Índice

5. Desenvolvemento de aplicacións web con código embebido

5.1 Fundamentos da POO en PHP

Fundamentos da programación orientada a obxectos

Hoxe en día, a meirande parte das linguaxes de programación permiten a *programación orientada a obxectos (POO)*.

Cando falamos de programación orientada a obxectos estamos a falar dunha **técnica** de **programación que se achega máis ao xeito no que pensamos as personas**. A programación orientada a obxectos proporciona ferramentas que permiten ao programador representar os elementos do problema (obxectos) de xeito xeral, sen limitarse a un problema concreto. Non é que non podamos facer as mesmas cousas sen utilizar esta técnica de programación, senón que ao programar usando POO obtemos un código máis lexible e reutilizable.

Podemos pensar en calquera **elemento da vida real** como un sistema baseado en obxectos: un televisor, un teléfono, un coche, o unha persoa poden ser obxectos que teñen unhas determinadas características e capacidades. Por exemplo, un televisor ten un tamaño, resolución, frecuencia de refresco, ... e **pode facer tareas** como acender, apagar, cambiar de canal, ... O mesmo sucede cunha persoa: ten unha estatura, cor de pelo, cor de ollos e outras moitas características que diferencian a unha persoa doutra.

Además, cando temos un elemento do mundo real, somos capaces de saber de qué tipo é en función das súas características. Se vemos un teléfono, sabemos distinguir que se trata dun teléfono e non dun televisor polas características que o definen.

Isto, qué é tan obvio, é a base da programación orientada a obxectos. A idea que temos do que representa unha persoa sería unha ***clase***, mentres unha persoa concreta sería unha ***instancia*** desa clase.

Un exemplo dunha aplicación desenvolvida empregando POO sería a xestión das contas bancarias. Conta bancaria sería unha clase con propiedades como número de conta, saldo, tipo de conta e titular/es. Cliente sería unha clase con propiedades como NIF, nome completo, data de nacemento, enderezo, teléfono, .. E movementos sería una clase con información relativa aos movementos que se realizan nesa conta bancaria.

Unha conta bancaria pode ter como titulares a un ou máis clientes, e unha conta terá movementos, de xeito que estas clases poderían ir enlazadas dende os métodos. Por exemplo, a clase conta bancaría pode ter un método novo_movemento que se executaría por cada movemento que se fixese nesa conta bancaria.

A POO ten como **vantaxes** a reutilización de código, a mellor comprensión, a flexibilidade, a facilidade para dividir as tarefas e a capacidade de ampliar unha aplicación. No noso exemplo, poderiamos ampliar a aplicación incorporando unha xestión de préstamos asignándolle este traballo a unha nova persoa.

Creación de clases

Unha **clase** é un molde do que despois poderemos crear múltiples obxectos, con similares características. As variables que están definidas dentro dunha clase reciben o nome de ***atributos* ou *propiedades*** e as funcións que están definidas dentro dunha clase reciben o nome de ***métodos***. Os métodos da clase son usados para manipular as súas propias propiedades.

A clase define os atributos e métodos comúns aos obxectos dese tipo, pero despois cada obxecto terá os seus propios valores e compartirán as mesmas funcións. Unha clase debe estar definida para poder crear obxectos (instancias) que pertenzan a esta.

As clases defínense empreñando a palabra clave `class` seguida do nome da clase e dun bloque de código entre chaves. A sintaxe é a seguinte:

```
<?php
   class Nome_clase
   {
     //declaración de propiedades
     const CONSTANTE="valor";
     public $propiedad_1;
     public $propiedad_2;

     //declaración de métodos
     function método_1($parametro)
     {
       instrucciones_del_método;
     }
   }
?>
```

Para definir unha **propiedade** con valor variable:

- Ata PHP4 era *obrigatorio* antepor `var` ao nome da variable, a partir de PHP5 basta con indicar o **nivel de acceso** que pode ser **public**, **private** ou **protected** (do que falaremos máis adiante),

  Con PHP5, no caso de declarar unha propiedade utilizando *var* en lugar de *public*, *protected*, ou *private*, PHP tratará dita propiedade como se houbera sido definida como *public*.

