5.- Programación Orientada a Obxectos en PHP

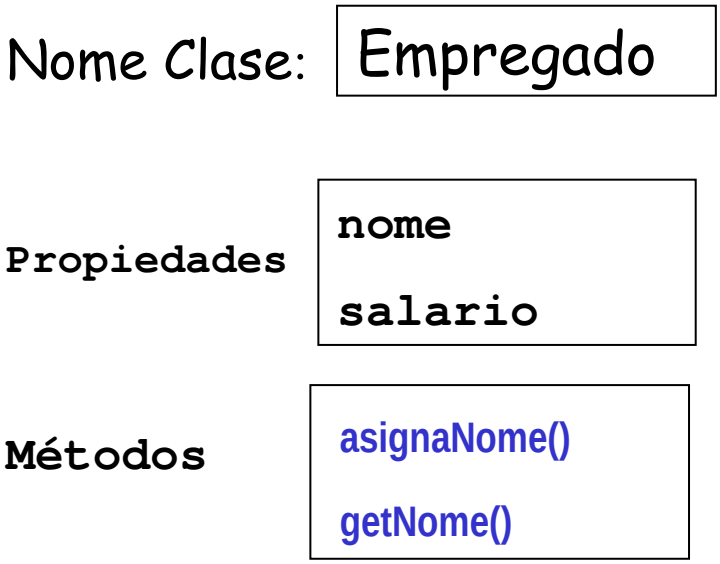
Índice

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( )
{
...
}
```

Método máxicos __get e __set

PHP incorpora uns métodos máxicos, ***__get*** e ***__set***, que nos evitan declarar un método set e get para cada propiedade da clase. Cando tentamos acceder a un atributo como se fose público, PHP chama automaticamente a __get, e cando establecemos o valor a un atributo chama a ao método __set.

```
<?PHP
class Persoa
{
   private $nome;
   private $apelido1;
   private $apelido2;
   public function __construct($nome, $apelido1, $apelido2)
   {
     $this->nome = $nome;
     $this->apelido1 = $apelido1;
     $this->apelido2 = $apelido2;
   }
   public function __set($atributo, $valor)
   {
      if (property_exists(__CLASS__, $atributo))
      {
         $this->$atributo = $valor;
      }
      else
      {
      echo "Non existe o atributo $atributo.";
      }
   }
   public function __get($atributo)
   {
      if (property_exists(__CLASS__, $atributo))
      {
      return $this->$atributo;
      }
   return NULL;
   }
}

$persoa1 = new Persoa("Xan", "Sueiro", "Ferreiro");
echo $persoa1->NIF; //intento mostrar un atributo que no existe
$persoa1->nome = "Xián"; //cambio o valor do atributo nome
echo $persoa1->nome; //mostro o novo valor
?>
```

Outros métodos máxicos

Os métodos máxicos son métodos que se chaman automaticamente por PHP cando ocorre algunha acción sobre un obxecto ou clase. Empezan pola dobre barra baixa '__'. Por exemplo, chámase automaticamente ao construtor cando se crea un obxecto.
Existen outros métodos máxicos, entre os que están:

- **__isset():** Método que se dispara cando tratamos de comprobar se existe un atributo ou se comprobamos o seu contito empregando a función empty( ). Recibe o nome de atributo e deberá devolver true ou false. Por exemplo:

```
class Persoa {
        private $nome;
        public function __set($nome, $valor) {
            $this->$nome = $valor;
        }
        public function __isset($atributo) {
            return isset($this->$atributo);
        }
    }
    $p = new Persoa();
    $p->nome = "Xan";
    if(isset($p->nome)) {
        echo "atributo definido";
    }
    else {
        echo "atributo non definido";
    }
```

- **__unset():** Método que se dispara cando se tenta destruír un atributo que non existe ou é privado empregando a función unset( ). Así modificamos o comportamento de unset( ):

