Índice

Programación orientada a objetos

La programación orientada a objetos (POO, u OOP en lenguaje inglés), es una metodología de programación basada en objetos. Un objeto es una estructura que contiene datos y el código que los maneja.

La estructura de los objetos se define en las clases. En ellas se escribe el código que define el comportamiento de los objetos y se indican los miembros que formarán parte de los objetos de dicha clase. Entre los miembros de una clase puede haber:

- **Métodos**. Son los miembros de la clase que contienen el código de la misma. Un método es como una función. Puede recibir parámetros y devolver valores.
- **Atributos o propiedades**. Almacenan información acerca del estado del objeto al que pertenecen (y por tanto, su valor puede ser distinto para cada uno de los objetos de la misma clase).

A la creación de un objeto basado en una clase se le llama instanciar una clase y al objeto obtenido también se le conoce como **instancia de esa clase**.

Los pilares fundamentales de la POO son:

- **Herencia**. Es el proceso de crear una clase a partir de otra, heredando su comportamiento y características y pudiendo redefinirlos.
- **Abstracción**. Hace referencia a que cada clase oculta en su interior las peculiaridades de su implementación, y presenta al exterior una serie de métodos (interface) cuyo comportamiento está bien definido. Visto desde el exterior, cada objeto es un ente abstracto que realiza un trabajo.
- **Polimorfismo**. Un mismo método puede tener comportamientos distintos en función del objeto con que se utilice.
- **Encapsulación**. En la POO se juntan en un mismo lugar los datos y el código que los manipula.

Las ventajas más importantes que aporta la programación orientada a objetos son:

- **Modularidad**. La POO permite dividir los programas en partes o módulos más pequeños, que son independientes unos de otros pero pueden comunicarse entre ellos.
- **Extensibilidad**. Si se desean añadir nuevas características a una aplicación, la POO facilita esta tarea de dos formas: añadiendo nuevos métodos al código, o creando nuevos objetos que extienden el comportamiento de los ya existentes.
- **Mantenimiento**. Los programas desarrollados utilizando POO son más sencillos de mantener, debido a la modularidad antes comentada. También ayuda seguir ciertas convenciones al escribirlos, por ejemplo, escribir cada clase en un fichero propio. No debe haber dos clases en un mismo fichero, ni otro código aparte del propio de la clase.

Creación de clases

La declaración de una clase en PHP se hace utilizando la palabra class. A continuación y entre llaves, deben figurar los miembros de la clase. Conviene hacerlo de forma ordenada,

primero las propiedades o atributos, y después los métodos, cada uno con su código respectivo.

```
class Producto
{
    private $codigo;
    public $nombre;
    public $PVP;

    public function muestra() {
        echo "<p>" . $this->codigo . "</p>";
    }
}
```

Es preferible que cada clase figure en su propio fichero (**producto.php**). Además, muchos programadores prefieren utilizar para las clases nombres que comienzan por letra mayúscula, para de esta forma, distinguirlos de los objetos y otras variables.
Una vez definida la clase, podemos usar la palabra new para instanciar objetos de la siguiente forma:

```
$p = new Producto();
```

Para que la línea anterior se ejecute sin error, previamente debemos haber declarado la clase. Para ello, en ese mismo fichero tendrás que incluir la clase poniendo algo como:

```
require_once('producto.php');
```

Los atributos de una clase son similares a las variables de PHP. Es posible indicar un valor en la declaración de la clase. En este caso, todos los objetos que se instancian a partir de esa clase, partirán con ese valor por defecto en el atributo.

Para acceder desde un objeto a sus atributos o a los métodos de la clase, debes utilizar el **operador flecha** (fíjate que sólo se pone el símbolo $ delante del nombre del objeto):

```
$p->nombre = 'LG';
$p->muestra();
```

Cuando se declara un atributo, se debe indicar su nivel de acceso. Los principales niveles son:

- **public**. Los atributos declarados como public pueden utilizarse directamente por los objetos de la clase. Es el caso del atributo **$nombre** anterior.
- **private**. Los atributos declarados como private sólo pueden ser accedidos y modificados por los métodos definidos en la clase, no directamente por los objetos de la misma. Es el caso del atributo **$codigo**.