- no caso de que queramos **asignarlle un valor** bastará con poñer detrás do nome da variable o signo = e a continuación o valor.

Os métodos definidos nas clases teñen unha sintaxe similar a de calquera outra función en PHP cunha salvedade importante: Dende un método non se pode facer referencia a unha propiedade empregando directamente o seu nome. Deberemos usar a variable **this**.

A variable $this

Esta variable é unha variable especial de auto-referencia, que permite acceder ás propiedades e métodos da clase actual. Por exemplo:

```
<?php
```

```
class Persoa {
  private $nome;

  function set_nome($nom)
  {
    $this->nome=$nom;
  }
  function imprimir()
  {
    echo $this->nome;
  }
   }
?>
```

Creación de objetos

Un obxecto é unha variable que actúa como unha copia dunha clase. Para crear un *obxecto(instancia)* dunha clase determinada debemos usar a seguinte sintaxe:

```
$p = new Persoa()
```

donde *Persoa* é unha clase definida e a variable $p é unha instancia ou copia personalizada (obxecto) da clase Persoa. A palabra clave `new` fai que Persoa xunto con todas as súas propiedades e métodos se copien a $p.

Para **acceder aos seus métodos e atributos públicos** pódese usar o operador frecha:

```
$p->nombre = 'Ximena';
$p->imprimir();
```

Constantes

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 dunha 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, se temos definida a constante NUM_MESES:

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

e creamos un obxecto pertencente a esta clase

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

As dúas seguintes sentenzas serían equivalentes:

```
echo Calendario::NUM_MESES;

echo $cal::NUM_MESES;
```

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.

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 *Libro*, esta debería crearse dentro dun arquito de nome *Libro.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.

Ademais, aínda que os nomes das clases non son sensibles a maiúsculas e minúsculas, moitos programadores prefiren utilizar para as clases nomes que comecen por letra *maiúscula*, para, desta forma, distinguilos dos obxectos e outras variables.

Nivel de acceso ás propiedades e métodos

Un dos fundamentos da POO é a *encapsulación*. Esta baséase en que o código e máis limpo e mellor se o acceso ás propiedades está restrinxido nos seus obxectos. Esto faise empregando os modificadores de acceso que, permiten controlar cómo accede a xente ás clases e, poden ser:

- ***Público(****`public`****):*** é o modificador por defecto, co cal se non poñemos nada enténdese que o modificador é public. Calquera pode acceder ás propiedades e métodos declarados como `public`.
- ***Privado(****`private`****):*** o acceso as propiedades e métodos declarados como private están restrinxidos a clase na que foron creados.
- **Protegido**(`protected`): unicamente poden acceder a estas propiedades e métodos da propia clase e as súas clases derivadas. Isto está relacionado coa herdanza e falaremos dela máis adiante.

Por exemplo, se definimos unha clase cun atributo de cada tipo non poderemos acceder aos atributos privados e protexidos directamente:

```
class Clase
{
    public  $atributoPublico = "Atributo público";
    private  $atributoPrivado = "Atributo privado";
    protected  $atributoProtexido = "Atributo protexído";
```

```
    function imprimirAtributos()
    {
       echo $this->atributoPublico;
       echo $this->atributoPrivado;
       echo $this->atributoProtexido;
    }
}

$a = new Clase();

echo $a->atributoPublico;      //funcionaría correctamente
echo $a->atributoPrivado;      //erróneo
echo $a->atributoProtexido;    //erróneo

$a->imprimirAtributos();       //funcionaría correctamente
```

Un dos motivos para crear atributos privados é que o seu valor forma parte da información interna do obxecto (**principio de ocultación**). Outro motivo é manter control sobre os seus posibles valores Por exemplo, se non queres que se poida cambiar libremente o valor do código dun produto. Ou necesitas coñecer cal será o novo valor antes de asignalo. Nestes casos, adóitanse definir eses atributos como privados e crear dentro da clase métodos para permitirnos obter e/ou modificar os valores deses atributos.

Propiedades e métodos estáticos / non estáticos

As propiedades dunha clase están encapsuladas no obxecto/clase e únicamente son accesibles a través da clase ou o obxecto. Non globalmente.