```
function __unset($atributo) {
          if(isset($this->$atributo))
               unset($this->$atributo);
    }
```

- **__toString**: Método que devolve un string cando se usa o obxecto como se fose unha cadea de texto, por exemplo cun echo ou cun print. Deverá devolver un string,

```
class Persoa
{
        private $_nome;
        public function __construct( $nome ){
           $this->_nome = $nome;
        }
        public function __toString(){
           return $this->_nome;
        }
}
$person = new Persoa('Xan');
echo $person;
```

- **__invoke( )** : Dispárase automaticamente cando se tenta chamar a un obxecto como se fose unha función:

```
 class Persoa {
      public function __invoke() {
          echo "Son unha persoa";
      }
  }
  $p = new Persoa();
  $p();  //Isto chamaría ao método máxico __invoke
```

- **__call :** Dispárase automáticamente cando se chama a un método que non está definido na clase ou que é inaccesible dentro do obxecto (por exemplo: cando se trata dun método privado). Recibe como parámetros o nome do método empregado na chamada, e un array coa lista de argumentos que lle estábamos a pasar.

```
class Clase
   {
       public function __call($metodo, $parametros){
           $str = "Método inaccesible: <br />" . $metodo .
                  "<br /> Parámetros: <br /> ";
           // mostramos os parámetros pasados ao método
           foreach($parametros as $parametro){
               $str .= " ". $parametro . "<br />";
           }
           echo $str;
       }
   }
$a = new Clase();
$a->metodoNoNExistente(TRUE,'dato',23);
```

Clases abstractas

As **clases abstractas** son aquelas que NON poden ser concretadas: NON permite instancias do mesmo. A idea é definir algunhas propiedades e métodos que obrigatoriamente teñan que estar definidos nas súas clases fillas.

Exemplo:

```
   abstract class Persoa
   {
   protected $nome;
   protected $apelidos;
   ...

   }

   class Empregado extends Persoa
   {
   ...  //HERDARÁ AS PROPIEDADES DA SÚA CLASE NAI
   }
```

Se definimos algún método como *abstract* na clase abstracta o método ten que estar definido obrigatoriamente nas clases fillas, pero na clase abstracta só se define a súa cabeceira:

```
abstract class Persoa
{
...
abstract public function ocupacionPrincipal();

}  //FIN DA CLASE PERSOA

class Empregado extends Persoa
{
  ...
  public function ocupacionPrincipal()
  {
      return 'traballar';
  }

}
```

Agora non estaría permitido crear instancias da clase Persoa, e ao mesmo tempo obrígase a todas as clases fillas a que implementen o método abstracto.

Toda clase que leve un método abstracto é automaticamente abstracta e non pode ter instancias desa clase.

Polimorfismo

O polimorfismo refírese á capacidade de que unha función ou identificador teña diferente comportamento en función do contexto no que é executado.

Por exemplo, temos unha clase abstracta que serve de base a varias derivadas, e nelas definimos a mesma función con diferente comportamento.

Exemplo:

```
<<?php

abstract class Coche
{
   abstract public function consumir() ;
}

class CocheGasolina extends Coche
{
   public function consumir()
   {
   return 'O ' . __CLASS__ . ' consume Gasolina';
   }
}

class CocheElectrico extends Coche
{
   public function consumir()
   {
   return 'O ' . __CLASS__ . ' consume electricidade';
   }
}

// A FUNCIÓN mostra() EMPREGA UN OBXECTO Coche COMO PARÁMETRO: EN TEMPO DE
// EXECUCIÓN DECIDIRÁ CAL FUNCIÓN É A APROPIADA.

function mostrar(Coche $coche)
{
   echo $coche->consumir(). "<br>";
}

$cochegasolina = new CocheGasolina();
$cocheelectrico = new CocheElectrico();

mostrar($cochegasolina);
mostrar($cocheelectrico);
```

Interfaces

Unha **interface** permite indicar as características dunha clase: permite definir os métodos que deben ser declarados nunha clase de xeito similar ás clases abstractas. A principal diferencia é que non se pode herdar dunha interfaz: as clases implementan unha interfaz ou varias interfaces.
Unha **interface** pode declarar só métodos, non propiedades, e eses métodos non poden ter implementada a súa lóxica.

