DESARROLLO WEB CON PHP 7 Y MYSQL UTILIZANDO MVC

Clases y objetos

Ponente: Alejandro Amat Reina

Introducción

- Con PHP podemos utilizar dos estilos de programación: estructurada y orientada a objetos
- Ejemplo conexión BBDD con mysqli:

```
// utilizando programación estructurada
$dwes = mysqli_connect('localhost', 'user', 'password', 'bbdd');

// utilizando POO
$dwes = new mysqli();
$dwes->connect('localhost', 'user', 'password', 'bbdd')
```

Evolución de la POO en PHP

- PHP no se diseñó con características de orientación a objetos
- En la versión 3 se empezaron a introducir algunos rasgos de POO
- Esto se potenció en la versión 4, aunque todavía de forma muy rudimentaria
- Por ejemplo, en PHP4:
  - Los objetos se pasan siempre por valor, no por referencia
  - En una clase todos los miembros son públicos
  - No existen los interfaces
  - No existen métodos destructores
- A partir de la versión 5, se reescribió y potenció el soporte de orientación a objetos del lenguaje
- La versión 7 se ha reescrito de nuevo, mejora el rendimiento y añade ciertas funcionalidades interesantes como es el caso del tipado de las funciones

Soporte de POO en PHP

- A partir de PHP 5 se soporta:
  - Métodos estáticos
  - Métodos constructores y destructores
  - Herencia
  - Interfaces
  - Clases abstractas
- No se soporta:
  - Herencia múltiple
  - Sobrecarga de métodos (incluidos los métodos constructores)
  - Sobrecarga de operadores

Clases en PHP

- Similar a otros lenguajes como Java o C#
- Se declaran con la palabra reservada class
- Entre llaves se declaran los atributos y métodos, que pueden ser privados, públicos o protegidos
- Por defecto son públicos
- Buenas prácticas (aunque no obligatorias):
  - El nombre de la clase siempre empezará por mayúsculas
  - Siempre definiremos cada clase en un fichero que llamaremos nombre_clase.class.php
  - Los atributos siempre serán privados o protegidos (crearemos getters y setters para acceder a ellos)

acceder a sus atributos y métodos

- Al igual que en otros lenguajes usamos new para instanciar objetos:

```
$p = new Producto;
```

- Tendremos que hacer un require del fichero de la clase
- Para acceder desde un objeto a sus atributos o métodos, utilizamos el operador flecha:
  -
```
$p->nombre = 'Samsung Galaxy S';
$p->muestra();
```

El objeto $this

- Cuando desde un objeto se invoca un método de la clase, a éste se le pasa siempre una referencia al objeto que hizo la llamada
- Esta referencia se almacena en la variable $this
- Ejemplo:

```
print "<p>" . $this->codigo . "</p>";
```

Constantes de clase

- En una clase se pueden definir constantes, utilizando la palabra const
- Para acceder a las constantes de una clase, se debe utilizar el nombre de la clase (o self si estamos dentro de la clase) y el operador ::

```php
class DB
{
    const USUARIO = 'dwes';
    public function muestraUsuario()
    {
        echo self::USUARIO; // desde dentro de la clase
    }
}
echo DB::USUARIO; // desde fuera de la clase
```

Constantes mágicas

- Hay nueve constantes mágicas que cambian dependiendo de dónde se emplean
- Por ejemplo, el valor de__LINE__ depende de la línea en que se use en el script
- Se resuelven durante la compilación, a diferencia de las constantes normales que lo hacen durante la ejecución
- Son sensibles a mayúsculas
- Ver:
  - http://php.net/manual/es/language.constants.predefined.php

Métodos mágicos

- Métodos predefinidos que son llamados automáticamente en determinadas circunstancias.
- Los nombres de los métodos __construct(), __destruct(), __call(), __callStatic(), __get(), __set(), __isset(), __unset(), __sleep(), __wakeup(), __toString(), __invoke(), __set_state(), __clone() y __debugInfo() son mágicos en las clases PHP
- No se puede tener métodos con estos nombres en ninguna clase a menos que se desee la funcionalidad mágica asociada a estos

Constructores y destructores

- *void* __construct ([*mixed* $args = "" [, $...]])
  - Sólo puede haber uno y se llamará __construct
  - Será invocada automáticamente al hacer new
  - Es ideal para inicializar los datos del objeto antes de usarlo
- *void* __destruct (*void*)
  - será llamado tan pronto como no hayan otras referencias a un objeto determinado, o en cualquier otra circunstancia de finalización
  - será invocado aún si la ejecución del script es detenida usando exit()
  - Llamar a exit() en un destructor evitará que se ejecuten las rutinas restantes de finalización
  - No se pueden lanzar excepciones desde un constructor

Sobrecarga

- Ofrece los medios para "crear" dinámicamente propiedades y métodos.
- Se procesan por métodos mágicos
- Se invoca a los métodos de sobrecarga cuando se interactúa con propiedades o métodos que no se han declarado o que no son visibles en el ámbito activo
- Todos los métodos sobrecargados deben definirse como *public*.

