

Desarrollo Web en Entorno Servidor

4.- Objetos en PHP II

IES Severo Ochoa



Índice

- Metodos encadenados
- Traits
- Métodos mágicos
- Namespaces
- Gestión de Errores
 - Niveles de Error
 - Excepciones
- SPL

Métodos encadenados

- Programación funcional
- Sobre un objeto se realizan varias llamadas
- Los métodos mutadores (que cambian el contenido) devuelven `$this`.

```
$p1 = new Libro();  
$p1->setNombre("Harry Potter");  
$p1->setAutor("JK Rowling");  
echo $p1;
```

```
// Method chaining  
$p2 = new Libro();  
$p2->setNombre("Patria")->setAutor("Aramburu");  
echo $p2;
```

POO con encadenamiento

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

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

Traits

- Permite reutilizar código
- No se puede instanciar
- Solución para la herencia múltiple

```
trait Nombre {  
    ...  
}
```

- Dentro de la clase → **use** NombreTrait
- Similar a copiar y pegar el código dentro la clase que lo usa

trait Log

```
trait Log {  
  private $DEBUG = "<span style='color:blue'>debug</span>";  
  private $WARNING = "<span style='color:orange'>warning</span>";  
  private $ERROR = "<span style='color:red'>error</span>";  
  
  public function logDebug($mensaje) : void {  
    $this->log($this->DEBUG, $mensaje);  
  }  
  
  public function logWarning($mensaje) : void {  
    $this->log($this->WARNING, $mensaje);  
  }  
  
  public function logError($mensaje) : void {  
    $this->log($this->ERROR, $mensaje);  
  }  
  
  public function log($nivel, $mensaje) : void {  
    echo $nivel.":". $mensaje."<br />";  
  }  
}
```

Usando el log

```
class Libro {  
    private string $nombre;  
    private string $autor;  
  
    use Log;  
  
    public function getNombre() : string {  
        return $this->nombre;  
    }  
  
    public function setNombre(string $nombre) : Libro {  
        $this->nombre = $nombre;  
        $this->logDebug("Modificado nombre del libro a $nombre");  
        return $this;  
    }  
    //...  
}
```

Ejemplo Trait Singleton

```
<?php
trait Singleton {
    private static $instance;

    public static function getInstance() {
        if ( is_null( self::$instance ) ) {
            self::$instance = new self();
        }

        return self::$instance;
    }
}

class Cero {
}
class Uno extends Cero {
    use Singleton;
}
class Dos extends Cero {
    use Singleton;
}

$a = Uno::getInstance();
$b = Dos::getInstance();
```


Magic methods I

- Todas las clases PHP ofrecen un conjunto de métodos que se pueden sobrescribir para sustituir su comportamiento.
- <https://www.php.net/manual/es/language.oop5.magic.php>
- `__construct()`
- `__destruct()` → se invoca al perder la referencia
 - Cerrar una conexión a BD, cerrar un fichero
- `__toString()` → representación del objeto como cadena

Magic Methods II

- `__get(propiedad)` , `__set(propiedad, valor)`
 - Permitiría acceder a las propiedad privadas
 - Mejor codificar los getter/setter
- `__isset(propiedad)` , `__unset(propiedad)`
- `__sleep()` , `__wakeup()`
 - Se ejecutan al recuperar (unserialize) o almacenar un objeto que se serializa (serialize).
 - Permite definir que propiedades se serializan
- `__call()` , `__callStatic()`
 - Se ejecutan al llamar a un método que no es público. Permiten sobrecargar métodos

Namespaces

- \geq PHP 5.3
- Similar a los paquetes Java
- Permite organizar las clases/interfaces/traits, funciones y/o constantes.
- Recomendable: un sólo namespace por archivo y colocarlos en carpetas
 - Igual que Java

Declaración

- Se declara en la primera línea
- **namespace** Nombre;
- namespace Nombre\Subnombre;

```
<?php
namespace Dwes\Ejemplos;

const IVA = 0.21;

class Producto {
    public $nombre;

    public function muestra() : void {
        print "<p>Prod:" . $this->nombre . "</p>";
    }
}
```

\Dwes\Ejemplos\Producto.php

Uso de Namespaces

- Para acceder a un elemento (constante, función, clase ...):
 - Primero hemos de hacerlo disponible:
`include/require.`
 - Después podemos acceder a él.
 - **Si tenemos dos recursos en el mismo NS, podemos acceder a él directamente (sin cualificar)**
- El acceso puede ser:
 - Sin cualificar: `recurso`
 - Cualificado: `rutaRelativa/recurso`
 - Totalmente cualificado: `/rutaAbsoluta/recurso`

Acceso

```
<?php
namespace Dwes\Ejemplos;

include_once("Producto.php");

echo IVA; // sin cualificar
echo Utilidades\IVA; // daría error, no existe utilidades
// cualificado -> \Dwes\Ejemplos\Utilidades\IVA
echo \Dwes\Ejemplos\IVA; // totalmente cualificado

$p1 = new Producto(); // \Dwes\Ejemplos\Producto
$p2 = new Model\Producto(); // daría error, no existe model
// \Dwes\Ejemplos\Model\Producto
$p3 = new \Dwes\Ejemplos\Producto();
// \Dwes\Ejemplos\Producto
```

use

- Para evitar la referencia cualificada podemos declarar el uso.

