5.- Programación Orientada a Obxectos en PHP

Índice

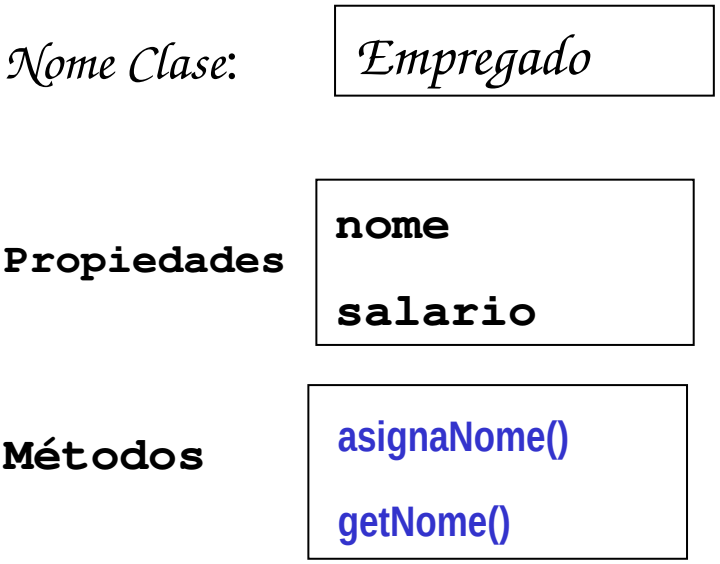
Na **Programación Orientado ao Obxectos** (POO), trabállase con conceptos como **clase**, **obxecto**, **herdanza**, **encapsulamento** e **polimorfismo**. Nesta unidade veremos unha introdución ás clases con PHP, e a estes conceptos de POO.

Clases

Unha **clase** é unha descrición xenérica ou ***plantilla*** dun determinado tipo de obxectos: por exemplo un clase ***Empregado***, indicará as características que terán TODOS os obxectos Empregado.
Cada instancia da clase, ou **obxecto** (p.ex, cada empregado) terá unhas **propiedades** (tamén se lles pode chamar atributos) e uns **métodos** (**funcións** asociadas a ese obxecto). Estas propiedades e métodos deben estar definidos cando definimos a clase:
En PHP decláranse as clases coa sentencia **class**:

```
class Empregado
{
   //PROPIEDADES
   public $nome;
   public $salario;

   //MÉTODOS
   public function asignaNome($nomeIndicado)
   {
      $this->nome=$nomeIndicado;
   }
   public function getNome()
   {
      return $this->nome;
   }
}
//FIN DA CLASE
```



Para crear un obxecto desa clase, empregamos a instrución **new**. Esta sentencia crea o novo obxecto e devolve un identificador, que gardamos nunha variable. Unha vez creado un obxecto, podemos acceder ás súas propiedades e métodos usando o operador frecha: **->**. Exemplo: Creamos 2 obxectos *Empregado*:

```
$miguel= new Empregado( );
$ana= new  Empregado( );
$miguel->asignaNome("Miguel");
$ana->nome="Ana";
echo "os novos empregados son". $miguel->nome." e ".$ana->getNome();
```

Recomendacións na creación de clases

É recomendable que cada clase figure no seu propio ficheiro que debe ter como nome o nome da clase que contén. Por ejemplo, se temos unha clase **Empregado**, esta debería crearse dentro dun arquito de nome **Empregado**.***php***. Para poder facer uso desa clase, é dicir, para crear instancias dela é imprescincible que a súa definición se inclúa no arquivo onde se vai crear esa instancia.
Aínda que os nomes das clases non son sensibles a maiúsculas e minúsculas, é recomendable que as clases comecen por letra *maiúscula*, para, desta forma, distinguilos dos obxectos e outras variables.

Construtores

Pode existir unha función ***construtor***, esta será chamada cada vez que se crea un obxecto. Ten que chamarse coa sentencia **__construct** (construct con 2 barras baixas antes)

Por exemplo:

```
class Empregado {
   //PROPIEDADES
   public $nome;
   public $salario;

   //MÉTODOS
   public function __construct() {
      $this->nome=”sen nome”;
   }
   public function asignaNome($nomeIndicado) {
      $this->nome=$nomeIndicado;
   }
   public  function getNome() {
      return $this->nome;
   }
}
//FIN DA CLASE
```

Cada vez que se cree un empregado a súa propiedade nome tomará o valor “sen nome”, ata que sexa cambiada.

