Tema 9
Orientación a Objetos en PHP

Recurrimos a la RAE para averiguar que, del latín, y éste del griego παράδειγμα, **paradigma** se define como una teoría o conjunto de estas cuyo núcleo central se acepta sin cuestionar, suministrando base y modelo para resolver problemas con objeto de avanzar en el conocimiento.

Sabiendo esto, y centrándonos en el campo que nos ocupa, podemos decir que un **paradigma de programación** definirá la manera en que se deben estructurar y organizar las tareas que debe llevar a cabo un programa. Generalmente, los paradigmas se encuentran asociados a diferentes **modelos de computación** y estilos de programación definidos, permitiendo que los lenguajes de programación implementen, a veces de forma parcial, diferentes paradigmas.

Son muchos los paradigmas de programación que podríamos enumerar, aunque PHP se apoya en cinco de ellos: **imperativo**, **estructurado**, **reflexivo** y **orientado a objetos**. El primero de estos se basa en indicar **cómo** deberá realizarse una tarea y no el porqué, imponiéndose como el estándar de facto para la mayoría de los lenguajes de programación. PHP es **estructurado**, incorporando a su vez características de los paradigmas **funcional** y **procedural**. Además, se dice que es **reflexivo**, pues, en su sentido más amplio, tiene la capacidad de observar, analizar y, opcionalmente, modificar la estructura del código en tiempo de ejecución. Finalmente, PHP es **orientado a objetos**, pues incorpora elementos de este paradigma, que estudiaremos más detenidamente a lo largo de este tema.

La mayoría de los lenguajes de programación actuales implementan el paradigma de orientación a objetos, siendo algunos de ellos Java, JavaScript, Swift, C#, VB.NET, Delphi, Python, y el propio PHP, entre otros muchos. Acostumbrados a la programación estructurada, la orientación a objetos nos introduce en una nueva filosofía del pensamiento, que nos obligará a definir una nueva línea de pensamiento a la hora de definir, delimitar, descomponer y desarrollar soluciones para un determinado problema.

Conceptos básicos

Los lenguajes de programación tradicionales basan su estructura en el concepto de procedimiento o función. Sin embargo, en un lenguaje orientado a objetos, el elemento fundamental es el **objeto**. Esto es, una representación de un concepto del mundo real, que no diferirá demasiado de lo que realmente entendemos por objeto en la vida real.

Cuando utilizamos el paradigma de orientación a objetos, modelamos la realidad a través de objetos que representan entidades existentes en el entorno que estamos estudiando, y de los que nos interesa procesar información. En definitiva, programar se convertirá en una manera de representar la realidad a través de una serie de objetos y sus interacciones.

Supongamos un coche: es un objeto del mundo real con una serie de características o **atributos**, como la matrícula, marca, modelo, color, etc. Cualquier coche presenta este conjunto de atributos aunque, lógicamente algunos pueden coincidir y otros no. Como el ser humano tiende por naturaleza a agrupar seres o cosas con características similares en grupos o **clases**, en términos de programación orientada a objetos diremos que nuestro coche, o el del vecino, son **instancias** de la clase coche. Estos, se convierten por tanto, en un concepto genérico que agrupa a todos los objetos con características similares.

Definición de clases en PHP

Describir un objeto implicará detallar tanto sus atributos como su comportamiento. Partimos del ejemplo anterior y definimos la clase persona que, en principio, podría contar con dos atributos: **nombre** y **edad**; ambos caracterizan a una persona

```
class Persona {
  public $nombre ;
  public $edad ;
}
```

Hemos definido una clase con un par de atributos. En realidad, sendos atributos no son más que dos variables precedidas por un grado de visibilidad; esta nos permite especificar el ámbito de las dichas variables. De esta manera, la visibilidad de las propiedades podrá ser **private**, **protected** o **public**. No obstante, si definimos una propiedad como pública, no será necesario especificarlo explícitamente.

El grado de visibilidad irá *in crescendo* de manera que, una propiedad **privada** será accesible desde la clase donde se ha definido. Si se define como **protegida**, su accesibilidad se amplía a clases heredadas y, finalmente, una propiedad **pública** será visible, tanto fuera como dentro de la clase.

Recuerda que, según los estándares de desarrollo, deberemos utilizar notación *CamelCase* para nombrar las clases. Llegados a este punto, podremos crear o **instanciar** tantos objetos de clase Persona como deseemos, para lo cual emplearemos la palabra reservada new

```
$foo = new Persona ;
```

A partir de este momento, la variable **$foo** será un objeto de clase Persona y podremos acceder al valor de sus propiedades utilizando el operador flecha, tal y como vemos en el siguiente ejemplo

```
echo $foo->nombre ;
```

Observa que, aunque la propiedad se llama **$nombre**, hemos omitido el símbolo de dólar tras el operador flecha; esto será siempre así. Además, en este caso, al no haber inicializado el valor de la propiedad, no aparecerá nada en pantalla. Podríamos haberle asignado un valor por defecto, en cuyo caso, sí obtendremos un resultado por pantalla tras ejecutar la línea de código anterior.