- Se hace en la cabecera, tras el namespace

```
use const nombreCualificadoConstante  
use function nombreCualificadoFuncion  
use nombreCualificadoClase
```

- También podemos renombrar elementos

```
use nombreCualificadoClase as NuevoNombre
```

Ejemplo use

```
<?php  
include_once("Dwes\Ejemplo\Producto.php");
```

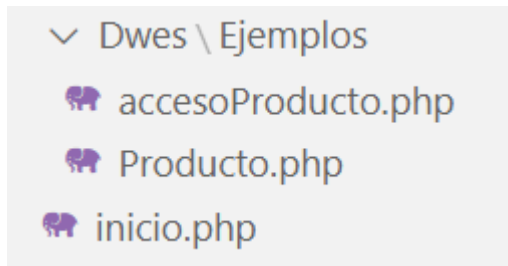
\inicio.php

```
use const Dwes\Ejemplos\IVA;  
use \Dwes\Ejemplos\Producto;  
  
echo IVA;  
$p1 = new Producto();
```

- `use` permite acceder sin cualificar a recursos que están en otro NS
 - Si estamos en el mismo NS, no necesitamos `use`

Organización

- Crear NS anidados para separar funcionalidades
- Crear una organización de carpetas similar



```
Dwes > Ejemplos > Producto.php > ...
1  <?php
2  namespace Dwes\Ejemplos;
3
4  const IVA = 0.21;
5
6  class Producto {
7      public $nombre;
8
9      public function muestra() : void {
10         print "<p>Prod:" . $this->nombre . "</p>";
11     }
12 }
```

```
inicio.php > ...
1  <?php
2  include_once("Dwes/Ejemplos/Producto.php");
3
4  use const Dwes\Ejemplos\IVA;
5  use Dwes\Ejemplos\Producto;
6
7  echo IVA;
8  $p1 = new Producto();
```

Hasta ahora...

- Colocamos cada recurso en un fichero aparte.
 - En la primera línea su NS
 - Si usamos otros recursos: `include_once` de esos recursos (clases, interfaces, traits, etc...)
 - Si los recursos están en un NS diferente al que estamos, usaremos `use` con la ruta completa para luego utilizar referencias sin cualificar.
- Cada recurso debe incluir todos los recursos que emplee
 - Las clase de la que hereda, interfaces que implemente, clases utilizadas/recibidas como parámetros, etc...

Autoload

- Permite cargar las clases (no las constantes ni las funciones) que se van a utilizar y evitar tener que hacer el `include_once` de cada una de ellas.
- **`spl_autoload_register`**

```
spl_autoload_register( function( $NombreClase ) {  
    include_once $NombreClase.'.php';  
} );
```

- Antes se hacía con `__autoload()`
 - *Deprecated* desde PHP7.2

Organizando con autoload

- Colocamos el código dentro de una carpeta `app`
 - Más tarde colocaremos las pruebas en `test`
 - Librerías en `vendor` → Composer

> OPEN EDITORS

▼ NSAL

▼ app \ Dwes \ Ejemplos

🐘 accesoProducto.php

🐘 Producto.php

> test

> vendor

🐘 autoload.php

🐘 inicio.php

🐘 inicio.php > ...

1 <?php

2 **include**("autoload.php");

3

4 **use** Dwes\Ejemplos\Producto;

5

6 **echo** Producto::IVA;

7 **\$p1** = **new** Producto();

8

9

<?php

spl_autoload_register(**function**(\$NombreClase) {

| **include_once** "app/".\$NombreClase.'.php';

});

autoload.php

Gestión de errores

- PHP clasifica los errores que ocurren en diferentes niveles.
- Cada nivel se identifica con una constante.
 - `E_ERROR`: errores fatales, no recuperables. Se interrumpe el script.
 - `E_WARNING`: advertencias en tiempo de ejecución. El script no se interrumpe.
 - `E_NOTICE`: avisos en tiempo de ejecución.
- Listado de constantes:
<https://www.php.net/manual/es/errorfunc.constants.php>

Configuración de errores

- A nivel de `php.ini`:
- `error_reporting`: indica los niveles de errores a notificar
 - `error_reporting = E_ALL & ~E_NOTICE`
 - Todos los errores menos los avisos en tiempo de ejecución.
- `display_errors`: indica si mostrar o no los errores por pantalla
 - En entornos de producción es común ponerlo a `off`

Configurando errores en código

- `error_reporting(codigo)`
 - Controla qué errores notificar
- `set_error_handler(nombreManejador)`
 - Indica que función se invocará cada vez que se encuentre un error
 - El manejador recibe como parámetros el nivel del error y el mensaje

Ejemplo manejo errores

```
<?php
error_reporting(E_ALL & ~E_NOTICE & ~E_WARNING);
$resultado = $dividendo / $divisor;

error_reporting(E_ALL & ~E_NOTICE);
set_error_handler("miManejadorErrores");
$resultado = $dividendo / $divisor;
restore_error_handler(); // vuelve al anterior

function miManejadorErrores($nivel, $mensaje) {
    switch($nivel) {
        case E_WARNING:
            echo "<strong>Warning</strong>: $mensaje.<br/>";
            break;
        default:
            echo "Error de tipo no especificado: $mensaje.<br/>";
    }
}
```

Error de tipo no especificado: Undefined variable: dividendo.
Error de tipo no especificado: Undefined variable: divisor.
Error de tipo Warning: Division by zero.