As propiedades por defecto non son estáticas, co cal forman parte do prototipo do obxecto. Cando creamos un obxecto creanse as propiedades e poden modificarse usando o **operador frecha ->**.

Nalgúns casos queremos que os **atributos ou métodos da clase sexan accesibles sen necesidade de instanciar a clase** (non é necesario crear un obxecto para acceder a eles). En PHP *os elementos de clase* **defínense mediante a palabra `static`**.

Debemos ter en conta que unha propiedade declarada como static non pode ser accedida cun obxecto de clase instanciado (inda que un método estático sí que o pode facer). Así non podemos acceder a unha propiedade estática a través do obxecto empregando o operador frecha `->`.

Para acceder a unha propiedade ou método non se pode usar `$this->propiedade` / `$this->metodo`, senón que que debe empregarse **`self::propiedade /self::metodo`**.

Isto débese a que `$this` usase para as chamadas a propiedades e métodos que están dentro do obxecto (non estáticos), `self` é unha palabra reservada que fai referencia ao nome da clase actual.

No exemplo seguinte definimos unha clase de nome Data que ten unha propiedade (calendario) e un método (getData) estáticos:

```
class Data {
    public static $calendario = "Calendario gregoriano";

    public static function getData(){
       $ano = date('Y');
       $mes = date('m');
```

```
        $dia = date('d');
        return $dia . '/' . $mes . '/' . $ano;
    }
}
```

Co cal para amosar a propiedade deberiamos empregar o nome da clase:

```
echo Data::$calendario;
```

E para chamar ao método non necesitaríamos crear un obxecto da clase:

```
echo Data::getData();
```

Construtores e destructores

As clases teñen un método incorporado chamado *construtor*, que lle permite inicializar as propiedades dos obxectos (dar valores ás propiedades) cando instancias (creas) un obxecto.

Se creamos unha función cun nome identico ao nome da clase, está convertirase en construtor, e execútase de forma automática no momento en que se define un novo obxecto con `new`.

No seguinte exemplo imos crear un construtor para a clase Libro que recibe como parámetro o título do libro e actualiza o contador de libros por cada novo obxecto que se crea.

```
class Libro {
   private $titulo;
   private static $cont_libros = 0;

   function Libro($titulo) {
     $this->titulo = $titulo;
     self::$cont_libros++;
   }
   ....
```

Do mesmo xeito que o construtor se executa automáticamente ao crear un novo obxecto, o destrutor execútase no momento no que o obxecto deixa de existir, co cal debemos incluir en este as tarefas que queiramos executar no momento de liberar un obxecto.

PHP5 incorpora os **métodos máxicos `__construct` e `__destruct`**, que como o seu propio nome indican se executarán automáticamente ao crear ou destruir un obxecto da clase.

Un obxecto pode ser destruido, por exemplo, cando é unha variable local a unha función e remata a execución desa función. Tamén podemos destruir un obxecto de xeito explícito empregando o método `unset()`, que permite destruir objetos pasándolle como parámetro o nome do obxecto que queremos destruir.

Así o exemplo anterior, poderíamos amplialo incorporando un destrutor que actualiza o contador de libros:

```
class Libro {
   private $titulo;
   private static $cont_libros = 0;

   function __construct($titulo) {
     $this->titulo = $titulo;
```

```
        self::$cont_libros++;
    }

    function __destruct() {
        self::$cont_libros--;
    }

    public static function num_libros(){
        return self::$cont_libros;
    }
}

 $libro1 = new Libro("O neno mentirán");
 $libro2 = new Libro("Amigos");
 echo "Na biblioteca temos " .Libro::num_libros() . " libros. <br />";
 unset($libro1);
 echo "Se eliminamos un pasamos a ter ".Libro::num_libros()." libros.<br />
```

o que amosará no navegador:

```
Na biblioteca temos 2 libros.

Se eliminamos un pasamos a ter 1 libros.
```