O constructor tamén podería recibir valores:

```
class Empregado {
   ...
   //MÉTODOS
   public function __construct($nome, $salario)
   {
      $this->nome=$nome;
      $this->salario=$salario;
   }
...
}
//FIN DA CLASE
```

Como PHP non permite a sobrecarga de operadores de xeito nativo, teremos un único construtor por clase.

Destrutores

PHP tamén ten o concepto de función “destrutora”, que é chamada **XUSTO ANTES** de eliminar o obxecto. O servidor libera recursos ao rematar o script, e por tanto elimina todos os obxectos da nosa páxina, pero pode ser interesante **ANTES** realizar algunha acción ou rexistrar algún dato.

Podemos eliminar un obxecto coa función unset(obxecto), por exemplo, **unset($ana);**

A sintaxe é coa palabra reservada destruct, con 2 barras baixas diante: **__destruct( )**

Exemplo:

```
class Empregado {
   ...
```

```
   public function __destruct()
   {
      echo "<br>obxecto de nome ".$this->nome." foi destruído<br>";
   }
}
//FIN DA CLASE
```

Visibilidade de propiedades e métodos. Encapsulación

- ✗ **public**: son accesibles desde calquera lugar do código. Se non se especifica na definición dos métodos, estes serán públicos por defecto.
- ✗ **private**: só accesibles desde a clase que os define (non clases fillas), coas funcións definidas na clase.
- ✗ **protected**: só accesibles desde a clase que define o elemento (e as clases **derivadas** que herden dela)

Por exemplo ser definimos nome como privado, non podemos acceder a el directamente:

```
class Empregado {
   //PROPIEDADES
   private $nome;
   public $salario;
   ....

}

$miguel= new Empregado( );
$ana= new  Empregado( );
$miguel->asignaNome("Miguel");
$ana->nome="Ana";

echo "os novos empregados son". $miguel->nome." e ".$ana->getNome();
```

Neste punto (miguel->nome) o servidor PHP dará un erro, pois nome é agora unha propiedade privada.

***Fatal error**: Uncaught Error: Cannot access private property Empregado::$nome...*

A variable $this

A variable **$this** proporciona unha referencia ao propio obxecto, e coa notación -> permite acceder ás propiedades e métodos do propio obxecto ao que fai referencia.

Está só dispoñible polos métodos **DENTRO** do contexto do obxecto, na definición da clase.

```
class Empregado
 {
   ...
   public function asignaNome($nomeIndicado)
   {
   $this->nome=$nomeIndicado;
   }
 ...
```

Definindo get e set

Os métodos que permite acceder e modificar os valores das propiedades son normalmente coñecidos como **getters** e **setters**. Se queremos *implementar* algunha **lóxica** na entrada dos datos teremos que facelo nestes métodos. Deste modo, o obxecto ten control sobre os seus datos, non accedendo directamente aos seus datos deste o exterior da clase (*encapsulación*):

```
class Empregado  {
   ...
   private $idade;
   ...
   public function getIdade()    {
   return $this->idade;
   }

   public function setIdade($anos) {
      if ($anos>0 && $anos<120) {
         $this->anos = $anos;
      }
   }
 ...
```

Funcións de clases/obxectos

Existen en PHP algunhas funcións de clases/obxectos, que nos dan información sobre as clases e obxectos:

FUNCIÓN		**EXEMPLO**

class_exists( )	Devolve true se a clase está definida e false en caso contrario.	if (class_exists('Persoa') {
		$p = new Persoa();
		...
		}

get_class( )	Devolve o nome da clase do obxecto.	echo "A clase é: " . get_class($p);
get_class_methods ( )	Devolve un array cos nomes dos métodos dunha clase que son accesibles dende ónde se fixo a llamada.	print_r(get_class_methods('Persoa'));
get_class_vars ( )	Devolve un array cos nomes dos atributos dunha clase que son accesibles dende onde se fixo a chamada.	print_r(get_class_vars('Persoa'));
get_declared_classes ( )	Devolve un array cos nomes das clases definidas.	print_r(get_declared_classes());
get_declared_interfaces ( )	Devolve un array cos nomes das interfaces definidas.	print_r(get_declared_interfaces());
class_alias( )	Crea un alias para unha clase.	class_alias('Persoa', 'Habitante'); $p = new Habitante();
get_object_vars( )	Devolve un array coas propiedades non estáticas do obxecto especificado. Se unha propiedade non ten asignado un valor devolverase cun valor NULL.	print_r(get_object_vars($p));