Excepciones

- PHP ≥ 5
- Similar a Java
 - try / catch / finally
 - throw new Exception
 - Adjuntar mensaje[, código][y excepción] que lo ha provocado.

```
<?php
try {
    if ($divisor == 0) {
        throw new Exception("División por cero.");
    }
    $resultado = $dividendo / $divisor;
} catch (Exception $e) {
    echo "Se ha producido el siguiente error: ".$e->getMessage();
}
```

Exception

- Clase padre de todas las excepciones
- **Constructor:** `mensaje[, codigoError]`
`[, excepcionPrevia]`
- **A partir de un objeto** `Exception`:
 - `getMessage()`
 - `getCode();`

Creando excepciones

- Heredan de `Exception`
- Consejos:
 - Sobrecargar el constructor y llamar al constructor del padre
 - Sobrecargar `__toString` ¿y llamar al método del padre?

```
<?php
class MiExcepcion extends Exception {
    public function __construct($msj, $codigo = 0, Exception $previa = null) {
        // código propio
        parent::__construct($msj, $codigo, $previa);
    }
    public function __toString() {
        return __CLASS__ . ": [{$this->code}]: {$this->message}\n";
    }
    public function funciónPersonalizada() {
        echo "Una función personalizada para este tipo de excepción\n";
    }
}
```

Excepciones múltiples

- Se pueden usar excepciones múltiples para comprobar diferentes condiciones.
 - De más específica a más general.

```
<?php
$email = "ejemplo@ejemplo.com";
try {
    // Comprueba si el email es válido
    if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE) {
        throw new MiExcepcion($email);
    }
    // Comprueba la palabra ejemplo en la dirección email
    if(strpos($email, "ejemplo") !== FALSE) {
        throw new Exception("$email es un email de ejemplo");
    }
} catch (MiExcepcion $e) {
    echo $e->miFuncion();
} catch (Exception $e) {
    echo $e->getMessage();
}
```

¿Qué pasaría...?

```
<?php
class MainException extends Exception {}
class SubException extends MainException {}

try {
    throw new SubException("Lanzada SubException");
} catch (MainException $e) {
    echo "Capturada MainException " . $e->getMessage();
} catch (SubException $e) {
    echo "Capturada SubException " . $e->getMessage();
} catch (Exception $e) {
    echo "Capturada Exception " . $e->getMessage();
}
```

Capturada MainException Lanzada SubException

Capturando múltiples Excepciones

- Operador |

```
<?php
class MainException extends Exception {}
class SubException extends MainException {}

try {
    throw new SubException("Lanzada SubException");
} catch (MainException | SubException $e) {
    echo "Capturada Exception " . $e->getMessage();
}
```

Relanzar excepciones

- Capturar una excepción de sistema y lanzar una de aplicación.

```
<?php
class AppException extends Exception {}

try {
    // Código de negocio que falla
} catch (Exception $e) {
    throw new AppException("AppException: ".$e->getMessage(),
        $e->getCode(), $e);
}
```

Throwable

- PHP ≥ 7
- Interfaz que implementan tanto los errores como las excepciones

```
<?php
try {
    // tu codigo
} catch (Throwable $e) {
    echo 'Forma de capturar errores y excepciones a la vez';
}
```


Error

- PHP ≥ 7
- Clase que agrupa todos los errores fatales.

```
try {  
    // Genera una notificación que no se captura  
    echo $variableNoAsignada;  
    // Error fatal que se captura  
    funcionQueNoExiste();  
} catch (Error $e) {  
    echo "Error capturado: " . $e->getMessage();  
}
```

- Standard PHP Library
- <https://www.php.net/manual/es/book.spl.php>
- Estructuras de datos
 - Pila, cola, cola de prioridad, lista doblemente enlazada, etc...
- Conjunto de iteradores diseñados para recorrer estructuras agregadas
 - arrays, resultados de bases de datos, árboles XML, listados de directorios, etc.
 - <https://diego.com.es/tutorial-de-la-libreria-spl-de-php>

Excepciones SPL

- `LogicException` (**extends** `Exception`)
 - `BadFunctionCallException`
 - `BadMethodCallException`
 - `DomainException`
 - `InvalidArgumentException`
 - `LengthException`
 - `OutOfRangeException`
- `RuntimeException` (**extends** `Exception`)
 - `OutOfBoundsException`
 - `OverflowException`
 - `RangeException`
 - `UnderflowException`
 - `UnexpectedValueException`

¿Alguna pregunta?