```
interface Mostrando
{
   public function listaTodo();
}
```

```
class Persoa implements Mostrando
{
    ...
    public function listaTodo()
    {
      //MÉTODO ONDE LISTO TODAS AS PROPIEDADES DE Persoa
      ...
    }
```

Cando sabemos a interface que implementa unha clase coñecemos os métodos que ten definidos, que parámetros recibirán.
Ten en conta que unha clase podería implementar varias interfaces:

```
class Persoa implements Mostrando, nomeInterfaz2, ...
{
   //DEFINICIÓN DE TODOS OS MÉTODOS DAS INTERFACES Mostrando e nomeInterfaz2
   ...
}
```

Existen algunhas 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
		{ ... }
		}

Traits (Trazos ou rasgos)

Os traits son un mecanismo para reutilizar código, nas linguaxes con herdanza simple, como PHP.
Un trait é similar a unha clase, co único obxectivo de agrupar elementos específicos dun xeito coherente. A diferencia é que un trait non se pode instanciar directamente. Así , permite engadir características á herdanza tradicional, e permite combinar membros de clases sen ter que empregar a herdanza:

```
trait OlaMundo
{
    private $saudo='Ola mundo';
    public function getSaudo( )
   {
      return $this->saudo;
}

class Cliente
{
    use OlaMundo;
```

```
    ...
}

$anton = new Cliente();
echo $anton->getSaudo( );   //MOSTRARÍA 'Ola mundo'
```

Unha clase podería usar varios ***traits***:

```
class Cliente
{
   use OlaMundo, atalogoMundo;
   ...
}
```

Espazos de nomes

Se o noso proxecto é grande, ou se empregamos clases de outros programadores, podemos precisar ter clases ou outros elementos co mesmo nome. Para diferencialas empréganse os espazos de nomes, que é como se fose un directorio para **clases**, **traits**, **interfaces**, **funcións** e **constantes**. Debe ser a primeira liña do noso ficheiro para que todo o seguinte estea nese *namespace*.

```
//exemploNamespace.php
<?php
namespace proxecto;
const E = 2.7182;
function saudo(){
    echo "Bos días";
}
class Proba{
    private $nome;
    public function probando(){
        echo "Método probando da clase Proba";
    }
}
```

Emprego dos espazos de nome

Agora podemos empregar o archivo de 2 xeitos:

```
<?php
include "exemploNamespace.php";
echo proxecto\E; // accedemos á constante
proxecto\saudo(); // accedemos á función
$proba = new proxecto\Proba();
$proba->probando();
```

ou tamén:

```
<?php
include "exemploNamespace.php";
use const proxecto\E;
use function proxecto\saudo;
use proxecto\Proba;
// e agora xa podemos usalos
echo E;
saudo();
$proba = new Proba();
$proba->probando();
```

Excepcións. Tratamento de erros

En PHP podemos empregar o modelo de tratamento de excepcións que existen noutras linguaxes de programación. A excepción pode ser lanzada e atrapada:

- meteremos nun bloque **try { }** o código susceptible de producir algún erro.
- Se se produce un erro podemos lanzar unha excepción empregando a instrución **throw.**
- O bloque try terá como mínimo un bloque **catch** que se encargará de procesar o erro
- Se non houbo ningún erro séguese coa execución a continuación do bloque catch. A diferencia doutras linguaxes de programación, teremos acceso ás variables que se crearon dentro do bloque **try**.

Se lanzamos unha excepción podemos enviar unha mensaxe que será despois accedida con ***getMessage().*** Existe tamén un método ***getCode( ),*** que devolve o código de erro se existe.
No exemplo seguinte definimos no método un lanzamento de excepción para poder capturalo en tempo de execución e obrar en consecuencia:

```
<?php
function inverso($x) {
    if (!$x) {
        throw new Exception('División por cero.');
    }
    return 1/$x;
}

try {
    echo inverso(5) . "\n";
    echo inverso(0) . "\n";
} catch (Exception $e) {
    echo 'Excepción capturada: ',  $e->getMessage(), "\n";
}

// Continuar la ejecución
echo 'Hola Mundo\n';
?>
```

Tipado estrito en PHP (strict_types=1)

O tipado estrito é unha característica que permite ter un control máis rigoroso sobre os tipos de datos empregados nas funcións e métodos. A diferencia de outras linguaxes de programación fortemente tipados, PHP non o implementa automaticamente, e temos que habilitalo explicitamente, mediante a declaración **declare(strict_types=1)**, ao comezo dun archivo PHP.

As súas características principais serán:

- Forza ao método a que acepte variables do tipo exacto que se declare
- Aplícase nas chamadas ás funcións dentro do arquivo no que se declara.
- Lanza unn **TypeError** se se tenta usar un tipo de dato incorrecto.

Os seus beneficios son entre outros:

- Mellora a robustez do código
- Prevé erros de tipo en tempo de execución.

Exemplo:

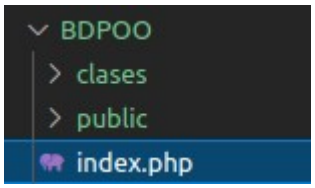
```
<?php
declare(strict_types=1);

class Vehiculo
{
    private string $marca;
    private float $prezo;

    public function __construct(string $marca,  float $prezo)
    {
        $this->marca = $marca;
        $this->precio = $prezo;
    }

    public function getMarca(): string
    {
        return $this->marca;
    }

    public function getPrezo(): float
    {
        return $this->prezo;
    }

    public function aplicarDescuento(float $porcentaxe): void
    {
        $this-> prezo -= $this-> prezo * ($porcentaxe / 100);
    }
}
```

1ª Parte dun CRUD con POO (previo ao patrón modelo vista-controlador)

Móstrase unha parte dun exemplo de como traballaríamos coas Bases de Datos empregando POO con PHP. Un **CRUD** (**Create, Read, Update, Delete**) completo precisaría de definir máis métodos e funcións.
Temos unha táboa cliente

nome	apelidos	email
Ana	Rueiro	anarueiro@hotmail.com
Xan	Rúa	xanrua@gmail.com

A estrutura de carpetas será do seguinte modo:



Distinguiremos as clases (que irán na carpeta **clases)** dos outros ficheiros, que irán na carpeta public. Teremos a nosa páxina de entrada **index.php** na raíz (**DBPOO**)
Nesta 1ª parte móstrase como inserir un cliente, e mostrar todos os clientes da táboa.
Precisamos unha clase **Conexión,** que será filla da clase PDO, e nos permite conectar á base de datos. Teremos despois dispoñibles para esa clase **Conexión** os métodos da clase PDO, podemos traballar con ***PDOStatement***, facer consultas preparadas, etc.

```
//Conexion.php
<?php
declare(strict_types=1);
class Conexion extends PDO
{
   private string $host = "db";
   private string $db = "proba";
   private string $user = "root";
   private string $pass = "root";
   private string $dsn;

   public function __construct() {
      $this->dsn = "mysql:host={$this->host};dbname={$this->db};charset=utf8mb4";
      try{
         parent::__construct($this->dsn, $this->user, $this->pass);
         $this->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
         }catch(PDOException $e){
            die("Erro na conexión: mensaxe: " . $e->getMessage());
         }
      }
}
?>
```

Definimos unha clase **Cliente** coas propiedades que coinciden cos campos da nosa táboa. Para acurtar non se mostran os getters nin setters. Teremos por agora un método para inserir un obxecto e outro para devolver todos os rexistros da táboa:

```
<?php
declare(strict_types=1);
include_once('Conexion.php');

class Cliente
{
   private string $nome;
   private string $apelidos;
   private string $email;
   const TABLA = 'clientes';

   public function __construct(string $nom, string $apel, string $mail){
   $this->nome = $nom;
   $this->apelidos = $apel;
   $this->email = $mail;
   }

   public function mostra(): void {
      echo "Nome:{$this->nome}, apelidos:{$this->apelidos}, email:{$this->email} <br>";
   }

   /* PARA Insertar un obxecto*/
      public function gardar(): bool {
      $conexion = new Conexion();
      try {
         $pdoStmt = $conexion->prepare('INSERT INTO ' . self::TABLA .' (nome , apelidos, email)
            VALUES(:nome, :apelidos, :email)');
         $pdoStmt->bindParam(':nome', $this->nome);
         $pdoStmt->bindParam(':apelidos', $this->apelidos);
         $pdoStmt->bindParam(':email', $this->email);
         $pdoStmt->execute();
      } catch (PDOException $e) {
            die ("Houbo un erro coa inserción: ". $e->getMessage());
      }
      $conexion = null;
      return true; //Se todo foi ben devolver true
   }
   //DEVOLVE un array con todos os clientes da táboa. Método de clase (static)
   public static function devolveTodos() : PDOStatement {
   $conexion = new Conexion();
   try {
      $consulta = "select * from clientes";
      $pdoStmt = $conexion->prepare($consulta);
      $pdoStmt->execute();
   } catch (PDOException $e) {
      die ("Houbo un erro en devolveTodos". $e->getMessage());
   }
   return $pdoStmt; //DEVOLVEMOS TODAS AS FILAS nun PDOStatement
   }

}
```

Teremos un ficheiro **funcions.php** para ordenar o código:

```
<?php
declare(strict_types=1);
function listado():void {
//QUERO LISTAR TODOS
   $clientes = Cliente::devolveTodos();  //UN PDOStatement
   while($filas = $clientes->fetch(PDO::FETCH_OBJ))
         {
            echo $filas->nome." ". $filas->apelidos." ".$filas->email. "<br>";
         }

}

function insertar(Cliente $clienteNovo) :void {

   if($clienteNovo->gardar())
        {
            echo "Cliente gardado correctamente <br>";
        }

}
```

O noso **index.php** permitirá elixir a cal das funcións chamar (novo ou todos):

```
<?php
declare(strict_types=1);
include_once ('cliente.php');

   if(isset($_GET['novo'])) {
      //TERIAMOS QUE RECOLLER OS VALORES DESDE UN FORMULARIO...

      $clientenovo = new Cliente('Ana','conxo','ana@gmail.com');
      insertar($clientenovo);
   }