Uno de los motivos para crear atributos privados es que su valor forma parte de la información interna del objeto. Otro motivo es mantener cierto control sobre sus posibles valores.

Por ejemplo, no quieres que se pueda cambiar libremente el valor del código de un producto. O necesitas conocer cuál será el nuevo valor antes de asignarlo. En estos casos, se suelen definir esos atributos como privados y además se crean dentro de la clase métodos para permitirnos obtener y/o modificar los valores de esos atributos. Por ejemplo:

```
private $codigo;
public function setCodigo($nuevo_codigo)
{
  if (noExisteCodigo($nuevo_codigo))
  {
     $this->codigo = $nuevo_codigo;
     return true;
  }
  return false;
}
public function getCodigo()
{
  return $this->codigo;
}
```

Aunque no es obligatorio, el nombre del método que nos permite obtener el valor de un atributo suele empezar por **get**, y el que nos permite modificarlo por **set**.

En PHP5 se introdujeron los llamados métodos mágicos, entre ellos **__set** y **__get**. Si se declaran estos dos métodos en una clase, PHP los invoca automáticamente cuando desde un objeto se intenta usar un atributo no existente o no accesible. Por ejemplo, el código siguiente simula que la clase Producto tiene cualquier atributo que queramos usar.

```
class Producto
{
   private $atributos = array();

   public function __get($atributo)
   {
     return $this->atributos[$atributo];
   }

   public function __set($atributo, $valor)
   {
     this->atributos[$atributo] = $valor;
   }
}
```

```
$p= new Producto();

print_r($p);

$p->noexisto = "Ahora si que existo";

print_r($p);
```

Cuando desde un objeto se invoca un método de la clase, a éste se le pasa siempre una referencia al objeto que hizo la llamada. Esta referencia se almacena en la variable **$this**. Se utiliza, por ejemplo, en el código anterior para tener acceso a los atributos privados del objeto (que sólo son accesibles desde los métodos de la clase).

```
public function muestra()
{
  echo "<p>" . $this->codigo . "</p>";
}
```

Además de métodos y propiedades, en una clase también se pueden definir **constantes**, utilizando la palabra **const**. Es importante que no confundas los atributos con las constantes. Son conceptos distintos: las constantes no pueden cambiar su valor (obviamente, de ahí su nombre), no usan el carácter **$** y, además, su valor va siempre entre comillas y está asociado a la clase, es decir, no existe una copia del mismo en cada objeto. Por tanto, para acceder a las constantes de una clase, se debe utilizar el nombre de la clase y el operador ::, llamado **operador de resolución de ámbito** (que se utiliza para acceder a los elementos de una clase).

```
class DB
{
      const USUARIO = 'dwes';
      ...
}
echo DB::USUARIO;
```

Es importante resaltar que no es necesario que exista ningún objeto de una clase para poder acceder al valor de las constantes que defina. Además, sus nombres suelen escribirse en mayúsculas.

Tampoco se deben confundir las constantes con los miembros estáticos de una clase. En PHP, una clase puede tener atributos o métodos estáticos, también llamados a veces atributos o métodos de clase. Se definen utilizando la palabra clave static.

```
class Producto
{
  private static $num_productos = 0;
```

```
  public static function nuevoProducto()
  {
      self::$num_productos++;
  }
  ...
}
```

Los atributos y métodos estáticos no pueden ser llamados desde un objeto de la clase utilizando el operador →. Si el método o atributo es público, deberá accederse utilizando el nombre de la clase y el operador de resolución de ámbito.

```
Producto::nuevoProducto();
```

Si es privado, como el atributo **$num_productos** en el ejemplo anterior, sólo se podrá acceder a él desde los métodos de la propia clase, utilizando la palabra **self**. De la misma forma que **$this** hace referencia al objeto actual, **self** hace referencia a la clase actual.

```
self::$num_productos ++;
```

Los atributos estáticos de una clase se utilizan para guardar información general sobre la misma, como puede ser el número de objetos que se han instanciado. Sólo existe un valor del atributo, que se almacena a nivel de clase.

