Índice

5. Desenvolvemento de aplicacións web con código embebido

5.1 Fundamentos da POO en PHP

Herdanza

A herdanza é un concepto da POO co que **unha clase pode herdar todas as propiedades e métodos doutra e ademáis engadir os seús propios**. Por exemplo, a clase Alumno e a clase Profesor poderian herdar o contido da clase Persoa e, cada unha delas, ademáis podería ter características propias. Para que unha clase descenda de outra utilízase a palabra clave `extends`.

```
class Persoa {
    protected $nome;
    protected $apelidos;
    …
}
class Alumno extends Persoa {
    private $numExpediente;
}
```

Así Alumno automáticamente obterá todas as propiedades públicas e protexidas de Persoa así como os métodos de Persoa, co cal o alumno terá as tres propiedades: nome, apelidos e numExpediente.

A herdanza permite reutilizar eficientemente o código da clase base. No exemplo podemos dicir que *Alumno é un tipo de Persoa*, e estaremos reutilizando o mesmo código en dúas clases distintas facendo o código máis lixeiro.

As **novas clases que herdan** coñécense tamén co nome de ***subclases***. **A clase da que herdan** chámase ***clase base*** **ou** ***superclase***. Os novos obxectos que se instancien a partir da subclase son tamén obxectos da superclase, o que podemos comprobar como sigue:

```
$a = new Alumno();
if ($a instanceof Persoa) {
    // como Alumno é un tipo de Persoa cumpliría a condición
    …
}
```

Sobrescribiendo métodos

Cando se usa a herdanza, **pode suceder que un dos métodos da clase base** non nos sexa útil, ou que **necesite que cambie o seu comportamento**. Nese caso o que temos é que sobreescribir o método.

Para **sobreescribir** un método basta con declaralo de novo na subclase, o que fai que cando se chame ao método PHP use o da subclase, pero debemos ter en conta que se sobreescribimos un método, este debe ter a **mesma firma que a do pai** (debe recibir os mesmos parámetros). En caso contrario xerarase un error de nivel E_STRICT. A única **excepción** é o construtor, que pode redefinirse cos parámetros que se desexe.

```
class Persoa {
    protected $nome;
    protected $apelidos;
    public function __contruct($nom)
    {
        $this->nome=$nom;
    }
    …
}
class Alumno extends Persoa {
    private $numExpediente;
    public function __contruct($exp, $nom)
    {
        $this->numExpediente=$exp;
        $this->nome=$nom;
    }
}
$a=new Alumno(1089,'Sabela');
```

Neste exemplo executarase o construtor da subclase. No caso de que queiramos que se chame ao da clase deberemos solicitalo explícitamente. Así o construtor de Alumno podería quedar como segue:

```
public function __contruct($exp, $nom)
    {
        $this->numExpediente=$exp;
        parent::__construct($nom);
    }
```

Onde a palabra clave **`parent` fai referencia á clase da cal se está herdando**. A instrucción **`parent::metodo()`** pódese facer tanto sobre obxectos como sobre métodos estáticos de clases.

Herdanza e visibilidade

Agora é cando podemos falar do nivel de protección `protected`. Este nivel é similar ao privado no senso de que bloquea o acceso fora da clase, pero permite que as clases fillas poidan acceder e manipular o atributo ou método en cuestión.

Debemos ter en conta que os niveis nunca poden ser máis restrictivos na subclase que na clase base, esto quere dicir que si un método na clase base é `public` na subclase deberá seguir sendo `public`, pero se na clase base á `private` ou `protected`, na subclase poderá ser `public`.

Por exemplo, se temos unha clase base cun método protexido:

```
class Clase
{
    //Método Protexido
    protected function metodo(){
        echo "Método protexido da clase base";
    }
}
```

Poderemos chamar a este método dende a subclase:

```
class ClaseFilla extends Clase
{
     public function chamarMetodo(){
         $this->metodo();
     }

}
```

E se creamos un obxecto pertencente a subclase, para chamar ao método protexida da clase pai deberemos facelo a través do método público declarado nesta,

```
$c = new ClaseFilla();
$c->chamarMetodo();
```

xa que se intentamos acceder directamente ao método protexido da clase base non funcionaría.

```
 $c->miMetodo(); //ERRO
```

Isto é debido a que os métodos protexidos garantizan que certa funcionalidade unicamente se poda aplicar baixo un estado seguro que coñece o obxecto.

Por ouro lado, os métodos e propiedade privados diferencianse dos protexidos só no caso de que exista herdanza. Os métodos e propiedades privados son visibles unicamente á clase base (ou obxectos pertencentes a esta clase) Nos fillos actúan coma se non existiran.

Impedir a herdanza con final

Nalgúns casos poderiamos querer evitar que se podia herdar dunha clase. Isto faise engadindo na declaración da clase o operador `final`. Por exemplo:

```
final class Clase { ... }
```

bloquea a herdanza para Clase.