Tipado de clases

A partir de la versión 7.4 se introduce el tipado de clases. Se podrá emplear cualquier tipo de datos, excepto **void** o **callable**. Además, únicamente se podrán tipar aquellas propiedades que estén acompañadas por un modificador de visibilidad.

```
class Persona {
   public string $nombre ;
   public int $edad ;
}
```

Si empleamos el tipado de clases, debemos prestar además especial atención a las variables no inicializadas ya que, cuando intentemos acceder a ellas, el intérprete lanzará un error. A esto debemos sumarle el hecho de que, si utilizamos **unset** sobre una propiedad tipada, ésta tomará el valor especial **unitialized**.

Constructor de la clase

Aunque no es obligatorio, generalmente, tras definir las propiedades, podemos añadir el **constructor** de la clase. Este se invocará automáticamente cada vez que se cree una nueva instancia de la clase, proporcionando un lugar idóneo para realizar tareas de inicialización del objeto.

```
public function __construct() { ... }
```

Observamos que hemos definido el método como una función normal y corriente y hemos establecido que será pública; su nombre tendrá que ser obligatoriamente el que vemos y siempre deberemos indicar su visibilidad. El constructor también puede recibir parámetros. Supongamos que, al instanciar la clase, queremos inicializar el nombre y la edad de la persona

```
public function __construct(string $nom, int $edd) {
  $this->nombre = $nom ;
  $this->edad = $edd ;
}
```

Introducimos un nuevo elemento: la *pseudovariable* **$this**. Esta variable crea una referencia a la instancia de la clase, permitiéndonos referenciar propiedades o invocar métodos dentro del contexto de dicho objeto. En el fragmento de código anterior, **$this** nos permitirá acceder a cada una de las propiedades y asignarles el valor que se nos pasa como parámetro. Así, cuando instanciamos cualquier objeto de clase **Persona**, lo haremos como sigue

```
$foo = new Persona("Mikey Walsh", 14) ;
```

Promoción de propiedades

PHP 8 incorpora una nueva característica que ya existe en otros lenguajes como Kotlin o TypeScript. La promoción de propiedades nos permite declararlas directamente en la firma del constructor, lo que simplifica sensiblemente el código del constructor. De esta manera, la definición de la clase Persona quedaría como sigue:

```
class Persona {
  public function __construct(public string $nombre, public string edad)
  { }
}
```

Destructor de la clase

El contrapunto al constructor será, obviamente el **destructor**. Su uso no es obligatorio ya que, cada vez que el intérprete finaliza la ejecución de un script se destruyen automáticamente todas las variables que se han definido en él. Sin embargo, podrá sernos útil en ocasiones cuando deseemos realizar alguna tarea de finalización del objeto. Recuerda que el destructor no recibe ningún parámetro.

```
public function __destruct() { ... }
```

Setters y Getters

Hemos definido nuestra clase y creado una nueva persona, a la que hemos llamado Mikey Walsh que, en breve cumplirá los 15 años. Llegado el momento, tendremos que modificar su edad y, como la propiedad ha sido definida pública, podremos hacerlo sin problema a través del objeto

```
$foo->edad = 15 ;
```

Sin embargo, definir una propiedad como pública viola el principio de **ocultación** de la programación orientada a objetos. Teniendo en cuenta que los atributos de un objeto definen su estado, la propiedad de ocultación nos asegura que cada una de las propiedades de la clase permanecerán aisladas del exterior, garantizando que se produzcan efectos laterales indeseados.

Así, propiedades y métodos que pertenecen a una misma entidad se encontrarán **encapsulados** en un mismo nivel de abstracción, facilitando la cohesión de los componentes y proporcionando una interfaz que define la manera en que podrá interactuar con otros objetos de la misma o diferente clase.

Estableceremos las propiedades de la clase Persona como privadas. Pero, si lo hacemos, ¿cómo podremos acceder a su valor? Fácil. Utilizaremos lo que se conoce como **setter**, esto es, una función que define una interfaz a través de la cual poder modificar el valor de una propiedad de la clase

```
public function setEdad(int $edd) { $this->edad = $edd ; }
```

Ahora podemos establecer el valor de la propiedad utilizando el *setter* que acabamos de definir a tal efecto

```
$foo->setEdad(15) ;
```

Además de los *setters* podemos contar con los **getters**, cuya utilidad cae por su propio peso. Como ambas propiedades de la clase están definidas como privadas, no sólo no podemos acceder a ellas para modificar su valor, sino también para consultarlo. De forma análoga a como hemos hecho con **setEdad**, crearemos un método público que nos permita consultar el valor de las propiedades que deseemos

```
public function getEdad(): int { return $this->edad ; }
```

¿Significa esto que, por cada propiedad privada o protegida definida en nuestra clase, hemos de crear obligatoriamente *setters* y *getters*? Simple y sencillamente, no. Es más,

utilizar de forma masiva este tipo de métodos viola nuevamente los principios de ocultación y encapsulamiento, revelando así detalles de implementación de nuestra clase.

Operador nullsafe

Con la versión 8 de PHP se incorpora otra novedad: el operador **nullsafe** (**?->**). Éste nos ayudará a solucionar el problema que se nos plantea siempre que queremos comprobar si una propiedad es nula o un método devuelve dicho valor. Supongamos la siguiente situación:

```
if ($sesion != null) {
  if ($sesion->usuario!=null)
    $asignaturas = $sesion->usuario->getAsignaturas() ;
```

Simplificamos la sucesión de comprobaciones utilizando el operador **nullsafe** obteniendo un código mucho más limpio y favoreciendo el tiempo de ejecución.

```
$asignaturas = $sesion?->usuario?->getAsignaturas() ;
```

El uso de este operador comprobará el valor de **$sesion** y, si éste es nulo devuelve dicho valor y anula la ejecución del resto de la sentencia; en otro caso, continúa con las comprobaciones.