Los métodos estáticos suelen realizar alguna tarea específica o devolver un objeto concreto. Por ejemplo, las clases matemáticas suelen tener métodos estáticos para realizar logaritmos o raíces cuadradas. No tiene sentido crear un objeto si lo único que queremos es realizar una operación matemática.

Los métodos estáticos se llaman desde la clase. No es posible llamarlos desde un objeto y por tanto, no podemos usar **$this** dentro de un método estático.

En PHP puedes definir en las clases métodos constructores, que se ejecutan cuando se crea el objeto. El constructor de una clase debe llamarse **__construct**. Se pueden utilizar, por ejemplo, para asignar valores a atributos.

```
class Producto
{
  private static $num_productos = 0;
  private $codigo;

  public function __construct()
  {
      self::$num_productos++;
  }
  ...
}
```

El constructor de una clase puede llamar a otros métodos o tener parámetros, en cuyo caso deberán pasarse cuando se crea el objeto. Sin embargo, sólo puede haber un método constructor en cada clase.

```
class Producto
{
   private static $num_productos = 0;
   private $codigo;

   public function __construct($codigo)
   {
        $this->$codigo = $codigo;
        self::$num_productos++;
   }
   ...
}
```

```
$p = new Producto('MOTOROLA5G');
```

También es posible definir un método destructor, que debe llamarse **__destruct** y permite definir acciones que se ejecutarán cuando se elimine el objeto.

```
class Producto
{
   private static $num_productos = 0;
   private $codigo;

   public function __construct($codigo)
   {
      $this->codigo = $codigo;
      self::$num_productos++;
   }

   public function __destruct()
   {
      self::$num_productos--;
   }
   ...
}
```

```
$p = new Producto('MOTOROLA5G');

unset($p);
```

Utilización de objetos

Ya sabes cómo instanciar un objeto utilizando **new**, y cómo acceder a sus métodos y atributos públicos con el operador **flecha**:

```
$p = new Producto();

$p->nombre = 'MOTOROLA5G);

$p->muestra();
```

Una vez creado un objeto, puedes utilizar el operador instanceof para comprobar si es o no una instancia de una clase determinada.

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

Además, en PHP se incluyen una serie de funciones útiles para el desarrollo de aplicaciones utilizando POO.

Funciones de utilidad para objetos y clases en PHP		
Función	**Ejemplo**	**Significado**
get_class	echo "La clase es: " . get_class($p);	Devuelve el nombre de la clase del objeto.

class_exists	if (class_exists('Producto') {
	$p = new Producto();
	…
	}
		Devuelve true si la clase está definida o false en caso contrario.

get_declared_classes	print_r(get_declared_classes());	Devuelve un array con los nombres de las clases definidas.
class_alias	class_alias('Producto', 'Articulo'); $p = new Articulo();	Crea un alias para una clase.
get_class_methods	print_r(get_class_methods('Producto'));	Devuelve un array con los nombres de los métodos de una clase que son accesibles desde dónde se hace la llamada.
method_exists	if (method_exists('Producto', 'vende') { …	Devuelve true si existe el método en el objeto o la clase que se indica, o false

	}	en caso contrario, independientemente de si es accesible o no.
get_class_vars	print_r(get_class_vars('Producto'));	Devuelve un array con los nombres de los atributos de una clase que son accesibles desde dónde se hace la llamada.
get_object_vars	print_r(get_object_vars($p));	Devuelve un array con los nombres de los atributos de un objeto que son accesibles desde dónde se hace la llamada.

property_exists	if (property_exists('Producto', 'codigo') {
	…
	}
		Devuelve true si existe el atributo en el objeto o la clase que se indica, o false en caso contrario, independientemente de si es accesible o no.

Desde PHP5, puedes indicar en las funciones y métodos de qué clase deben ser los objetos que se pasen como parámetros. Para ello, debes especificar el tipo antes del parámetro.

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

Si cuando se realiza la llamada, el parámetro no es del tipo adecuado, se produce un error que podrías capturar.