Se o que queremos é simplemente evitar que se sobrescriba un método, para así asegurarnos de que sempre se executara tal e como se ten definido na clase base, basta con engadir o operador final na súa declaración:

```
public final function metodo(){ ... }
```

Enlace estático en tempo de execución

Imos ver o que acontece cando herdamos dunha clase que contén un método estático. Supoñamos que temos o seguinte método estático na clase pai para contar o número de persoas de cada tipo que se van engadindo:

```
class Persoa {
    public static function novaPersoa() {
        self::$numPersoas++;
    }
    …
}
class Alumno extends Persoa {
    private static $numPersoas = 0;
```

```
    …
}
class Profesor extends Persoa {
    private static $numPersoas = 0;
    …
}
```

Así, cando facemos o seguinte:

```
Alumno::novaPersoa();
```

o que queremos é que engada 1 á variable `$numPersoas` da clase Alumnos, e cando fagamos

```
Profesor::novaPersoa();
```

queremos que engada 1 á variable `$numPersoas` da clase Profesor.

Non obstante, o código anterior non funciona como esperamos pois, ao compilar encontra no método `novaPersoa` unha referencia a `self::$numPersoas`, e intenta enlazar cunha variable da clase a que pertence (*Persoa*) que non existe.

Para solventalo temos que indicar a PHP que enlace coa variable en tempo de execución, non en tempo de compilación, e empregue as propiedades da clase que fai a chamada ao método. Isto faise empregando a palabra `static` no canto de `self` co operador de resolución de ámbito `::`.

Tamén teremos que definir as propiedades `numPersoas` como `protected` para darlles acceso dende o método herdado. Isto é:

```
class Persoa {
    public static function novaPersoa () {
        static::$numPersoas ++;
    }
    …
}
class TV extends Alumno {
    protected static $numPersoas = 0;
    …
}
class Ordenador extends Profesor {
    protected static $numPersoas = 0;
    …
}
```

O mesmo acontece cando temos que acceder ao nome da clase. Se empregamos `__CLASS__` dende o método `novaPersoa` devolveranos `Persoa`. Para obter en tempo de execución o nome da clase que fai a chamada teremos que empregar a función `get_called_class()`.

Outra forma de arranxar o problema da herdanza de métodos estáticos é emprengando *trazos*.

Funcións relacionadas coa herdanza

FUNCIÓN		EXEMPLO
get_parent_class	Devolve o nome da clase pai do obxecto ou a	class Alumno extends Persoa {

	clase que se indica.	function sonFillo()
		{
		echo "Son fillo de ", get_parent_class($this) ;
		}
		}

is_subclass_of	Verifica se o obxecto ten esta clase como un dos seus pais.	$a= new Alumno();
		
		if (is_subclass_of($a, 'Persoa')) {
		echo " \$a é unha subclase de Persoa";
		} else {
		echo "\$a non é unha subclase de Persoa";
		}

Clases abstractas

A partires de PHP5 introducíronse na linguaxe as clases e os métodos abstractos. As clases abstractas declaranse con palabra clave `abstract` e son similares ás clases normais excepto en dous aspectos:

- Non se poden crear obxectos a partir delas, isto é: unha clase abtracta non pode ser instanciada.
- Unha clase abstracta pode incorporar métodos abstractos, que son aqueles dos que so existe a súa declaración, deixando a implementación para as clases fillas ou derivadas.

O normal é que declaremos unha clase como abstracta cando sabemos que non vai ser instanciada e que a súa función é servir de clase base a outra clase mediante a herdanza. Calquera clase que defina un método abstracto debe ser declarada como abstracta (isto é lóxico pois non ten implementado algún método e por tanto non se poderán instanciar obxectos a partires dela).

Por exemplo, se non imos traballar con obxectos da clase Persoa, poderíamos facela abstracta e empregala como base para definir os métodos comúns ás clases que deriven dela:

```
abstract class Persoa {
    protected $nome;
    protected $apelidos;
    …
    abstract public function imprimir();
}
```

Polimorfismo

O polimorfismo é un dos pilares basicos da programacion orientada a obxectos. Consiste en que un mesmo identificador ou función poida comportarse de xeito diferente en función do contexto no que sexa executado.

Por exemplo, podemos ter unha clase abstracta que serve de base a tres ou catro clases fillas, de xeito que as funcionalidades compartidas estarán na clase base e unicamente modificas ou engades pequeños matices nas subclases.