Principio Tell don’t ask

Existen diferentes tendencias metodológicas a la hora de enfrentarse a esta cuestión. Algunas, como el principio **Tell don’t ask**, nos recuerdan que no debemos preguntar a un objeto por su estado, tomar una decisión y actuar sobre dichos datos, sino simplemente indicarle al objeto qué debe hacer. Esto es, la lógica que estamos implementando no es nuestra responsabilidad, sino del propio objeto.

Supongamos que estamos desarrollando una aplicación para la gestión de un centro educativo y, cada uno de los objetos de clase Persona representan a diferentes alumnos. Imaginemos también que, un alumno podrá matricularse en un determinado curso si tiene una determinada edad. Suponiendo que **$alumno** es un objeto de clase **Persona**, actuaremos normalmente como sigue

```
if ($alumno->getEdad() > 18) $alumno->matricular() ;
```

Es decir, matriculamos al alumno si su edad es mayor al valor establecido. Sin embargo, la metodología *tell don’t ask* nos dice que, la lógica empleada debe ser responsabilidad del

objeto. Así, el método matricular sería el encargado de comprobar si el alumno en cuestión cumple los requisitos para acceder al curso.

Esto no significa que debamos convertirnos en aniquiladores de *setters* y *getters*. Ningún principio de diseño es el Santo Grial de la programación. Simplemente, bastará con conocerlos y aplicar unos u otros según la situación a la que nos estemos enfrentando en cada momento.

Métodos mágicos

PHP introduce un conjunto de métodos especiales que responden a determinados eventos. Conocidos como métodos mágicos, nos permiten determinar cómo debe reaccionar un objeto ante dichos eventos. Conocemos ya dos de ellos: **__construct** y **__destruct**, cuya funcionalidad hemos estudiado anteriormente. Sin embargo, PHP cuenta con un total de quince métodos mágicos. En este libro nos centraremos en los más habituales.

__get

Creeremos realmente en la magia cuando conozcamos la utilidad de este método. Hablábamos anteriormente de los *getters*, y de cómo tendríamos que crear uno por cada propiedad que deseemos sea accesible. Gracias al método mágico __get, podremos acceder al valor de cualquiera de las propiedades definidas en nuestra clase.

```
public function __get($key) { return $this->$key ; }
```

Una vez declarado el método mágico en nuestra clase, podremos acceder a cualquier propiedad como si su visibilidad fuese pública. Supongamos que $foo es una instancia de la clase Persona

```
$foo->nombre ;
```

Si, como dijimos anteriormente, a la hora de acceder a cualquier propiedad de una clase, tenemos que omitir el símbolo **$** después del operador flecha, ¿por qué aquí lo estamos utilizando? Es necesario porque, lo que realmente queremos hacer es utilizar el valor contenido en la variable **$key** como nombre de una propiedad. Si, como hacemos habitualmente, no usáramos el símbolo dólar PHP interpretará que hemos definido una propiedad llamada **$key** en la clase e intentará acceder a ella.

Como vemos, este método simplifica nuestro trabajo, ya que nos evita crear un *getter* por cada propiedad. Sin embargo, hemos de extremar el cuidado ya que, el programador puede intentar acceder a atributos que no existen.

```php
public function __get($key)
{
  if (property_exists($this, $key)) return $this->$key ;
  return new Exception ;
}
```

La función **property_exists** comprobará si una propiedad existe para una determinada clase. En el código anterior, realizamos tal comprobación y, si ésta existe devolvemos el valor de la propiedad; en otro caso, lanzamos una excepción.

__set

Conociendo el funcionamiento de su contrapunto, nos resultará sencillo comprender que **__set** nos permitirá modificar el valor de cualquier propiedad de nuestra clase

```php
public function __set($key, $value) { $this->$key = $value ; }
```

Obviamente, tanto **__get** como **__set** simplifican nuestro trabajo como programadores. Sin embargo, utilizarlos nos proporciona acceso a todas las propiedades de la clase, lo que equivale a declararlas todas como públicas y, como sabemos, esto no respeta precisamente los principios de la programación orientada a objetos. Además, debemos tener mucho cuidado ya que, si la propiedad no existe, la crea.

__call

Si está definido, este método será invocado automáticamente cuando se realiza una llamada a un método que no es accesible en el contexto del objeto. Supongamos que en nuestra clase hemos definido dos métodos privados o protegidos. Sería imposible acceder a ellos a través de una instancia de la clase debido a su visibilidad.

Utilizando **__call** podríamos permitir el acceso al método que nos interese, lanzar una excepción, mostrar un mensaje de error o realizar cualquier otra acción

```php
public function __call($met, $args)
{
  if (method_exists($this, $met)) $this->$met($args) ;
  return new Exception ;
}
```

Observamos que, de manera similar a cómo hicimos anteriormente, utilizamos la función **method_exists** para comprobar que el método existe. En ese caso, lo llamamos y le pasamos los parámetros que, posiblemente se nos hayan proporcionado en la llamada.