Una característica de la POO que debes tener muy en cuenta es qué sucede con los objetos cuando los pasas a una función, o simplemente cuando ejecutas un código como el siguiente:

```
$p = new Producto();

$p->nombre = 'Samsung Galaxy S';

$a = $p;
```

En PHP el código anterior simplemente crearía un nuevo identificador del mismo objeto. Esto es, en cuanto se utilice uno cualquiera de los identificadores para cambiar el valor de algún atributo, este cambio se vería también reflejado al acceder utilizando el otro identificador. Recuerda que, aunque haya dos o más identificadores del mismo objeto, en realidad todos se refieren a la única copia que se almacena del mismo.

Para crear nuevos identificadores en PHP a un objeto ya existente, se utiliza el operador =. Sin embargo, como ya sabes, este operador aplicado a variables de otros tipos, crea una copia de la misma. En PHP puedes crear referencias a variables (como números enteros o cadenas de texto), utilizando el operador &:

```
$a = 'Samsung Galaxy S';

$b = & $a;
```

En el ejemplo anterior, **$b** es una referencia a la variable **$a**. Cuando se cambia el valor de una de ellas, este cambio se refleja en la otra.

Por tanto, en PHP no puedes copiar un objeto utilizando el operador **=**. Si necesitas copiar un objeto, debes utilizar **clone**. Al utilizar **clone** sobre un objeto existente, se crea una copia de todos los atributos del mismo en un nuevo objeto.

```
$p = new Producto();

$p->nombre = 'Samsung Galaxy S';

$a = clone($p);
```

Además, existe una forma sencilla de personalizar la copia para cada clase particular. Por ejemplo, puede suceder que quieras copiar todos los atributos menos alguno. En nuestro ejemplo, al menos el código de cada producto debe ser distinto y, por tanto, quizás no tenga sentido copiarlo al crear un nuevo objeto. Si éste fuera el caso, puedes crear un método de nombre **__clone** en la clase. Este método se llamará automáticamente después de copiar todos los atributos en el nuevo objeto.

```
class Producto
{
   ...
   public function __clone($atributo)
   {
     $this->codigo = nuevo_codigo();
   }
   ...
}
```

A veces tienes dos objetos y quieres saber su relación exacta. Para eso, en PHP puedes utilizar los operadores == y ===.

Si utilizas el operador de comparación ==, comparas los valores de los atributos de los objetos. Por tanto, dos objetos serán iguales si son instancias de la misma clase y, además, sus atributos tienen los mismos valores.

```
$p = new Producto();
```

```
$p->nombre = 'Samsung Galaxy S';

$a = clone($p);

// El resultado de comparar $a == $p da verdadero

// pues $a y $p son dos copias idénticas
```

Sin embargo, si utilizas el operador **===**, el resultado de la comparación será true sólo cuando las dos variables sean referencias al mismo objeto.

```
$p = new Producto();

$p->nombre = 'Samsung Galaxy S';

$a = clone($p);

// El resultado de comparar $a === $p da falso

//  pues $a y $p no hacen referencia al mismo objeto

$a = & $p;

// Ahora el resultado de comparar $a === $p da verdadero

//  pues $a y $p son referencias al mismo objeto.
```

Mecanismos de mantenimiento del estado

Hemos aprendido a usar la sesión del usuario para almacenar el estado de las variables, y poder recuperarlo cuando sea necesario. El proceso es muy sencillo; se utiliza el array **$_SESSION**, añadiendo nuevos elementos para ir guardando la información en la sesión.

El procedimiento para almacenar objetos es similar, pero hay una diferencia importante. Todas las variables almacenan su información en memoria de una forma u otra según su tipo. Los objetos, sin embargo, no tienen un único tipo. Cada objeto tendrá unos atributos u otros en función de su clase. Por tanto, para almacenar los objetos en la sesión del usuario, hace falta convertirlos a un formato estándar. Este proceso se llama serialización.