No seguinte exemplo temos unha clase Coche e dúas clases derivadas desta que consumiran diferente cantidade de gasolina cando arrancan.

```
class Coche{
```

```
    protected $gasolina = 50;
    public function setGasolina($gas){
        $this->gasolina=$gas;
    }
    public function getGasolina (){
        return $this->gasolina;
    }
}
class Ferrari extends Coche{
    public function arrancar(){
        $this->gasolina -= 5;
    }
}
class Seat extends Coche{
    public function arrancar(){
        $this->gasolina -= 2;
    }
}
```

Interfaces

Os interfaces permiten establecer as características que debe cumplir unha clase. Permiten definir qué métodos deben ser declarados nunha clase de xeito similar ao que fan as clases abstractas. A principal diferenza é que **non se pode herdar código dun interface**. Un interface **impleméntase**, o que permite herdar dunha clase base e implementar un ou máis interface á vez.

O interface é o recurso ideal para a implementación do *polimorfismo*, xa que os interfaces unicamente declaran funcións ou métodos que deben ser codificadas nas clases que os implementan.

O xeito de declarar un interface é similar á declaración dunha clase, substituindo a palabra `class` por **`interface`**. Un interface non permite definir atributos a implementar nas clases, só métodos, e estos deberán ser sempre públicos.

Por exemplo, pensemos nun coche e un cartel luminoso. En principio non teñen nada en común, pero ambolos dous han de ser prendidos e apagados nun momento dado. Polo tanto, poderiamos definir un interface que obligue as clase que o implementen a ter estes dous métodos.

```
interface Accions {
     public function arrancar();
     public function apagar();
 }
```

Para traballar cunha interfaz úsase a palabra reservada **`implements`**. Todos os métodos que están definidos na interfaz deberán estar nas clases que a implementan, senón isto producirá un *fatal error*.

```
class Seat extends Coche implements Accions{
    public function arrancar(){
        ...
    public function avanzar($velocidade){
        ...
    }
```

Cada clase pode implementar máis dunha interfaz e para iso basta con separalas por comas. Unha clase non pode implementar interfaces que teñan funcións co mesmo nome, xa que isto produciría unha colisión.

Un interface é como un contrato que a clase debe cumprir. Ao implementar todos os métodos declarados no interface asegúrase a interoperabilidade entre clases. Se sabes que unha clase implementar un interface determinado, sabes que nome teñen os seus métodos, que parámetros lles debes pasar e, probablemente, poderás descubrir doadamente con que obxectivo foron escritos.

Por exemplo, na libraría de PHP está definido o interface `Countable`. Se queremos facer un contador personalizado poderemos implementar este interface.

```
Countable {
    abstract public int count ( void );
}
```

Funcións relacionadas cos interface

FUNCIÓN		EXEMPLO
get_declared_interfaces	Devolve un array cos nomes dos interfaces declarados	print_r(get_declared_interfaces());

interface_exists	Comproba se un interface está definido	if (interface_exists('Accions')) {
		class Clase implements Accions
		{ ... }
		}

Excepcións

A partir da versión 5 introduciuse en PHP un modelo de tratamento de excepcións similar ao existente noutras linguaxes de programación. **Unha excepción pode ser lanzada e atrapada**, e ademáis:

- Meteremos dentro de un **bloque `try`** o código susceptible de producir algún erro para facilitar a captura de posibles excepcións.
- cando se produce algún erro, **lánzase unha excepción utilizando a instrución `throw`**.
- Cada bloque `try` debe poseer como **mínimo un bloque `catch`**, que será o encargado de procesar o erro en caso de que unha excepción sexa lanzada.
- se unha vez rematado o bloque `try` non se lanzou ningunha excepción, continúase coa execución na liña seguinte ao bloque ou bloques `catch`.
- a partires de PHP 5.5 pódese utilizar un **bloque `finally`** despois dos bloques `catch`. O código de dentro do bloque `finally` sempre se executará despois dos bloques `try` e `catch`, independentemente de que se lanzase unha excepción ou non, e antes de que o fluxo normal de execución continúe.

PHP ofrece unha **clase base `Exception`** para utilizar como manexador de excepcións. Para lanzar unha excepción non é necesario indicar ningún parámetro, aínda que de forma opcional se pode pasar unha mensaxe de erro e un código de erro. Entre os métodos que se poden usar cos obxectos da clase Exception están:

- `getMessage`._ Devolve a mensaxe, en caso de que se puxese algún.
- `getCode`._ Devolve o código de erro se existe.

```
function division($dividendo, $divisor) {
  try {
    if ($divisor == 0) {
        throw new Exception('Non se pode dividir por 0');
    }
    else return $dividendo/$divisor;
  } catch (Exception $e) {
      echo 'Excepción capturada: ',  $e->getMessage(), "\n";
  }
}
```

As funcións internas de PHP e moitas extensións como MySQLi usan o manexador de erros clásico. Só as extensións máis modernas orientadas a obxectos, como é o caso de PDO, utilizan este modelo de excepcións. Neste caso, o máis común é que a extensión defina os seus propios manexadores de erros herdando da clase Exception.