Supongamos que en nuestra clase Persona hemos definido un método privado llamado saludo

```
$foo->saludo("¡Hola mundo!") ;
```

Sin embargo, si ejecutamos el código anterior, el intérprete mostrará en pantalla un mensaje de error. Esto sucede porque, el parámetro **$args** del método mágico es de tipo *array* y se propagan de esta manera al método que estamos llamando, pudiendo no coincidir en tipo

```
public function __call($met, $args)
{
  if (method_exists($this, $met))
    call_user_func_array([$this, $met], $args) ;

  return new Exception ;
}
```

Utilizamos la función **call_user_func_array**, tal y como vemos en el ejemplo, para solucionar convenientemente el problema expuesto anteriormente.

__isset

Este método se invoca siempre que se utilicen las funciones **isset** o **empty** sobre propiedades inaccesibles, esto es, privadas o protegidas. Supongamos que queremos comprobar si se ha definido en la clase la propiedad $nombre

```
isset($foo->nombre) ;
```

Obviamente, al no ser visible desde el objeto, obtendremos un mensaje de error. Sin embargo, si hemos definido el método **__isset** en nuestra clase, no se mostrará dicho mensaje y podremos realizar cualquier acción adicional en su lugar

```
public function __isset($prop) { return isset($this->$prop ; }
```

__unset

Cuando trabajamos con objetos, podemos destruir propiedades públicas, pero si intentamos hacer un unset sobre una propiedad privada o protegida se invocará (si existe) automáticamente el método **__unset**. De forma similar a lo que ocurría con **__isset**, éste nos permitirá implementar la lógica necesaria para evitar un mensaje de error y realizar alguna tarea en su lugar.

```php
public function __unset($prop)
{
  echo "Eliminando la propiedad '$prop'.<br/>" ;
  unset($this->$prop) ;
}
```

__invoke

Siempre que esté definido en nuestra clase, este método será invocado si utilizamos el objeto como una función. Supongamos que, como hasta ahora, la variable $foo es un objeto de clase **Persona**.

```php
$foo() ;
```

Al realizar la llamada como si fuese un objeto, de manera automática se invocará el método **__invoke**, donde podremos añadir la lógica que estimemos oportuna.

```php
public function __invoke() { ... }
```

Lógicamente, si este método mágico no existe en la clase, la llamada realizada anteriormente provocaría un error. El método **__invoke** admite también el paso de parámetros, si así fuese necesario.

__clone

Tradicionalmente, cuando copiamos un objeto asignándolo a otra variable, ésta última seguirá haciendo referencia al elemento original. De esta manera, al modificar el valor de una propiedad en uno de los objetos, también variará en el otro.

Entonces, ¿cómo podríamos clonar un objeto en caso de que lo necesitemos? PHP introduce la palabra reservada clone para tal cometido y, finalizada la copia, se llamará al método **__clone** si estuviese definido en la clase.

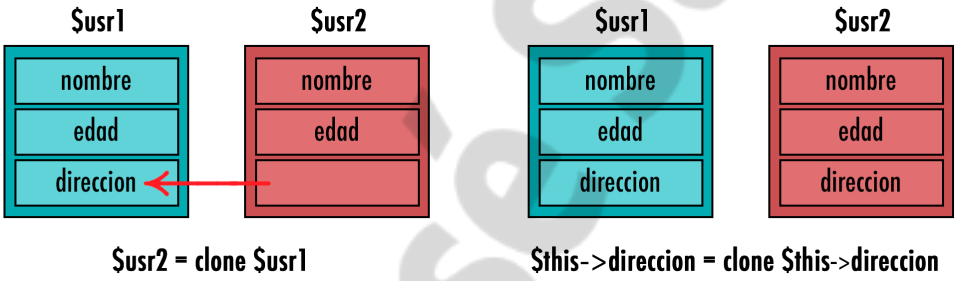
Es importante recordar que, siempre se realiza el clonado manteniendo referencias a otros objetos. Supongamos que, además de **Persona**, tenemos la clase **Dirección**.

```php
class Direccion {
    private $direccion ;
    private $edad ;
    private $localidad ;
    ...
}
class Persona { private $direccion ; ... }
```

También hemos añadido a la clase Persona una nueva propiedad que a su vez, nos permitirá almacenar un objeto de tipo Dirección con objeto de ilustrar la utilidad del método mágico **__clone**. Teniendo en cuenta que **$usr1** es una instancia de **Persona**, supongamos ahora que clonamos el contenido de **$usr1** en **$usr2** utilizando la instrucción **clone**.

```
$usr2 = clone $usr1 ;
```

Según dijimos anteriormente, tendremos dos copias diferentes del mismo objeto. Sin embargo, esto no es del todo cierto. Así como **$usr1** y **$usr2** son completamente diferentes, la referencia interna de la propiedad **$direccion** se mantiene. De esta manera, si cambiamos la dirección para un objeto, también lo estaremos haciendo para el otro



Como el método **__clone** se ejecuta cuando intentamos copiar el valor de **$usr1** en **$usr2**, lo utilizamos para solucionar el problema anterior, clonando los objetos que hayamos creado en su interior; en este caso, **$curso**, tal y como vemos en la imagen anterior

```
public function __clone() {
    $this->direccion = clone $this->direccion ;
}
```

__toString

El *kit* mágico de herramientas de PHP nos proporciona con **__toString** un método para realizar una acción cuando cualquier instancia de una clase es tratada como una cadena. Observa el siguiente ejemplo

```
public function __toString() {
    return "Tu nombre es $this->nombre y tienes $this->edad." ;
}
```

Definido el comportamiento del método, suponiendo que **$foo** es una instancia de la clase Persona, podríamos hacer lo siguiente

```
echo $foo ;
```

Mostrando en pantalla el mensaje:

> *Tu nombre es Mike Walsh y tienes 14 años*

Herencia

Definimos anteriormente la clase Persona para ilustrar el concepto de objeto. Supongamos que continuamos con nuestra aplicación para gestionar alumnos y profesores de un centro educativo. Todos ellos comparten una serie de propiedades comunes, como el nombre, apellidos, edad, dirección, etc. Sin embargo, existen otras características que son propias de uno u otros. En este sentido, podríamos decir que Persona representa a un tipo de objeto muy genérico y limitado, por lo que sería necesario extender sus características.

