNAME = “abc123.”
}
```

Al igual que las propiedades estáticas, se acceden vía nombre de la clase y el operador ::

```
MiClase::MICONSTANTE
```

Constructor

- **Establece ciertos aspectos del objeto en el momento de la instanciación.**

```
class MiClase{
  Var $color;
  function __construct($c){
     $this->color=$c;
     // hacer otras cosas
     echo “El objeto se ha creado”;
}
}

// Inicializa color a rojo y muestra “El objeto se ha creado”
$obj1= new MiClase('rojo');
```

Constructor

- **Podemos pasar argumentos a constructores, como cualquier otro método.**
- **La función __construct() nunca declarará tipo de retorno, pues siempre retornará una instancia de la clase. Por ejemplo, function __construct() : void {} hará que el objeto no funcione.**

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

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

	public function mostrarDetalles() {
		echo "$this->nombre $this->apellido, edad $this->edad años";
	}

} // end class Persona

$p= new Persona ("Juan", "de Bombón", 77);
$p->mostrarDetalles(); // Muestra “Juan de Bombón, edad 77 años”
```

Constructor

- **Mismo nombre que la clase como método constructor (en versiones anteriores a PHP5). Se considera obsoleto y desaparecerá en futuras versiones.**
- **Siempre usaremos __construct() en código nuevo.**

```
class Miclase{
      function Miclase(){
         Echo “Instanciado Miclase”;
       }
}

$obj= new Miclase;   //Muestra “Instanciando Miclase” en PHP3 y 4.
```

Constructor

- **Si una clase NO define un constructor, la clase heredará el constructor del ancestro más cercano que lo haya definido.**

```
class Abuelo{
  function __construct(){
      echo “Instanciando Abuelo”;
  }
}

class Padre extends Abuelo{
}

class Hijo extends Padre{
}

$obj= new Hijo;      //Muestra el texto “Instanciando Abuelo”
```

Constructor

- **Si es necesario, podemos hacer que una clase hija llame al constructor de su padre con**

  parent::__construct()

Conclusión: El constructor padre no es llamado implícitamente si la clase hija define un constructor.

```
class Padre {
   function  __construct() {
      echo "Ejecutando construct<br/>";
   }
   function Padre() {
      echo "Iniciando padre <br/>";
   }
}
class Hijo extends Padre {
   function   __construct() {
      //otras_cosas;
      parent::__construct();
   }
}
// Muestra “Iniciando padre” en PHP4
// Muestra 'Ejecutando construct' en PHP5;
$hijo = new Hijo();
```

Destructor

- Contiene código que se ejecutará justo antes de que el objeto sea eliminado
- Se crea de modo similar a un constructor.
- no acepta argumentos.

```
function __destruct(){
    // instrucciones;
}
```

- Una clase hijo puede heredar un destructor de la clase padre si no implementa uno propio, al igual que un constructor.
- Una clase puede explícitamente llamar al destructor de la clase padre con parent::__destruct()

Destructor

Ejemplo de destructor

```
class Persona {
	public function guardar() {
		echo "Guardando el objeto en la base de datos";
	}
	public function   __destruct() {
		$this -> guardar();
	}

} // end class Persona

$p1= new Persona;
unset ($p1);
$p2= new Persona;
die (“¡Algo ha ido mal!”);
```

Tanto unset($p1) (por destrucción del objeto) como die (por finalización del script) activan el destructor.

Sobrecargar objetos

PHP permite crear dinámicamente propiedades y métodos. Se implementa por medio de métodos mágicos que se pueden añadir a la clase.

Métodos 'mágicos'

- **__get($n): Se llama cuando el código llamante intenta leer una propiedad invisible del objeto.**
- **__set($n, $v): Se llama cuando el código llamante intenta escribir una propiedad invisible del objeto.**
- **__call($f, $p): Se llama cuando el código llamante intenta llamar a un método invisible.**

Nota: Invisible significa que la propiedad o el método no es visible para el código llamante,bien sea porque no existe en la clase, o porque es privado o protegido.