En PHP, para serializar un objeto se utiliza la función **serialize**. El resultado obtenido es un **string** que contiene un flujo de bytes, en el que se encuentran definidos todos los valores del objeto.

```
$p = new Producto();

$a = serialize($p);
```

Esta cadena se puede almacenar en cualquier parte, como puede ser la sesión del usuario, o una base de datos. A partir de ella, es posible reconstruir el objeto original utilizando la función **unserialize**.

```
$p = unserialize($a);
```

Debes conocer
Las funciones **serialize** y **unserialize** se utilizan mucho con objetos, pero sirven para convertir en una cadena cualquier tipo de dato, excepto el tipo **resource** (Un valor tipo resource es una variable especial, que contiene una referencia a un recurso externo). Cuando se aplican a un objeto, convierten y recuperan toda la información del mismo, incluyendo sus atributos privados. La única información que no se puede mantener utilizando estas funciones es la que contienen los atributos estáticos de las clases.

Si simplemente queremos almacenar un objeto en la sesión del usuario, deberíamos hacer por tanto:

```
session_start();

$_SESSION['producto'] = serialize($p);
```

Pero en PHP esto aún es más fácil. Los objetos que se añadan a la sesión del usuario son serializados automáticamente. Por tanto, no es necesario usar **serialize** ni **unserialize**.

```
session_start();

$_SESSION['producto'] = $p;
```

Para poder deserializar un objeto, debe estar definida su clase. Al igual que antes, si lo recuperamos de la información almacenada en la sesión del usuario, no será necesario utilizar la función **unserialize**.

```
session_start();

$p = $_SESSION['producto'];
```

El mantenimiento de los datos en la sesión del usuario no es perfecta; tiene sus limitaciones. Si fuera necesario, es posible almacenar esta información en una base de datos. Para ello tendrás que usar las funciones **serialize** y **unserialize**, pues en este caso PHP ya no realiza la serialización automática.

En PHP además tienes la opción de personalizar el proceso de serialización y deserialización de un objeto, utilizando los métodos mágicos **__sleep** y **__wakeup**. Si en la clase está definido un método con nombre **__sleep**, se ejecuta antes de serializar un objeto. Igualmente, si existe un método **__wakeup**, se ejecuta con cualquier llamada a la función **unserialize**.

Herencia

La herencia es un mecanismo de la POO que nos permite definir nuevas clases en base a otra ya existente. Las nuevas clases que heredan también se conocen con el nombre de **subclases**. La clase de la que heredan se llama clase **base** o **superclase**.

Por ejemplo, en nuestra tienda web vamos a tener productos de distintos tipos. En principio hemos creado para manejarlos una clase llamada **Producto**, con algunos atributos y un método que genera una salida personalizada en formato HTML del código.

```
class Producto
{
   public $codigo;
   public $nombre;
   public $nombre_corto;
   public $PVP;

   public function muestra()
   {
      echo "<p>" . $this->codigo . "</p>";
   }
}
```

Esta clase es muy útil si la única información que tenemos de los distintos productos es la que se muestra arriba. Pero, si quieres personalizar la información que vas a tratar de cada tipo de producto (y almacenar, por ejemplo para los televisores, las pulgadas que tienen o su tecnología de fabricación), puedes crear nuevas clases que hereden de **Producto**. Por ejemplo, **TV**, **Ordenador**, **Movil**.

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

Como puedes ver, para definir una clase que herede de otra, simplemente tienes que utilizar la palabra **extends** indicando la superclase. Los nuevos objetos que se instancian a partir de la subclase son también objetos de la clase base; se puede comprobar utilizando el operador instanceof.

```
$t = new TV();
if ($t instanceof Producto)
{
  // Este código se ejecuta pues la condición es cierta
  ...
```

```
}
```

Antes hemos visto algunas funciones útiles para programar utilizando objetos y clases. Las de la siguiente tabla están además relacionadas con la herencia.

Funciones de utilidad en la herencia en PHP		
Función	**Ejemplo**	**Significado**
get_parent_class	echo "La clase padre es: " . get_parent_class($t);	Devuelve el nombre de la clase padre del objeto o la clase que se indica.

is_subclass_of	if (is_subclass_of($t, 'Producto') {
	…
	}
		Comprueba si el objeto tiene esta clase como uno de sus padres.