Implementar get y set en PHP

Unha das principais características da POO é a **encapsulación** dos datos que manexamos. Non se debe acceder *directamente* dende o exterior aos datos que manexa un obxecto. Por isto, existe unha convención na meirande parte de linguaxes orientadas a obxetos para crear métodos que acceden ao valor dunha propiedade dende fora do obxeto. Cun destes métodos establecemos un novo valor para a propiedade e co outro o rescatamos. Así posibilitamos que o obxecto teña control total sobre o que fai cos seus datos e simplificamos o acceso a estes dende o exterior. Por exemplo:

```
<?php
class Persona {
   private $nome;
   function setNome($nom) {
      $this->nome = $nom;
   }
   function getNome() {
      return $this->nome;
   }
}
?>
```

E a continuación, usamos os métodos para establecer e mostrar o nome de dúas persoas diferentes:

```
$persona1 = new Persona();
$persona2 = new Persona();
$persona1 ->setNome("Valeria");
$persona2 ->setNome("Antía");
echo "O nome do número 1 é: " . $persona1->getNome();
echo "O nome do número 2 é: " . $persona2->getNome();
```

PHP5 incorpora dous métodos máxicos, `__get` e `__set`, que nos evitan ter que declarar un método `set` e `get` para cada propiedade da clase. Con estes métodos, cando se intenta acceder a un atributo como se fose público, PHP chama automáticamente a _get, e cando establecemos o valor a un atributo chama ao método __set.

`__set` terá como parámetros de entrada o nome do atributo e o valor que se lle quere dar, mentres que `__get` unicamente terá como parámetro o nome do atributo do cal quere obter o valor.

Imos ver un exemplo, onde usamos a función **`property_exists`** para comprobar se existe nesa clase o atributo que se lle pasa como parámetro:

```
<?PHP
class Persona {
   private $nome;
   private $apelido1;
   private $apelido2;

   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;
   }
 }

 $persona1 = new Persona("Sara", "Plaza", "Ferreiro");
 echo $persona1->NIF; //intento mostrar un atributo que no existe
 $persona1->nome = "Saritísima";  //cambio o valor do atributo nome
 echo $persona1->nome; //mostro o novo valor

 ?>
```

Funcións de clases/obxectos

A continuación imos ver as **funcións de clases/obxectos máis empregadas**, que permiten obter informacións relativa ás clases e obxectos, como por exemplo saber a qué clase pertence un obxecto ou se un atributo pertence ou non a unha clase.

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') {
		…
		}

Ademáis destas funcións PHP dispón do operador `instanceof`, un operador de tipo que permite comprobar se unha instancia é ou non dunha clase determinada. Este operador foi introducido en PHP5 para substituir á función `is_a()` que agora está marcada como obsoleta. Un exemplo do seu uso:

```
if ($p instanceof Persoa) {
   echo "O obxecto é da clase Persoa";
}
```

Uso de obxectos

Referencias a obxectos

Unha referencia en PHP5 é un alias, que permite que dúas variables diferentes poidan escribir sobre o mesmo valor. Ou visto doutro xeito, son un mecanismo que nos permite acceder a un mesmo valor dende nomes de variables diferentes e que se comporten como se fosen a mesma variable.

Dende PHP5, o nome da variable e o contido da variable son dúas cousas totalmente distintas que se enlazan no que se chama a táboa de símbolos. Así, cando creamos unha referencia, o que sucede é que se engade un alias de dita variable na táboa de símbolos de PHP. Imos ver isto cun exemplo:

```
$p = new Persoa();
$p->nome = "Xurxo";
$a = $p;
```

Na primeria liña creamos un obxecto de tipo Persoa en memoria e engadimos na táboa de símbolos de PHP unha entrada que indica que a variable $a é unha referencia (apunta) ao obxecto creado.

Ata PHP4, a última liña do código fai unha copia do obxecto, isto é: crea un novo obxecto cos mesmo valores do orixina. Se a continuación modificamos o nome a un dos obxectos o outro mantería o seu valor, xa que se trata de dous obxectos distintos.

Nembargantes, en PHP5 este comportamiento varía xa que esta liña unicamente crearía un novo identificador do mesmo obxecto, de xeito que se modificamos o valor do nome empregando un dos identificadores, o cambio tamén se vería ao ao acceder empregando o outro identificador. Isto é: temos dous identificadores do mesmo obxecto que apuntan á única copia que se almacena do mesmo.