Sobrecarga: método1

```
class OverloadedClass {
        private function Param2($a, $b) { echo "Param2($a,$b)\n"; }
        private function Param3($a, $b, $c) { echo "Param3($a,$b,$c)\n";}
        public function Param() {
            $p = func_get_args();
            try {
                switch (sizeof($p)) {
                    case 2 : {
                            $this->Param2($p[0], $p[1]); break;
                        }
                    case 3 : {
                            $this->Param3($p[0], $p[1], $p[2]); break;
                        }
                    default : throw new Exception('Llamada a método ' .
get_class($this) .
                            '::Param, con número inadecuado de parámetros');
                }
            } catch (Exception $e) {
                echo "Entrando en la excepción:" . $e->getMessage();
            }
        }
    }
    $o = new OverloadedClass();
    $o->Param(4, 5);
    $o->Param(4, 5, 6);
```

Sobrecarga: método2

```
class OverloadedClass {

        public function __call($f, $p) {
            if (method_exists($this, $f.sizeof($p)))
                 return call_user_func_array(array($this, $f.sizeof($p)), $p);
                // function does not exists~
                throw new Exception('Tried to call unknown method
'.get_class($this).'::'.$f);
        }
        private function Param2($a, $b) {
                echo "Param2($a,$b)\n";
        }
        private function Param3($a, $b, $c) {
                echo "Param3($a,$b,$c)\n";
        }
}
$o = new OverloadedClass();
$o->Param(4,5);
$o->Param(4,5,6);
```

Otros métodos de sobrecarga

PHP proporciona otros métodos de sobrecarga, aunque no son tan utilizados como los anteriores.

Métodos 'mágicos'

- **__isset($n): Se llama cuando el código llamante intenta llamar a la función isset() en una propiedad invisible.**
- **__unset($n): Se llama cuando el código llamante intenta eliminar una propiedad invisible del objeto con la función unset(). No debería devolver un valor, pero debería "hacer lo necesario" para no establecer la propiedad (si aplica).**
- **__callStatic($f, $p): Se llama cuando el código llamante intenta llamar a un método estático invisible.**

Nota: Invisible significa que la propiedad o el método no es visible para el código llamante, bien sea porque no existe en la clase, o porque es privado o protegido.

Heredar

- **PHP permite herencia simple, por tanto, heredar propiedades y métodos de la clase padre.**
- **NO permite HERENCIA MÚLTIPLE.**
  - **Se puede simular herencia múltiple por medio de 'traits' (a partir de la versión 5.4). Ver http://programandolo.blogspot.com.es/2013/08/php-orientado-objetos-herencia-multiple.html**
- **Podemos añadir propiedades y métodos adicionales.**
- **Usaremos la palabra clave *extends* para heredar de una clase padre.**

```
class Padre {
        public function unMetodo ()  { echo “Función unMetodo de Padre”;}
}

class Hija extends Padre {

}
$h= new Hija();
$h->unMetodo();   // Muestra “Función unMetodo de Padre”.
```

Anular herencia

- **Cuando queremos sustituir en la clase hijo el comportamiento de un método de la clase padre, crearemos en la clase hijo un método con el mismo nombre que el de la clase padre.**

```
class Padre {
        public function unMetodo ()  { echo “Función unMetodo de Padre”;}
}

class Hija extends Padre {
        public function unMetodo() { echo "Función unMetodo de Hija"; }
}

$p = new Padre;
$p->unMetodo();   // Muestra “Función unMetodo de Padre”
$h= new Hija();
$h->unMetodo();   // Muestra “Función unMetodo de Hija”.
```

Acceder a método padre

- **Podemos llamar a un método anulado de la clase padre dentro del método de la clase hijo, escribiendo parent:: antes del nombre del método.**
  parent::unMetodo();

```
class Padre {
        public function unMetodo ()  { echo “Función unMetodo de Padre”;}
}

class Hija extends Padre {
       public function unMetodo() {
          // hacer cosas;
          parent::unMetodo();
      }
}

$h= new Hija();
$h->unMetodo();   // Muestra “Función unMetodo de Padre”.
```

Bloquear herencia en clases

- **Para que una clase no pueda ser 'extensible' añadiremos la palabra clave *final* antes de la palabra clave *class*.**

```
final class NoSePuedeHeredar {
}
```

Si luego intentásemos heredar esta clase con

```
class ClaseHijo extends NoSePuedeHeredar {
}
```

Nos daría un error.

Bloquear herencia en métodos

- **Para impedir que un método hijo anule un método de la clase padre, añadiremos en el método de la clase padre la palabra clave *final* antes de la definición del método.**

public final function metodoNoSustituible();

Crear ese método en la clase hijo, por ejemplo para anular el método definido en la clase padre:

Nos daría un error.