A este mecanismo se le llama **herencia**. Ciertas clases heredarán las propiedades y comportamiento de una clase padre, sin tener que implementarse nuevamente, permitiendo además la incorporación de características propias que ahora sí, diferencien a profesores de alumnos.

```
class Alumno extends Persona {
    private $nivel ;
    private $grupo ;
   ...
}
```

Como vemos, utilizamos la palabra reservada **extends** para indicar que una clase hereda de otra. A partir de este momento, la clase **Alumno** heredará todas las propiedades y métodos públicos y protegidos de su clase padre. Aún así, si es necesario, podríamos redefinir o **sobrecargar** los métodos heredados para adecuarlos al comportamiento de la clase hija.

```
public function __toString() {
    return "El alumno/a se llama {$this->nombre} y está matriculado en
                                {$this->nivel}{$this->grupo}" ;
}
```

Sobrecargar un método de la clase padre es tan sencillo como volver a implementar su comportamiento. Eso sí, debemos tener en cuenta que, no se llamará al método de la clase padre a no ser que lo invoquemos explícitamente. Supongamos que definimos el constructor de la clase **Alumno** como sigue:

```
public function __construct(private string $nombre,
                            private int $edad,
```

```
                            private int $nivel,
                            private string $grupo) {
    parent::__construct($nombre, $edad) ;
}
```

Observamos que, siendo públicas, podríamos haber accedido directamente a las propiedades **$nombre** y **$edad** y asignarles su valor. Sin embargo, hemos optado por llamar al constructor de la clase padre que, dicho sea de paso, es lo más habitual en circunstancias como esta. Esto nos permite introducir dos nuevos elementos: la palabra reservada parent y el operador de **resolución de ámbito** o Paamayim Nekudotayim (**::**). Utilizamos ambos de forma combinada cuando, desde la clase hija deseemos hacer referencia a elementos estáticos, constantes o métodos de la clase padre.

Seguramente, en alguna ocasión nos será necesario impedir que unas clases hereden de otras. Emplearemos para ello la palabra reservada final a la hora de definir nuestra clase padre, bloqueando de esta manera el mecanismo de herencia

```
final class claseBase { ... }
```

De esta manera, según la definición de **claseBase**, nos será imposible crear otras que hereden de ésta. La palabra final también puede utilizarse para evitar que determinados métodos puedan ser sobrecargados desde las subclases.

Clases abstractas

Básicamente, una clase abstracta es aquella que no puede instanciarse directamente. Supongamos que definimos cómo tal la clase **Persona**.

```
abstract class Persona { ... }
```

Esto implica que no podremos crear objetos de clase **Persona**. Sin embargo, sí podremos hacerlo con sus clases hijas. Esto es, aunque no podamos instanciar un objeto de clase Persona, si podremos hacerlo con las clases **Alumno** y **Profesor**.

Habremos de tener en cuenta que, una clase deberá ser obligatoriamente declarada como abstracta, siempre que definamos como tal uno de sus métodos. Éste no tendrá cuerpo y se declarará como protegido o público, obligando a implementar su funcionalidad en aquellas clases que hereden de ésta.

Constantes de clases

Hasta ahora habíamos definido propiedades y métodos en el interior de una clase, pero también podemos definir un valor y mantenerlo invariable durante la vida de cualquier instancia que hayamos creado de dicha clase. Esto es lo que conocemos como **constantes de clase** y se definen utilizando la palabra reservada **const**.

```
const SALUDO = "¡Hola mundo!" ;
```

Recuerda que, por convención, las constantes deben nombrarse en mayúsculas. Acceder a la constante desde cualquier método nos obligará a utilizar la palabra reservada **self** junto con el operador de resolución de ámbito

```
echo self::SALUDO;
```

Sin embargo, sí deseamos acceder a su valor a través de cualquier instancia de la clase, antepondremos el nombre de ésta al operador

```
echo Persona::SALUDO ;
```

El uso de las constantes no queda restringido únicamente a las clases, sino que también podremos emplearlas en clases abstractas e interfaces, que estudiaremos en el siguiente apartado.

Interfaces

El uso de una interfaz supone un compromiso para aquellas clases que la implementan. Las **interfaces** contienen únicamente la definición de métodos que, obligatoriamente deberán ser públicos y, cuya funcionalidad deberá definirse *a posteriori* en las clases que implementen dicha interfaz.

Supongamos que queremos crear las clases **Circulo**, **Cuadrado** y **Triangulo**. Cada una de ellas modela un determinado objeto geométrico que, lógicamente compartirán características, como su posición en el plano, escalado, etc. Esto es, esas tres clases podrían ser descendientes de otra que podríamos llamar **Figura** y que define todas aquellas propiedades y métodos que son comunes a **Circulo**, **Cuadrado** y **Triangulo**.