Crear referencias a variables

Como acabamos de ver, en PHP5 o operador = crea un novo identificador a un obxecto que xa existe. Pero este operador aplicado a variables doutros tipos (como números ou cadenas de texto) crea unha copia da mesma.

Para crear referencias a variables podemos usar o operador &. Así, se ao código anterior-lle engadimos:

```
$c = &$a;
```

O que ocurre é que teremos unha variable $c que fai referencia á variable $a. Imos ver un exemplo:

```
class Persoa {
    private $nome;

    public function __construct($nom)
    {
        $this->nome=$nom;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nom)
    {
         $this->nome=$nom;
    }
}

$p = new Persoa('Anxo'); // $p é unha referencia ao novo obxecto creado
$a = $p;                 //  $a é unha referenza ao mesmo obxecto

$c = &$a;                //$c é unha referencia á variable $a

echo '<br />Nome de $a: '.$a->getNome();
echo '<br />Nome de $c: '.$c->getNome();

$c->setNome('Edu');      // cambiamos o nome da Persoa empregando $c
```

```
$a = NULL;              //Eliminamos a referencia que relaciona $a co obxec-
to
                       //co cal $c tamén deixa de estar relacionado con es-
te.

     // A seguinte liña daría error ao perder a referencia
     // da variable $a ao obxecto
//echo '<br />Nome de $c: '.$c->getNome();

echo '<br />Nome de $p cambiado a través da referencia á variable $c: '.$p-
>getNome();   // $p segue apuntando ao obxecto
```

Como vemos neste exemplo, modificar calquera referencia a un obxecto implica modificar o obxecto en sí mesmo. A salida sería:

```
Nome de $a: Anxo
Nome de $c: Anxo
Nome de $p cambiado usando a referencia a variable $c: Edu
```

Clonar obxectos

As veces queeremos facer unha *copia* dun obxecto, non ter dúas referencias ao mesmo obxecto. Empregando a palabra clave `clone` obtemos dúas instancias distintas coas mesmas propiedades.

Se as propiedades teñen valor, os dous obxectos terán o mesmo valor. Se as propiedades son referencias a outras variables obteranse a mesma referencia copiada nos dous obxectos. Así, se o obxecto orixinal tiña unha propiedade que apuntaba a un obxecto X, no obxecto clonado tamén vai apuntar a X.

```
$p = new Persoa();
$p->nome = 'Anxo';
$s = clone($p);
```

Pode suceder que necesitemos facer algún cambio no obxecto ao clonalo, por exemplo darlle un nome especial, limpar algunhas variables que conteñan parámetros que xa non necesitemos, etcétera. Para eso existe o *método máxico* `__clone`.

O método máxico `__clone` dispárase automáticamente no momento de clonar o obxecto mediante a instrucción `clone`, permitíndonos executar sentenzas ao dispararse. Por exemplo, imos cambiar o atributo nome do obxecto ao clonalo:

```
class Persoa {
    private $localidade;
    private $nome;

    public function __construct($nom, $loc)
    {
        $this->nome=$nom;
        $this->localidade=$loc;
    }
    public function __clone(){
        $this->nome = 'Outro nome';
    }
```

```
}

$p = new Persoa( 'Pedro','Lugo' );
$x = clone $p;

var_dump($p);
var_dump($x);
```

e como podemos observar a instrucción `clone` disparou automáticamente o método máxico `__clone` que cambiou o nome do obxecto clonado, dando como resultado:

```
object(Persoa)[1]
  private 'localidade' => string 'Lugo' (length=4)
  private 'nome' => string 'Pedro' (length=5)
object(Persoa)[2]
  private 'localidade' => string 'Lugo' (length=4)
  private 'nome' => string 'Outro nome' (length=10)
```

Comparar obxectos

PHP dispón de dous operadores que permiten comparar obxectos:

- 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.

```
$p = new Persoa();
$p->nome = 'Sara';
$x = clone($p);
$a = &$p;
```

Así neste exemplo (supoñendo que o método máxico `__clone` non está definido):

o resultado de comparar $x == $p será verdadeiro xa que $x e $p son dúas copias idénticas (cos mesmos valores nos seus atributos)

O resultado de comparar $x===$p será falso xa que $x e $p non fan referencia ao mesmo obxecto

O resultado de comparar $a===$p será verdadeiro xa que $a e $p son referencias ao mesmo obxecto.