Nota: Las propiedades no pueden declararse como ‘final’. Solo las clases y los métodos

Clases abstractas

- **Una clase abstracta se declara anteponiendo la palabra clave *abstract* a la palabra clave *class*.**
- **No es posible instanciar una clase abstracta.**
- **Cualquier clase hija debe implementar los métodos abstractos de una clase abstracta (a menos que la clase hija sea declarada como abstracta).**
- **La clase abstracta contendrá los métodos (comportamiento) comunes a todos sus hijos, mientras dejará como métodos abstractos aquellos que son específicos a cada uno de sus hijos.**
- **Podemos mezclar métodos abstractos y no abstractos en una clase abstracta.**

Nota: Lo contrario de una clase abstracta, es decir, una clase que implementa todos los métodos de una clase padre abstracta, se denomina una clase concreta.

Métodos abstractos

- **Un método abstracto se declara anteponiendo la palabra clave *abstract* a la palabra clave *function*.**
- **No incluye ningún código que implemente el método.**
- **Si se declara uno o más métodos como abstractos, también se debe declarar la clase como abstracta.**

Incluir ejemplo de clase abstracta con la clase Forma, FormaInfo, y las subclases Rectangulo, Triangulo y Circulo.

Interfaces

- **Se crean igual que una clase pero sustituyendo la palabra clave '*class*' por la palabra clave ‘interface’. Luego incluirá los métodos de la interfaz.**
- **No pueden contener propiedades, pero admiten definición de constantes.**
- **Todos los métodos en una interfaz deben ser públicos (de lo contrario, no sería una interfaz).**
- **Solo declaran métodos y no pueden contener ningún código. Solo la signatura de la función.**
- **Pueden ‘extends’ otras interfaces pero no pueden ‘implements’.**
- **Una clase puede implementar más de una interfaz (siempre que los métodos declarados en las interfaces no colisionen). Esto permite crear unas clases muy potentes y adaptables que se pueden utilizar en muchas ocasiones.**
- **Para que una clase implemente un interfaz (o más) añadiremos la palabra clave** *implements* **después del nombre de la clase. Si implementa más de una interfaz, estas irán separadas por comas.**

Interfaces

- **Actualmente 7.4+ una clase puede implementar dos interfaces que especifiquen un método con el mismo nombre, siempre y cuando los métodos duplicados tengan la misma signatura.**

Traits

-

Traits

-

Denominación de archivos

Recomendaciones:

- **Cada clase estará definida en un script aparte: una clase por archivo.**
- **El nombre del archivo coincidirá con el de la clase.**
- **La denominación del archivo será** *nombredelaclase.php***, por ejemplo.**
- **Todas las clases se guardarán en un directorio de clases.**
- **Cuando instanciemos una clase la incluiremos previamente en el archivo con un include(“***clases/nombredelaclase.php***”).**

Si utilizamos un convenio adecuado podemos beneficiarnos de la autocarga.

Autoload

Siempre que un script intenta instanciar un nuevo objeto desde una clase no cargada previamente, se llama automáticamente a la función 'mágica' __autoload(), pasando como argumento el nombre de la clase buscada.

Nota: Esta función fue reemplazada por spl_autoload_register() y está 'deprecated', pero permite introducir el concepto de autocarga de una manera más sencilla. **A partir de la versión php 8.0 dará un 'Fatal Error'.**

```
function  __autoload( $nomclase){

   // la siguiente línea se adaptará para que genere una cadena adecuada para
   // encontrar la clase.

   $nomclase= str_replace ("..", "", $nomclase);
   require_once("clases/$nomclase.php");
};
```

Autoload

Cuando el motor PHP tiene que instanciar un objeto, por ejemplo, new Persona, comprueba si la clase Persona se ha definido (y está cargada).

Si no la encuentra, llama a la función __autoload(), que deberíamos definir previamente. Esta, a su vez, incluye y ejecuta el archivo Persona.php dentro de la carpeta “clases” y permite que se cree el objeto Persona.

Si el motor no puede encontrar una función __autoload(), o si su función __autoload() no puede cargar la clase Persona, el script termina con un error “Class 'Persona' not found”.