La nueva clase hereda todos los atributos y métodos públicos de la clase base, pero no los privados. Si quieres crear en la clase base un método no visible al exterior (como los privados) que se herede a las subclases, debes utilizar la palabra **protected** en lugar de private. Además, puedes redefinir el comportamiento de los métodos existentes en la clase base, simplemente creando en la subclase un nuevo método con el mismo nombre.

```
class TV extends Producto
{
  public $pulgadas;
  public $tecnologia;
  public function muestra()
  {
      echo "<p>" . $this->pulgadas . " pulgadas</p>";
  }
}
```

Existe una forma de evitar que las clases heredadas puedan redefinir el comportamiento de los métodos existentes en la superclase: utilizar la palabra **final**. Si en nuestro ejemplo hubiéramos hecho:

```
class Producto
{
  public $codigo;
  public $nombre;
  public $nombre_corto;
  public $PVP;
  final public function muestra()
  {
      echo "<p>" . $this->codigo . "</p>";
```

```
   }
}
```

En este caso el método **muestra** no podría redefinirse en la clase **TV**.

Incluso se puede declarar una clase utilizando **final**. En este caso no se podrían crear clases heredadas utilizándola como base.

```
final class Producto
{
   ...
}
```

Opuestamente al modificador **final**, existe también **abstract**. Se utiliza de la misma forma, tanto con métodos como con clases completas, pero en lugar de prohibir la herencia, obliga a que se herede. Es decir, una clase con el modificador **abstract** no puede tener objetos que la instancien, pero sí podrá utilizarse de clase base y sus subclases sí podrán utilizarse para instanciar objetos.

```
abstract class Producto
{
   ...
}
```

Y un método en el que se indique **abstract**, debe ser redefinido obligatoriamente por las subclases, y no podrá contener código.

```
class Producto
{
    ...
    abstract public function muestra();
}
```

Obviamente, no se puede declarar una clase como **abstract** y **final** simultáneamente, abstract obliga a que se herede para que se pueda utilizar, mientras que **final** indica que no se podrá heredar.

Vamos a hacer una pequeña modificación en nuestra clase **Producto**. Para facilitar la creación de nuevos objetos, crearemos un constructor al que se le pasará un array con los valores de los atributos del nuevo producto.

```
class Producto
{
    public $codigo;
```

```
    public $nombre;
    public $nombre_corto;
    public $PVP;

    public function __construct($row)
    {
       $this->codigo = $row['cod'];
       $this->nombre = $row['nombre'];
       $this->nombre_corto = $row['nombre_corto'];
       $this->PVP = $row['PVP'];
    }

    public function muestra()
    {
       echo "<p>" . $this->codigo . "</p>";
    }
}
```

¿Qué pasa ahora con la clase **TV**, qué hereda de **Producto**? Cuando crees un nuevo objeto de esa clase, ¿se llamará al constructor de **Producto**? ¿Puedes crear un nuevo constructor específico para **TV** que redefina el comportamiento de la clase base?

Empezando por esta última pregunta, obviamente puedes definir un nuevo constructor para las clases heredadas que redefinen el comportamiento del que existe en la clase base, tal y como harías con cualquier otro método. Y dependiendo de si programas o no el constructor en la clase heredada, se llamará o no automáticamente al constructor de la clase base.
En PHP, si la clase heredada no tiene constructor propio, se llamará automáticamente al constructor de la clase base (si existe). Sin embargo, si la clase heredada define su propio constructor, deberás ser tú el que realice la llamada al constructor de la clase base si lo consideras necesario, utilizando para ello la palabra **parent** y el operador de resolución de ámbito.

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

    public function muestra()
    {
        echo "<p>" . $this->pulgadas . " pulgadas</p>";
    }

    public function __construct($row)
    {
        parent::__construct($row);
        $this->pulgadas = $row['pulgadas'];
```

```
        $this->tecnologia = $row['tecnologia'];
    }
}
```

Ya viste con anterioridad cómo se utilizaba la palabra clave **self** para tener acceso a la clase actual. La palabra **parent** es similar. Al utilizar **parent** haces referencia a la clase base de la actual.

Interfaces

Un interface es como una clase vacía que solamente contiene declaraciones de métodos. Se definen utilizando la palabra **interface**.

Por ejemplo, antes viste que podías crear nuevas clases heredadas de **Producto**, como **TV** o **Ordenador**. También viste que en las subclases podías redefinir el comportamiento del método **muestra** para que generara una salida en HTML diferente para cada tipo de producto.

Si quieres asegurarte de que todos los tipos de productos tengan un método muestra, puedes crear un **interface** como el siguiente.