   if(isset($_GET['todos'])) {
      listado();
   }

?>
```

Programación en capas

A programación en capas é unha arquitectura cliente-servidor na que se se separa a lóxica do deseño da lóxica de negocio (información e acceso á información). O deseño en 3 capas é un dos máis empregados:

- **Capa de presentación:** é a capa que presenta a información ao usuario: será o que ve o usuario. Comunicarase coa capa de negocio.
- **Capa de negocio:** Comunícase coa capa de presentación para recibir os eventos e mostrar resultados, e coa capa de datos, para facer solicitudes tanto de almacenamento como de recuperación de información.
- **Capa de datos:** é onde residen os datos, formada polo xestor da base de datos. Recibirá solicitudes de almacenamento e recuperación de datos da capa de negocio.

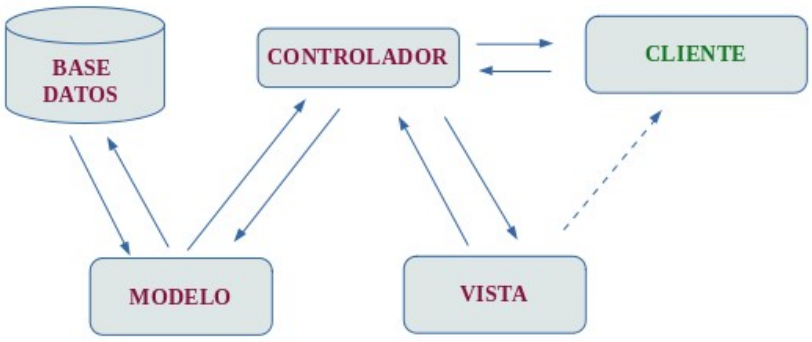
Cada capa é independente, podendo ser feito o desenvolvemento do software en varios niveis, e mesmo por diferentes equipos ou persoas. Será posible posteriormente facer cambios nun único nivel se é preciso. As vantaxes deste modelo de capas é que a nosa aplicación será máis fácil de manter e máis escalable.

Patrón Modelo-Vista-Controlador (MVC)

Na programación orientada a obxectos, un dos patróns máis empregados é o patrón **Modelo-Vista-Controlador**, que separa a lóxica de negocio da interface de usuario, dividindo a aplicación nos seguintes niveis:

- **Modelo**: representa a lóxica de negocios. Se traballamos con un xestor de base de datos, encárgase de almacenar e recuperar a información.
- **Vista**: Presenta a información ao usuario, e recibe eventos
- **Controlador**: fai de intermediario entre o modelo e avista. Recibirá eventos ou peticións do usuario, e fará as solicitudes ao modelo para poder entregar os resultados á vista.

Un esquema simplificado podería ser:



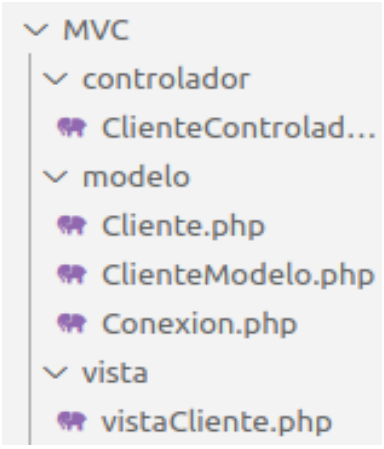
Isto é, o Controlador fará de intermediario entre o modelo e a vista, e o cliente interaccionará co controlador, e pode tamén proporcionar eventos á vista.

Nunha aplicación Web, algunhas das tecnoloxías a empregar serán:

- **Modelo**: Clases de obxectos (PHP, Javascript, etc), Bases de datos (SQL), etc.
- **Vista**: HTML, CSS, Javascript, AJAX, etc
- **Controlador**: código que xera HTML dinamicamente., PHP, AJAX, etc.

Exemplo (incompleto) do patrón Modelo-Vista-Controlador

Pasaremos o exemplo anterior a este patrón MVC. A estrutura de carpetas agora variará:



MODELO

No modelo teremos toda a parte dos datos. A clase **Conexion.php** segue sendo o mesmo:

```
//Conexion.php
<?php
declare(strict_types=1);
class Conexion extends PDO
{
   private string $host = "db";
   private string $db = "proba";
   private string $user = "root";
   private string $pass = "root";
   private string $dsn;

   public function __construct() {
      $this->dsn = "mysql:host={$this->host};dbname={$this->db};charset=utf8mb4";
      try{
         parent::__construct($this->dsn, $this->user, $this->pass);
         $this->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
         }catch(PDOException $e){
            die("Erro na conexión: mensaxe: " . $e->getMessage());
         }
   }
}
?>
```

Podemos separar os métodos de acceso á base de datos da clase base Cliente. Temos así a clase **Cliente.php** (non poño os getters e setters por simplicidade):

```
<?php
//Clase CLIENTE
class Cliente
{
   protected string $nome;
   protected string $apelidos;
   protected string $email;
   public function __construct(string $nom,string $apel, string $mail)
   {
      $this->nome = $nom;
      $this->apelidos = $apel;
      $this->email = $mail;
   }
   // ...