Autoload

Ver:

a) spl_autoload_register

b) estandares PSR-0 y PSR-4, del php-fig

c)https://github.com/jatubio/ja_duilio_curso-laravel-5/wiki/Autocarga-de-clases-en-Laravel-(Autoload)

Autoload

Parámetros válidos en la llamada a la función *spl_autoload_register()***:**

- registrando la función de autocarga
spl_autoload_register('mi_cargador1');

- registrando más de una función de autocarga, simultáneamente
spl_autoload_register('mi_cargador1' | 'mi_cargador2');

- Con función anónima
spl_autoload_register(function($class){//código de autocarga});

- Con un método de clase
spl_autoload_register('Miclase', 'mi_cargador');

Serialización

- **Podemos almacenar un objeto como dato textual en memoria**

- **PHP proporciona 2 funciones para serializar un objeto:**
  - **serialize();**
  - **unserialize()**

Serialización: otras funciones

Otras funciones útiles cuando usamos serialización son:

- **__sleep()**
- **__wakeup();**

get_object_vars(object $obj)

- **Devuelve un array asociativo con las propiedades no estáticas y accesibles del objeto especificado por $obj en el ámbito que se ejecuta.**
- **Si una propiedad no tiene asignado valor, será devuelta con un valor NULL.**

Retornará NULL si el parámetro no es un objeto.

```
<?php
class foo {
    private $a;
    public $b = 1;
    public $c;
    private $d;
    static $e;
    public function prueba() {
        var_dump(get_object_vars($this));
    }
}
$prueba = new foo;
var_dump(get_object_vars($prueba));

$prueba->prueba();
?>
```

El resultado sería:

```
array(2) {
  ["b"]=>  int(1)
  ["c"]=>  NULL
}
array(4) {
  ["a"]=>  NULL
  ["b"]=>  int(1)
  ["c"]=>  NULL
  ["d"]=>  NULL
}
```

get_object_vars (ejemplo 2)

```php
<?php

class Potatoe {
   public $skin;
   protected $meat;
   private $roots;

   function __construct ( $s, $m, $r ) {
      $this->skin = $s;
      $this->meat = $m;
      $this->roots = $r;
   }
}

$Obj = new Potatoe ( 1, 2, 3 );

echo "<pre> \n";
echo "Using get_object_vars:\n";

$vars = get_object_vars ( $Obj );
print_r ( $vars );

echo "\n\nUsing array cast:\n";

$Arr = (array)$Obj;
print_r ( $Arr );
echo “</pre>”;
?>
```

El resultado sería:

```
Using get_object_vars:
Array
(
    [skin] => 1
)
Using array cast:
Array
(
    [skin] => 1
    [ *meat] => 2
    [ Potatoeroots] => 3
)
```

Determinar la clase: get_class()

string get_class ([object $obj = NULL])

Devuelve el nombre de la clase a la que pertenece el objeto pasado como parámetro.

No es necesario pasar parámetro si se utiliza dentro de la clase.

Ejemplo

```
class MiClase {}
$obj = new MiClase();
echo get_class($obj); // Muestra “MiClase”;
```

Determinar la clase

Ejemplo

```
class Fruta {}
class Sinhueso extends Fruta {}
class ConHueso extends Fruta {}

function comerUnaFruta (array $frutaAComer) {
foreach ($frutaAComer as $itemDeFruta) {
    If (get_class($itemDeFruta)== “SinHueso” || get_class($itemDeFruta) == “ConHueso” {
        Echo “Comiendo la fruta -! <br/>”;
    }
}
} // end function comerUnaFruta
```

get_class()

Ejemplo de get_class()

```
<?php
abstract class bar {

   public function __construct()
      {
         var_dump(get_class($this));
         var_dump(get_class());
      }
}

class foo extends bar {
}
// la clase ‘bar’ no puede ser instanciada por ser abstracta.
new foo;
// Devuelve:
// string 'foo'   ((length = 3)
// string 'bar'   ((length = 3)
?>
```

Pertenencia a una clase

- **$obj instanceof NomClase;**

Determina si $obj es descendiente de NomClase.

```
class Abuelo{}
class Padre extends Abuelo{}
class HijoA extends Padre{}
class HijoB extends Padre{}

$hb= new HijoB;

echo "<br />Es hijo B instancia de Padre? " . ($hb instanceof Padre ? 'cierto':
'falso');
echo "<br />Es hijo B instancia de Abuelo? " . ($hb instanceof Abuelo ? 'cierto':
'falso');
echo "<br />Es hijo B instancia de HijoA? " . ($hb instanceof HijoA ? 'cierto':
'falso');
echo "<br />Es hijo B instancia de HijoB? " . ($hb instanceof HijoB ? 'cierto':
'falso');
echo "<br />Hijo B pertenece a la clase  '" . get_class($hb) . '\";
```

Clases anónimas

- **Son útiles cuando es necesario crear objetos sencillos y únicos.**
- **Pueden pasar argumentos a través de constructores, extender otras clases, implementar interfaces y utilizar traits al igual que una clase normal.**

Clases anónimas

-

Clases anidadas

http://stackoverflow.com/questions/16424257/nested-or-inner-class-in-php

Curso 2021-2022