Imaginemos ahora que con cada figura podemos realizar diferentes operaciones: **dibujar**, **mover**, **escalar** y **rotar**. En un principio, como todas estas funciones son comunes a cada una de las figuras, nos inclinaremos a definirlas en la clase **Figura**. Sin embargo, enseguida

nos daremos cuenta de que, el algoritmo que dibuja, escala o rota puede ser diferente para cada figura, así que nos olvidamos inmediatamente de implementarlos en la clase padre. Aún así, queremos obligar a que las clases hijas implementen dichos métodos y, aquí es donde nos resultará extremadamente útil el uso de interfaces.

```
interface Objeto {
    public function dibujar() ;
    public function escalar(float $x, float $y) ;
    public function rotar(rotar $grad)  ;
}
```

Hemos definido una interfaz y la hemos llamado **Objeto**. En ella, únicamente se han definido los métodos cuya implementación difiere de entre los diferentes elementos geométricos. Seguidamente, cada una de las clases **Circulo**, **Cuadrado** y **Triangulo**, no sólo heredarán las características de la su clase padre, sino que además implementarán la interfaz

```
class Circulo Extends Figura implements Objeto { ... }
```

Esto obligará a la clase **Circulo**, por ejemplo, a implementar obligatoriamente la lógica de cada uno de los métodos definidos en la interfaz, garantizando de esta manera que cada figura geométrica pueda realizar convenientemente cada una de esas acciones.

Al igual que una clase, las interfaces pueden extenderse de otras interfaces, además de poder definir en su interior constantes, que, aunque se utilizarán de igual manera que las constantes de clase, no podrán ser sobrescritas.

Atributos y métodos de clase

Los atributos y métodos de clase permiten mantener una única copia de éstos para toda la clase. Es decir, si se declara una propiedad de una clase como estática, su valor será el mismo para cada instancia de dicha clase. Es por esto por lo que se dice que los atributos y métodos estáticos existen a nivel de clase y no de objeto. Vamos a utilizar un ejemplo para comprender mejor todo esto.

```
class Persona {
   public static $totalAlumnos = 0 ;
   ...
   public static function getTotalProfesores() {
       return Persona::$totalProfesores ;
   }
}
```

En el fragmento de código anterior destacamos dos aspectos importantes. Primero, hemos utilizado la palabra reservada static para definir dos propiedades estáticas. En segundo lugar, como vemos, el acceso a este tipo de propiedades es un tanto especial; utilizamos el nombre de la clase, seguido del operador de resolución de ámbito y, por último, el nombre de la propiedad, símbolo **$** incluido.

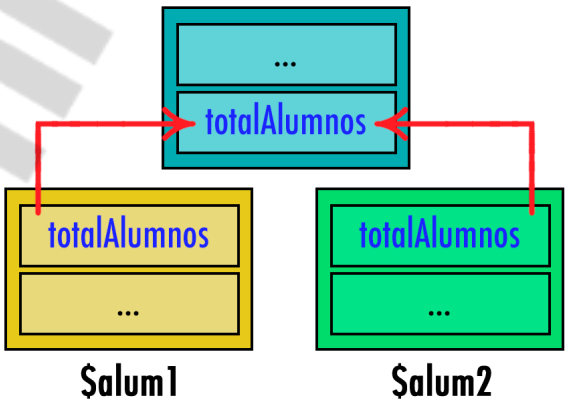
Seguidamente, desde el constructor de cualquiera de las subclases, **Alumno**, por ejemplo, incrementamos el valor de la propiedad correspondiente

```
public function __construct(public string $nom, public int $edd) {
    parent::__construct($nom, $edd) ;
    Persona::$totalAlumnos++ ;
}
```

Cómo precisamente está definida en la clase padre, el acceso a **$totalAlumnos** se hace tal y como se dijo anteriormente, aunque también podremos sustituir el nombre de la clase padre por el de la hija. Supongamos ahora que creamos dos instancias de la clase **Alumno**.

```
$alum1 = new Alumno ;
$alum2 = new Alumno ;
```

Cada ejecución del constructor implica un incremento del total de alumnos. En un principio podríamos pensar que, dado que el valor inicial de dicha propiedad es cero, el valor para ambas instancias será **1**. Sin embargo, esto no es así, ya que la propiedad se ha definido como estática y, esto quiere decir, que compartirá su valor con cada objeto. De esta manera, como hemos creado dos instancias de Alumno, **$totalAlumnos** se ha incrementado dos veces, por lo tanto, su valor será **2**.



En resumen, una propiedad o método estático es compartido por cada instancia de la clase. Acceder a cada una propiedad (pública) o método estático a través de cualquier objeto podrá hacerse de diferentes maneras

```
$alum1::$totalAlumnos ;
Persona::$totalAlumnos ;
Alumno::$totalAlumnos ;
```

Todo esto conlleva que, propiedades y métodos estáticos pueden ser accedidos sin necesidad de instanciar la clase y, precisamente esto es uno de los principales problemas que presenta todo esto. Aunque pueda resultarnos de extremada utilidad, no debemos abusar de su uso ya que rompe completamente con el modelo de orientación a objetos puro, oculta dependencias y produce un alto acoplamiento.