Sobrecarga de propiedades

public *void* __set (*string* $name , *mixed* $value)
public *mixed* __get (*string* $name)
public *bool* __isset (*string* $name)
public *void* __unset (*string* $name)

- __set() se ejecuta al escribir datos sobre propiedades inaccesibles
- __get() se utiliza para consultar datos a partir de propiedades inaccesibles
- __isset() se lanza al llamar a isset() o a empty() sobre propiedades inaccesibles
- __unset() se invoca cuando se usa unset() sobre propiedades inaccesibles
- El parámetro $name es el nombre de la propiedad con la que se está interactuando

Ejemplo

```php
$obj = new PropertyTest;

$obj->a = 1;
echo $obj->a . "\n\n";

var_dump(isset($obj->a));
unset($obj->a);
var_dump(isset($obj->a));
```

```php
class PropertyTest
{
    private $data = array();

    public function __set($name, $value)
    {
        $this->data[$name] = $value;
    }

    public function __get($name)
    {
        if (array_key_exists($name, $this->data)) {
            return $this->data[$name];
        }
        return null;
    }

    public function __isset($name)
    {
        return isset($this->data[$name]);
    }

    public function __unset($name)
    {
        unset($this->data[$name]);
    }
}
```

public *string* __toString(*void*)

- Permite a una clase decidir cómo comportarse cuando se le trata como un string.
- Por ejemplo, lo que **echo** $obj; mostraría
- Debe devolver un string, si no se emitirá error
- No se puede lanzar una excepción desde dentro de __toString()

void __clone (*void*)

- Se llama al clonar un objeto una vez que la clonación ha finalizado
- Para clonar un objeto haremos:

```
$obj1 = clone $obj2;
```

- Si $obj2 tuviera como atributo alguna referencia, por ejemplo otro objeto, habría que clonar también ese otro objeto
- Ver ejemplos en:
  - http://php.net/manual/es/language.oop5.cloning.php#object.clone

Atributos estáticos

- Una clase puede tener atributos o métodos estáticos
- Se definen utilizando la palabra clave static

```
class Producto
{
    private static $num_productos = 0;
    public static function nuevoProducto()
    {
        self::$num_productos++; // desde dentro de la clase
    }
}
PRODUCTO::nuevoProducto(); // desde fuera de la clase
```

Comprobar el tipo de objeto

- El operador instanceof comprueba si un objeto es una instancia de una clase determinada

```
if ($p instanceof Producto) {
    ...
}
```

A partir de PHP5 se incluyen una serie de funciones útiles para el desarrollo de aplicaciones utilizando POO:

- http://php.net/manual/es/ref.classobj.php

Tipado de objetos en funciones

- Podemos indicar en las funciones y métodos de qué clase deben ser los parámetros

```
public function vendeProducto(Producto $p) {
    ...
}
```

Herencia

- Para definir una clase que herede de otra usamos la palabra extends

```
class TV extends Producto
{
    public $pulgadas;
    public $tecnologia;
}
```

Funciones relacionadas con la herencia

- Podemos obtener el nombre de la clase padre con la función get_parent_class
- También podemos comprobar si un objeto hereda de otro con la función is_subclass_of

Clases y métodos finales

- Al igual que ocurre en otros lenguajes podemos utilizar la palabra reservada final para evitar que se pueda heredar de una clase o sobrescribir un método

Clases y métodos abstractos

- Para definir una clase o método abstracto utilizamos la palabra reservada abstract
- No se podrán instanciar objetos de la clase, sólo podremos heredar de ella

Llamar a métodos de la clase base

- En PHP, si una clase heredada no tiene constructor propio, se llamará automáticamente al constructor de la clase base
- Si la clase heredada define su propio constructor, habrá que realizar la llamada explícitamente
- Utilizaremos la palabra reservada parent que hace referencia a la clase base de la clase actual
- La palabra reservada self hace referencia a la clase actual

Interfaces

- Es como una clase vacía que solamente contiene declaraciones de métodos
- Se definen utilizando la palabra interface
- Si queremos que una clase implemente un interface, utilizaremos la palabra reservada implements
- Esto obligará a que existan en la clase todos los métodos del interface
- Una clase puede implementar más de un interface
- Todos los métodos de un interface deben ser públicos
- Un interface puede incluir constantes, pero no atributos
- Un interface puede heredar de otro utilizando extends
- PHP tiene una serie de interfaces ya definidos, por ejemplo, el interface Countable

Traits (Rasgos)

- Implementados en PHP 5.4
- Ideales para la reutilización de código
- Permiten la reutilización de conjuntos de métodos sobre varias clases independientes y pertenecientes a clases jerárquicas distintas
- Es similar a una clase, pero con el único objetivo de agrupar funcionalidades muy específicas y de una manera coherente
- No se puede instanciar directamente un Trait
- Habilita la composición horizontal de comportamientos; es decir, permite combinar miembros de clases sin tener que usar herencia.