method_exists ( )	Devolve true se existe o método no obxecto ou clase que se indica, e false no caso contrario, independentemente de se é accesible ou non.	if (method_exists('Persoa', 'impimir') {
		...
		}

property_exists( )	Devolve true se existe o atributo no obxecto ou clase que se indica, e false no caso contrario, independentemente de se é accesible ou non.	if (property_exists('Persoa', 'email') {
		...
		}

Nas funcións e métodos definidas por nós, é opcional indicar de que clase deben ser os obxectos que se pasen como parámetros e o tipo devolto:

```
public function prezoProduto(Produto $p) :float {
     . . .
     return $prezo;
}
```

Autocarga de clases

Se temos creada cada clase nun arquivo co seu nome, para incluílas no noso código teremos que empregar **require_once** ou **include_once** para pode dispoñer delas:

```
require_once(“conexion.php”);
require_once(“persoa.php”);
...
```

Cando temos un proxecto con moitas clases é máis práctico e seguro obrigar a cargar só as clases que se precisan no noso código. Podemos empregar para isto a función **`spl_autoload_register()`.**

O seguinte exemplo cargará as clases **MiClase1** e **MiClase2** se existen os ficheiros **MiClase1.php** e **MiClase2.php** (ver en https://www.php.net/manual/es/language.oop5.autoload.php)**:**

```
<?php
spl_autoload_register(function ($nome_clase) {
    include $nome_clase . '.php';
});

$obj  = new AClase1();
$obj2 = new AClase2();
?>
```

Teremos que definir a función ***spl_autoload_register*** no inicio do noso programa.

Utilización de obxectos

En PHP existen algunhas particularidades no uso dos obxectos.

Referencias a obxectos

En PHP, o nome da variable e o contido da mesma son cousas totalmente diferentes, que se gardan na chamada táboa de símbolos. Cando creamos unha referencia o que creamos é un alias, que permite que 2 variables podan escribir sobre o mesmo valor. Isto pasa por exemplo:

```
$miguel = new Empregado( );
$miguel->asignaNome(“miguel”);
$novo = $miguel;   //OLLO! Un alias de $miguel
$novo->setNome(“Ana”);  //ESCRIBIMOS TAMÉN O NOME DE $miguel
```

Clonar obxectos

Para facer unha **copia** dun obxecto, temos que empregar o método clone :

```
$miguel = new Empregado( );
$miguel->asignaNome(“miguel”);
$novo = clone $miguel;   //Xa NON un alias de $miguel
```

Se cando facemos a copia queremos facer algún cambio no obxecto, podemos definir o método máxico ***__clone*** na nosa clase***.*** Este método dispararase no momento de clonar un obxecto mediante a instrución **clone**.

```
class Empregado
{
   private $nome;
   $private $salario;
   //MÉTODOS
   public function __construct($nom, $salar)
   {
      $this->nome=$nom;
      $this->salario=$salar;
   }

   public function __clone( )
   {
      $this->nome = “Anónimo!”;
   }
...
}

$miguel = new Empregado( ‘Miguel’,’20000);
$ana = clone $miguel; //AGORA $ana  TERÁ COMO NOME “Anónimo”
```

Comparación de obxectos

Para comparar obxectos temos 2 operadores similares ao uso xa visto:

- ✗ O operador de comparación simple ('=='): dará como resultado verdadeiro se os obxectos que se comparan son instancias da mesma clase e os seus atributos teñen os mesmos valores.
- ✗ O operador de identidade ('==='): dará como resultado verdadeiro se as variables comparadas son referencias á mesma instancia.

Constantes de clases

Unha clase pode ter ***constantes***, que son valores fixos establecidos na definición da clase, que non se poden modificar en tempo de execución.

Para declarar unha constante no interior dunha clase usamos a palabra reservada *const* antes do nome da constante que **NON** debe levar o caracter $. Este nome recoméndase poñelo en maiúsculas para diferenciar facilmente unha constante dunha variable.