Antes de terminar, hemos pasado por alto un pequeño detalle. Si regresamos a la definición de la clase Persona nos daremos cuenta que, el método **getTotalAlumnos** se ha definido también como estático. Realmente, no parece necesario, ¿cierto? Si me has dado la razón, te equivocas, ya que todo método que haga uso de una variable estática, deberá ser definido obligatoriamente como estático.

Polimorfismo

Junto con la encapsulación y la herencia, el polimorfismo forma parte de los pilares básicos de la orientación a objetos. Como su propio nombre indica, este concepto sugiere la idea de múltiples formas, lo que, cuando hablamos de desarrollo, deriva en la capacidad de acceso a diferentes funciones a través de un mismo interfaz.

Aunque, en principio, el concepto puede resultar confuso, el **polimorfismo** es una característica bastante potente de la programación orientada a objetos. Retomemos el ejemplo sobre figuras geométricas que utilizamos para ilustrar la utilidad de las interfaces.

Si recuerdas, tenemos una clase padre, Figura, y sus tres clases hijas, Circulo, **Cuadrado** y **Triangulo**. Supongamos que, durante el desarrollo de nuestra aplicación es necesario implementar una función que reciba como parámetro cualquier tipo de figura y realice una acción sobre ella. Como cada una de estas se define a través de objetos de distinta clase, podríamos sentirnos tentados de crear una función para cada una. Sin embargo, esto no es necesario, ya que el polimorfismo nos garantiza que un objeto de tipo **Circulo**, **Cuadrado** o **Triangulo**, es también de tipo **Figura**, precisamente porque heredan de éste.

```
function foo(Figura $fig) { ... }
```

Bastará, como vemos, con crear una única función que reciba como parámetro un objeto de tipo Figura, evitando tener que implementar diferentes funciones según el tipo de objeto que reciba. Sabemos que PHP no es un lenguaje fuertemente tipado, por lo que no es

necesario especificar el tipo de los parámetros que recibe una función y, por tanto, el mecanismo de polimorfismo va implícito.

Serialización de objetos

A estas alturas, debemos ser conscientes de que, exceptuando los almacenados en una sesión, al recargar un script en el navegador, perderemos todas las variables que se hayan creado previamente. Los objetos no son diferentes, es más, son una variable más y, si recargamos el script o cambiamos el flujo de la aplicación de alguna manera, perderemos cualquier instancia que se haya creado.

Cómo es posible que, en ocasiones, necesitemos preservar el valor de un objeto en el tiempo, introducimos en este punto el concepto de **serialización de objetos**. Básicamente, consiste en transformar convenientemente cualquier estructura, de manera que pueda almacenarse como una variable.

La función serialize realizará la conversión de un objeto en un flujo de bytes capaz de ser almacenado, manteniendo su tipo y estructura. Recuperamos el valor original utilizando **unserialize**. A modo de ejemplo, retomamos la clase **Alumno**, creamos una instancia y, si no lo hemos hecho previamente, la *serializaremos* para almacenarla en una variable de sesión

```
session_start() ;
if (!$_SESSION["alumno"])
    $_SESSION["alumno"] = serialize(new Alumno( ... )) ;

$alumno = unserialize($_SESSION["alumno"]) ;
echo $alumno ;
```

Estudiemos detenidamente el fragmento de código anterior. Cuando cargamos la página, comprobamos si existe la variable de sesión **alumno**. No existirá porque es la primera vez que accedemos a la web, así que creamos dicha variable y guardamos en ella el resultado de *serializar* nuestro nuevo objeto.

Supongamos ahora que recargamos la página. Activamos la sesión y, como hemos creado previamente la variable alumno, pasamos directamente a *deserializar* nuestro objeto y guardarlo en la variable **$alum**. Obviamente, siempre que realicemos cualquiera de estas operaciones sobre un objeto, será necesario tener definida la clase previamente.

__sleep

Ahora que conocemos el concepto de *serialización*, introducimos dos nuevos métodos mágicos. El primero de ellos, _sleep, será invocado antes de *serializar* un objeto. No recibe ningún parámetro, pero devolverá siempre un *array* con los nombres de las propiedades que deseemos *serializar*.

```php
public function __sleep() { return ["nombre", "edad"] ; }
```

El fragmento de código anterior provocará la *serialización* de un objeto de clase **Alumno**, manteniendo el valor de las propiedades **nombre** y **edad**.

__wakeup

Tras *deserializar* un objeto, si se ha incluido en la clase, se llama a este método. En su interior podremos incluir la lógica necesaria para restablecer elementos que hayan podido perderse durante el proceso de *serialización*, como por ejemplo, conexiones con bases de datos, manejadores de ficheros, etc.

Traits

El paradigma de programación orientada a objetos contempla dos tipos de herencia: **simple** y **múltiple**. La primera de ellas es aquella en la que una clase hereda de tan sólo otra clase, como hemos hecho hasta ahora. Sin embargo, la herencia múltiple permitirá heredar de una o más clases, algo que, como sucede en con otros lenguajes, no podemos hacer directamente en PHP.

Aliviamos las limitaciones de la herencia simple incorporando los **rasgos** o **traits**, facilitándonos la posibilidad de implementar una herencia múltiple, reutilizando conjuntos de métodos sobre clases independientes y pertenecientes a diferentes jerarquías.