Sintaxis

- Los crearemos utilizando la palabra reservada trait
- Para que una clase pueda utilizar los métodos implementados en un trait tendremos que añadir una línea use indicándolo

Ejemplo

```
trait Log {
    protected function log($msg) {
        echo "{$msg}\n";
    }
}

class Table {
    use Log;
    public function save() {
        $this->log('save start');
    }
}
```

- El **trait** no impone ningún comportamiento en la clase, simplemente es como si copiaras los métodos del trait y los pegaras en la clase

Cuándo usar traits

- Supongamos el siguiente ejemplo:

```
// Lector DB
class DbReader extends Mysqli{}
// Lector de archivos
class FileReader extends SplFileObject{}
```

- Si queremos aplicar una misma funcionalidad a las dos clases tenemos un problema porque ambas están ya heredando
- La solución sería usar un trait

Usar múltiples traits

```php
trait Saludar {
    function decirHola(){
        return "hola";
    }
}
trait Despedir {
    function decirAdios(){
        return "adios";
    }
}
class Comunicacion {
    use Saludar, Despedir;
}
$comunicacion = new Comunicacion;
echo $comunicacion->decirHola() . ", que tal. " . $comunicacion->decirAdios();
```

Usar traits dentro de otros traits

```
trait SaludoYDespedida{
    use Saludar, Despedir;
}
class Comunicacion {
    use SaludoYDespedida;
}
$comunicacion = new Comunicacion;
echo $comunicacion->decirHola() . ", que tal. " . $comunicacion->decirAdios();
```

Precedencia entre traits y clases

- Puede haber métodos con el mismo nombre en diferentes traits o en la misma clase
- Tiene que haber un orden:
  - Métodos de un trait sobreescriben métodos heredados de una clase padre
  - Métodos definidos en la clase actual sobreescriben a los métodos de un trait

Conflictos entre métodos de traits

- Cuando se usan múltiples traits es posible que haya diferentes traits que usen los mismos nombres de métodos
- PHP devolverá un error fatal

```
trait Juego {
    function play(){
        echo "Jugando a un juego";
    }
}
trait Musica {
    function play(){
        echo "Escuchando música";
    }
}
class Reproductor {
    use Juego, Musica;
}
$reproductor = new Reproductor;
$reproductor->play();
// Devuelve Fatal error: Trait method play has not been applied,
// because there are collisions with other trait methods on Reproductor
```

Solución al conflicto

- Esto no se resuelve de forma automática
- Hay que elegir el método que queremos usar dentro de la clase mediante la palabra ***insteadof***

```
class Reproductor {
    use Juego, Musica {
        Musica::play insteadof Juego;
    }
}
```

Otra solución

```
class Reproductor {
    use Juego, Musica {
        Juego::play as playDeJuego;
        Musica::play insteadof Juego;
    }
}
$reproductor = new Reproductor;
$reproductor->play(); // Devuelve: Escuchando música
$reproductor->playDeJuego(); // Devuelve: Jugando a un juego
```

Acceder a propiedades y métodos de la clase

- Los traits se inyectan en la clase de forma que es como si sus métodos formaran parte de ella, por lo que cualquier método o propiedad private o protected podrá ser accedido desde un trait.

```
trait Mensaje {
    public function alerta(){
        echo $this->mensaje;
    }
}
class Mensajero {
    use Mensaje;
    private $mensaje = "Esto es un mensaje";
}
$mensajero = new Mensajero();
$mensajero->alerta(); // Devuelve: Esto es un mensaje
```

Métodos abstractos en traits

- Podemos tener métodos abstractos dentro de traits para forzar a las clases a implementarlos

```
trait Mensaje {
    private $mensaje;

    public function alerta(){
        $this->definir();
        echo $this->mensaje;
    }
    abstract function definir();
}
class Mensajero {
    use Mensaje;
    function definir(){
        $this->mensaje = "Esto es un mensaje";
    }
}
$mensajero = new Mensajero();
$mensajero->alerta(); // Devuelve: Esto es un mensaje
```