Implicación de Tipos

A partir da versión 5, PHP incorpora a posibilidade de especificar a clase á que deben pertencer os obxectos que se pasen como parámetros ás funcións. Esta característica chámase implicación de tipos e tamén pode empregarse con outros tipos como interfaces, arrays (dende PHP 5.1).

Chega con indicar o tipo antes do parámetro:

```
public function vendeProduto(Produto $p) {
 …
}
```

Se cando se realiza a chamada, o parámetro non é do tipo axeitado, prodúcese un erro que se poderá capturar.

Outros métodos interceptores ou "máxicos"

Os métodos interceptores, tamén chamados métodos máxicos, son uns métodos que se chaman automáticamente por PHP cando ocurre algunha acción concreta sobre un obxecto ou clase.

Xa vimos algúns métodos máxicos como o construtor, que se chama automáticamente cando se crea un obxecto. A continuación imos ver outros métodos máxicos:

- `__isset`: Este método dispárase automáticamente cando tratamos de comprobar que un atributo existe usando a función `isset()` ou se comprobamos o seu contido empregando a función `empty()`. Como parámetro recibe o nome do atributo e debe devolver un valor booleano representando a existencia dun valor. Con isto podemos alterar o comportamento, ou simplemente devolver `true` ou `false` á petición. 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();
$p->nome = "Pilar";
if(isset($p->nome)) {
    echo "atributo definido";
}
else {
    echo "atributo non definido";
}
```

- `__unset`: Este método dispárase automáticamente cando se intenta destruir un atributo que non existe uo é privado empregando a función `unset()`, o que nos permite modificar o comportamento desta función.

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

- `__toString`: Este método devolverá un string cando se usa o obxecto coma se fose unha cadea de texte, por exemplo dentro dun `echo` ou dun `print`. Debe devolver unha cadea de texto.

```
class Persoa
{
     private $_nome;
```

```
        public function __construct( $nome ){
           $this->_nome = $nome;
        }
        public function __toString(){
           return $this->_nome;
        }
    }

    $p = new Persoa( 'Mariña' );
    echo $p; //Escribirá o nome: Mariña
```

- `__invoke()`: Este método dispararase automáticamente cando se intenta 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`: Esté método 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->metodoNoExiste(TRUE,'dato',23);
```

- `__callstatic`: O método anterior `__call` era lanzado ao intentar chamar a un método inaccesible no contexto do obxecto, pero non está pensado se o método é inaccesible nun contexto estático. Se chamamos a un método estático que non existe, non se disparará automáticamente `__call` inda que esté definido, senón `__callstatic`. Por exemplo se no código anterior tivésemos

```
Clase::NoExiste();
```

  executaríase o método `__callstatic` no caso de estar definido.

- `__sleep`: este método e o seguinte permiten preparar un obxecto para ser serializado e reconstruir o que sexa necesario tras unha deserialización. Así, `__sleep` dispararase

automáticamente coa función `serialize()` e devolve un array cos atributos que queremos que se amosen na representación do obxecto (serialización)

```
    public function __sleep() {
        return array("nome", "email");  //indicamos os atributos a seriali-
zar
    }
```

- `__wakeup`: Este método dispárase automáticamente cando se aplica a función `unserialize()` sobre o obxecto. Non acepta argumentos e non devolve nada en especial. Sirve para restablecer conexións con bases de datos ou alterar algún atributo que se teña perdido coa serialización.

Ademáis dos métodos máxicos anteriores (e algún outro que se pode consultar na documentación de PHP), PHP resérvase o posible uso futuro como método máxico de calquera nome de método que comence por un dobre guión baixo "__" (underscore), polo que é recomendable non empregalos para outros fines nas clases que desenvolvamos.