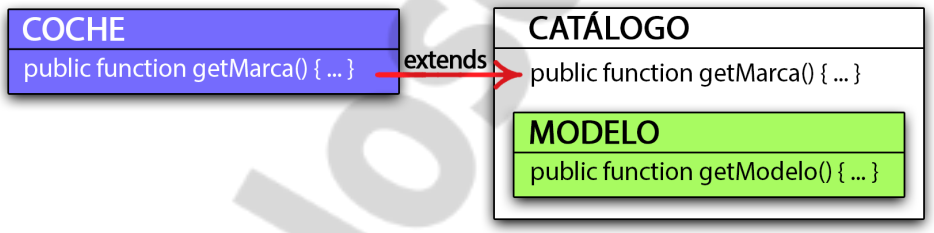
Básicamente, un **trait** es similar a una clase, excepto que no podremos instanciarlo. Los *rasgos* agrupan de forma coherente un conjunto de funcionalidades específicas. Observa el siguiente fragmento de código, con el que se intenta ilustrar la utilidad de este tipo de estructuras.

```php
class Coche {
  public function getMarca() { echo "Citröen" ; }
}

trait Modelo {
  public function getModelo() {
    parent::getMarca() ;
    echo " Xara Picasso." ;
  }
}

class Catalogo extends Coche {
```

```
    use Modelo ;
}
```

Leamos de abajo a arriba el fragmento de código anterior. Definimos **Catalogo** que hereda directamente de la clase **Coche**, permitiendo de esta manera el acceso a las propiedades y métodos, públicos y protegidos, definidos en esta última. La utilización de la palabra reservada **use** en la clase Catalogo, indicando al intérprete que deseamos utilizar además el conjunto de funciones definidas en el **trait Modelo**. Éste implementa el método **getModelo** que, a su vez realiza una llamada al método **getMarca** de su ascendente directo. Pero, un momento, si Modelo no hereda de nadie, ¿quién es su padre?

Si pensamos detenidamente comprenderemos que, la funcionalidad de **use** no es muy diferente a la de **include** o **require**. Cuando indicamos en **Catalogo** que vamos a utilizar el *trait*, los métodos definidos en éste último pasarán a formar parte de la clase., pudiendo, no sólo acceder a cualquier elemento de ésta, sino a aquellos que, por herencia, sean accesibles desde **Catalogo**.



Así, desde cualquier instancia de clase **Catalogo**, podremos acceder a los métodos del **trait**.

```
$foo = new Catalogo ;
$foo->getModelo() ;
```

A su vez, esta invocación provocará una llamada al método **getMarca** de la clase **Coche** que, como es público está dentro de nuestro del ámbito del rasgo. Debemos tener en cuenta que los miembros heredados de una clase se sobreescribirá si insertamos un método homónimo a través de un **trait**. Además, no sólo podremos utilizar múltiples *rasgos* en una clase, sino que también podemos hacer uso de *traits* dentro de otros, así como incluir propiedades y métodos estáticos.

Patrón Singleton

Como cualquier patrón de diseño, **Singleton** no es algo que venga a solucionar todos nuestros problemas. Debemos utilizarlo con sabiduría, de forma apropiada y siempre que pueda ser conveniente, pero no debemos basar todos nuestros desarrollos en él. Puede

resultar algo desalentador empezar a estudiar un patrón de diseño y que nos digan todo esto pero, tal y como veremos a continuación, Singleton dificulta las tareas de testing, aumenta el acoplamiento y presenta ciertas restricciones durante ejecuciones concurrentes.

Hasta aquí sólo hemos señalado los handicaps del patrón Singleton, pero, si presenta tantas desventajas, ¿para qué se utiliza? Y, lo más importante, ¿en qué consiste este patrón? Comenzamos respondiendo a la primera pregunta.

Singleton intenta resolver el problema de una instanciación excesiva de objetos de una misma clase, cuando realmente bastaría con tener un único punto de acceso hacia dicho recurso. Generalmente, utilizaremos este patrón cuando no exista concurrencia, o cuando el objeto es muy pesado en memoria y tener múltiples instancias del mismo generaría un elevado gasto de recursos.

Con respecto a la segunda pregunta que formulamos anteriormente, el patrón Singleton consiste en garantizar que no pueda existir más de una instancia de una determinada clase.

```
class Singleton {
    private static $instancia = null ;
    private function __construct() { }
    public static function getSingletonInstance()
    {
        if (parent::instancia == null)
        parent::instancia = new Singleton ;

        return parent::instancia ;
    }
}
```

Observamos, en primer lugar, que el constructor de la clase se ha definido como privado; de esta manera evitamos que puedan crearse diferentes instancias de nuestra clase. El único objeto que crearemos quedará contenido dentro de la misma clase **Singleton** y lo haremos a través del método **getSingletonInstance**. Éste comprobará si ya previamente hemos instanciado la clase. Si es así, devolvemos el objeto y listo; en otro caso, lo creamos y almacenamos en la propiedad estática **$instancia** para, seguidamente devolverlo.

Garantizamos de esta manera que la clase **Singleton** no pueda ser instanciada en más de una ocasión, por lo que siempre tendremos un único punto de acceso a los métodos encapsulados en esta. Sin embargo, el patrón no estaría completamente implementado hasta no haber controlado la clonación de objeto ya que, si pudiéramos realizar copias de nuestro objeto, no tendría sentido nada de lo que estamos planteando. Solucionamos este inconveniente, sobrecargando el método **__clone**, haciéndolo privado y dejando su cuerpo completamente vacío.

```
private function __clone() { }
```