```
interface iMuestra
{
    public function muestra();
}
```

Y cuando crees las subclases deberás indicar con la palabra **implements** que tienen que implementar los métodos declarados en este interface.

```
class TV extends Producto implements iMuestra
{
    ...
    public function muestra()
    {
        echo "<p>" . $this->pulgadas . " pulgadas</p>";
    }
    ...
}
```

Todos los métodos que se declaren en un interface deben ser públicos. Además de métodos, los interfaces podrán contener constantes pero no atributos

.

Un interface es como un contrato que la clase debe cumplir. Al implementar todos los métodos declarados en el interface se asegura la interoperabilidad entre clases. Si sabes

que una clase implementa un interface determinado, sabes qué nombre tienen sus métodos, qué parámetros les debes pasar y, probablemente, podrás averiguar fácilmente con qué objetivo han sido escritos.

Por ejemplo, en la librería de PHP está definido el interface Countable.

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

Si creas una clase para la cesta de la compra en una tienda web, podrías implementar esta interface para contar los productos que figuran en la misma.

Antes aprendiste que en PHP una clase sólo puede heredar de otra. En PHP no existe la herencia múltiple. Sin embargo, sí es posible crear clases que implementan varios interfaces, simplemente separando la lista de interfaces por comas después de la palabra implements.

```
class TV extends Producto implements iMuestra, Countable
{
    ...
}
```

La única restricción es que los nombres de los métodos que se deban implementar en los distintos interfaces no coincidan. Es decir, en nuestro ejemplo, el interface **iMuestra** no podría contener un método **count**, pues éste ya está declarado en **Countable**.

En PHP también se pueden crear nuevos interfaces heredando de otros ya existentes. Se hace de la misma forma que con las clases, utilizando la palabra **extends**.

Una de las dudas más comunes en POO, es qué solución adoptar en algunas situaciones: interfaces o clases abstractas. Ambas permiten definir reglas para las clases que los implementen o hereden respectivamente. Y ninguna permite instanciar objetos. Las diferencias principales entre ambas opciones son:

- En las clases abstractas, los métodos pueden contener código. Si van a existir varias subclases con un comportamiento común, se podría programar en los métodos de la clase abstracta. Si se opta por un interface, habría que repetir el código en todas las clases que lo implemente.
- Las clases abstractas pueden contener atributos, y los interfaces no.
- No se puede crear una clase que herede de dos clases abstractas, pero sí se puede crear una clase que implemente varios interfaces.

Por ejemplo, en una tienda online va a haber dos tipos de usuarios: clientes y empleados. Si necesitas crear en tu aplicación objetos de tipo Usuario (por ejemplo, para manejar de forma

conjunta a los clientes y a los empleados), tendrías que crear una clase no abstracta con ese nombre, de la que heredarían Cliente y Empleado.

```
class Usuario
{
   ...
}
class Cliente extends Usuario
{
   ...
}
class Empleado extends Usuario
{
   ...
}
```

Pero si no fuera así, tendrías que decidir si crearías o no **Usuario**, y si lo harías como una clase abstracta o como un interface.

Si por ejemplo, quisieras definir en un único sitio los atributos comunes a Cliente y a Empleado, deberías crear una clase abstracta Usuario de la que hereden.

```
abstract class Usuario
{
   public $dni;
   protected $nombre;
   ...
}
```

Pero esto no podrías hacerlo si ya tienes planificada alguna relación de herencia para una de estas dos clases.

Para finalizar con los interfaces, a la lista de funciones de PHP relacionadas con la POO puedes añadir las siguientes.

Funciones de utilidad para interfaces en PHP		
Función	**Ejemplo**	**Significado**
get_declared_interfaces	print_r (get_declared_interfaces());	Devuelve un array con los nombres de los interfaces declarados.
interface_exists	if (interface_exists('iMuestra') { … }	Devuelve true si existe el interface que se indica, o false en caso contrario.