   public function mostra()
   {
      echo "Nome:{$this->nome}, apelidos:{$this->apelidos}, email:{$this->email} <br>";
   }
}
```

Agrupamos os métodos de acceso á base de datos nunha clase **ClienteModelo.php**:

```
<?php
//CLASE CLIENTEMODELO

declare(strict_types=1);
require_once 'Conexion.php';
require_once 'Cliente.php';

class ClienteModelo extends Cliente
{
   public function __construct(string $nome,string  $apelidos,string  $email)
   {
      parent::__construct($nome,$apelidos,$email);
   }

   /* Insertar un obxecto*/
   public function guardar(): bool
   {
   $conexion = new Conexion();
   try {
      $pdoStmt = $conexion->prepare('INSERT INTO clientes(nome, apelidos, email)
            VALUES(:nome, :apelidos, :email)');
      $pdoStmt->bindParam(':nome', $this->nome);
      $pdoStmt->bindParam(':apelidos', $this->apelidos);
      $pdoStmt->bindParam(':email', $this->email);
      $pdoStmt->execute();
      } catch (PDOException $e) {
         die ("Houbo un erro coa inserción: ". $e->getMessage());
   }
```

```
   $conexion = null;
   return true;
   }

   //DEVOLVE un array con todos os clientes da táboa. Método de clase
   public static function devolveTodos() : PDOStatement {
      $conexion = new Conexion();
      try {
         $consulta = "select * from clientes";
         $pdoStmt = $conexion->query($consulta);
      } catch (PDOException $e) {
         die ("Houbo un erro en devolveTodos". $e->getMessage());
      }
      return $pdoStmt;
   }
   //... O RESTO DOS MÉTODOS DO CRUD
}
```

VISTA

En **vistaCliente.php** podemos ter as funcións para mostrar na nosa páxina web, ou poderíamos ter varios ficheiros diferentes, que serán chamados desde o controlador:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
<style type="text/css">
   td,th {border:solid 1px black; }
   table { border-collapse: collapse;}
</style>

</head>

<body>
<?php

function mostraTaboaCliente($array) : void
{
   echo "<table><tr><th>Nome</th><th>Apelidos</th><th>email</th></tr>";
   foreach ($array as $value)
   {
      echo "<td>{$value['nome']}</td><td>{$value['apelidos']}</td><td>{$value['email']}</
         td></tr>" ;
   }
   echo "</table>";
}


/* O RESTO DAS FUNCIÓNS PARA MOSTRAR NA PÁXINA */
//...
?>
</body>
</html>
```

CONTROLADOR

O controlador empregará tanto a parte de control, como da parte da vista. Temos pois **ClienteControlador.php,** que empezará por tanto cos require_once, e despois elixirá en función do que se pida a parte correspondente:

```
<?php
//ClienteControlador.php
require_once('../modelo/ClienteModelo.php');
require_once('../vista/vistaCliente.php');


/* PEDIRiAMOS Á PARTE DA VISTA A PÁXINA DE INICIO */

//SE QUERO LISTAR TODOS
if(isset($_GET['todos']))
{
   $clientes = ClienteModelo::devolveTodos(); //UN PDOStatement. O CONTROLADOR PIDE DATOS
                                              // AO MODELO
   // CONVIRTO A UN ARRAY E O DEVOLVO
   while($fila = $clientes->fetch(PDO::FETCH_ASSOC))
   {
      $clienteArray[]=$fila;
   }
   mostraTaboaCliente($clienteArray); //CHAMO Á FUNCIÓN NA PARTE DA VISTA, PARA MOSTRAR
}

/* if( ...)   SEGUIRÍAN AS DIFERENTES CONDICIÓNS, CHAMANDO AOS MÉTODOS DO CONTROLADOR E
                              MOSTRANDO RESULTADOS COAS FUNCIÓNS DA VISTA
```

Iniciaríamos a nosa aplicación con ***localhost:8000/ClienteControlador.php.*** Probamos a parte de todos con ***localhost:8000/ClienteControlador.php?todos***