O valor da constante está asociado á clase, e non se fai unha copia para cada obxecto que se cree. Para acceder ao valor da constante podemos usar o obxecto ou o nome da clase seguido de dous puntos dúas veces (::). Este operador (::) coñecese como operador de resolución de ámbito, e é o que se usa para acceder aos elementos dunha clase. Así, por exemplo:

```
class Calendario
{
   const NUM_MESES = 12;
}
```

creamos un obxecto pertencente a esta clase

```
$cal = new  Calendario();
```

As dúas seguintes sentenzas serían equivalentes:

```
echo Calendario::NUM_MESES;

echo $cal::NUM_MESES;
```

Constantes predefinidas

En PHP existen algunhas constantes predefinidas, entre as que se atopan:

- ✗ `__CLASS__`: devolve o nome da clase onde foi declarada
- ✗ `__METHOD__`: devolve o nome do método onde foi declarada
- ✗ `__FILE__`: Ruta completa e nome do arquivo. Usado dentro dun `INCLUDE` devolverá o nome do ficheiro do `INCLUDE`.

- ✗ `__DIR__`: Directorio do ficheiro. Usado dentro dun `INCLUDE`, devolverá o directorio do arquivo incluído.
- ✗ `__LINE__`: Liña actual do ficheiro.

Operador de resolución de ámbito (::)

Permite chamar a variables de clase, ou a un método ou propiedade definido nunha clase como ***static***, sen necesidade de ter creado previamente un obxecto. (*ver o seguinte apartado*)

Propiedades e métodos estáticos

Se unha propiedade ou método está definida/o como ***static*** (estática) é unha propiedade **de clase**, será accesible desde fóra do contexto do obxecto, coa sintaxe **NomeClase::propiedade**, ou N**omeClase::nomeMetodo()**.

Para acceder a esas propiedades ou métodos empregaremos o operador de resolución de ámbito **(::)**. Desde a propia clase a propia clase é accesible co palabra reservada **self**

Exemplo:

```
class Empregado {
   //PROPIEDADES
   public $nome;
   public static $numEmpregados=0;
   //MÉTODOS
...
public function __construct()
 {
      $this->nome=”sen nome”;
      self::$numEmpregados++;  //CADA EMPREGADO CREADO INCREMENTA A VARIABLE
 }
}


$miguel= new Empregado();
$ana= new  Empregado();
$miguel->asignaNome("Miguel");
$ana->asignaNome("Ana");
echo "os novos empregados son ". $miguel->nome." e ".$ana->getNome();
echo "<br>Levamos ".Empregado::$numEmpregados." empregados";
```

Mantemento do estado. Serialización de obxectos

As variables gardadas en PHP almacenan a súa información en memoria de diferente forma en función do seu tipo. Cando traballamos con obxectos, non temos un único tipo. Cada obxecto ten uns atributos en función da súa clase. Se precisamos gardalos nunha base de datos ou nun ficheiro debemos convertelos a texto. Isto chámase **serialización**
Para serializar un obxecto en PHP emprégase a función **serialize**

```
$miguel = new Persoa();
$miguelSerializado = serialize ($miguel);
```

Agora esta cadea de texto podemos gardala nunha base de datos, ou nun ficheiro. Para recuperar o obxecto inicial teremos que empregar a función **unserialize:**

```
   $miguel = unserialize($miguelSerializado);
```

Deste xeito podemos gardar obxectos como texto, sen perder ningunha propiedade. Só se perderían os valores das propiedades estáticas de clases.

Lembrar que os obxectos gardados nunha sesión NON precisan ser serializados, a serialización é feita automaticamente no array SESSION. Así, podemos gardar obxectos na sesión.
Podes ver o exemplo seguinte que tes en php.net na documentación de serialización:
https://www.php.net/manual/es/language.oop5.serialization.php:

```
<?php
// classa.inc:

  class A {
      public $one = 1;

      public function show_one() {
          echo $this->one;
      }
  }

// page1.php:

  include("classa.inc");

  $a = new A;
  $s = serialize($a);
  // ALMACENAMOS $s NALGÚN LUGAR NO QUE page2 PODE ATOPALO
  file_put_contents('store', $s);

// page2.php:

  // PRECISAMOS O inclue PARA QUE unserialize FUNCIONE CORRECTAMENTE
  include("classa.inc");
```

```
  $s = file_get_contents('store');
  $a = unserialize($s);

  // AGORA EMPREGAMOS O MÉTODO show_one() DO OBXECTO $a
  $a->show_one();
?>
```

Herdanza

Unha das características máis importantes da Programación Orientada a Obxectos é a posibilidade de crear unha clase empregando unha xa existente, aproveitando a súa funcionalidade. A clase filla herdará tanto as propiedades como os métodos da clase nai. Para indicar que unha clase é filla doutra empregamos a palabra clave ***extends***:

```
class Empregado {
...
}

class Operario extends Empregado
{
      private $turno;
      public function getTurno() {
         return $this->turno;
      }
...
}
```

Se declaramos un **construtor** para a clase filla hai que chamar explicitamente ao construtor da clase nai desde o construtor da clase filla, coa sentencia ***parent::__construct()***:

```
class Operario extends Empregado
{
      ...
      public function __construct(){
         parent::__construct();  //EXECÚTASE PRIMEIRO O CONSTRUTOR DE EMPREGADO
         $this->$turno="non especificado";
      }
      ...
}
```

O mesmo pasaría co destrutor: hai que chamar explicitamente ao destrutor da clase nai, desde o destrutor da clase filla.

Así:

```
class Empregado
{
   //PROPIEDADES
   public $nome;
   public $salario;
   public static $numEmpregados=0;

   //MÉTODOS
   public function __construct()
```

```
   {
      $this->nome="sen nome";
      self::$numEmpregados++;  //CADA EMPREGADO CREADO INCREMENTA A VARIABLE
   }

   public function asignaNome($nomeIndicado)
   {
      $this->nome=$nomeIndicado;
   }
   public function getNome()
   {
      return $this->nome;
   }
   public function __destruct()
   {
   echo "<br>obxecto de nome ".$this->nome." foi destruído<br>";
   }
}


// CLASE OPERARIO FILLA DE EMPREGADO:

class Operario extends Empregado
{
      private $turno;
      public function __construct()
      {
         parent::__construct();  //EXECÚTASE PRIMEIRO O CONSTRUTOR DE EMPREGADO
         $this->turno="non especificado";
      }
      public function __destruct()
      {
         parent::__destruct();  //EXECÚTASE O CONSTRUTOR DE EMPREGADO
      }
      public function getTurno()
      {
         return $this->turno;
      }
      public function setTurno($turnoEnviado){
         if ($turnoEnviado == “diurno” || $turnoEnviado == “nocturno”)
             $this->turno=$turnoEnviado;
      }
}
$miguel= new Empregado();
$ana= new  Empregado();
$miguel->asignaNome("Miguel");
$ana->asignaNome("Ana");
echo "os novos empregados son ". $miguel->nome." e ".$ana->getNome()."<br>";
$pedro=new operario();
$pedro->asignaNome("Pedro");
echo "O operario ", $pedro->getNome()," ten o turno ".$pedro->getTurno()."<br>";
//USAMOS O MÉTODO getNome() DA CLASE NAI

$pedro->setTurno("diurno");
echo "O operario ", $pedro->getNome()," ten o turno ".$pedro->getTurno()."<br>";

echo "<br>Levamos ".Empregado::$numEmpregados." empregados"."<br>";
```

Herdanza e visibilidade: protected

O nivel de visibilidade ***protected*** é similar a ***private*** pois bloquea o acceso desde fóra da clase, pero permite que as clases derivadas podan acceder e manipular a propiedade ou método.

O nivel de visibilidade non poderá ser máis restritivo na subclase que na clase nai: se é ***public*** na case nai deberá seguir sendo ***public***, pero se é ***private*** ou ***protected*** poderá ser ***public*** na clase derivada.

```
Class Persoa
{
...
//Método protexido
   protected function metodo( )
   {
   echo “método protexido na clase base”;
   }
}

class Traballador extends Persoa
{
   public function chamaMetodo()
   {
   $this->metodo();
   }
 }
```

Se creamos un obxecto da subclase debemos chamar ao método protexido a través do método público declarado na subclase (se non o sobreescribe)

```
$c = new Traballador();
$c->chamaMetodo();
```

Pois se accedemos directamente ao método protexido da clase base dará un erro:

```
$c->metodo( )   /ERRO
```

Impedir herdanza con final

Se queremos que unha clase non poda ter clases derivadas definirémola co operador final:

```
final class Operario
{
...
}
```

Se non queremos que unha clase derivada sobreescriba un método, podemos tamén facer isto:

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