TEMA 3.1

PHP Orientado a objetos

Programar utilizando POO: Introducción

- Cuando hacemos un "programa", tenemos siempre:
  - Algo de código: Bucles, sentencias if
  - Algunos datos: Quizá en forma de arrays indexados o tipo clave-valor
- En orientación a objetos, lo que hacemos es decir que tenemos objetos, y cada objeto tiene código y datos.
  - Un objeto tendrá un conjunto de funciones (código)
  - Un objeto tendrá un conjunto de datos
- Podemos visualizar, al crear estos objetos, que estamos construyendo librerías para nosotros, y también librerías para otros!!
- **Objetivo**: Encapsular lo más posible para poder compartir
- Esto nos lleva a la **POO** - Programación Orientada a Objetos

Clases y objetos: Terminología

- PHP sigue el paradigma de orientación a objetos basado en clases:
  - Se define una **clase:** Una clase es una "plantilla", el equivalente a un molde. Define las **propiedades** (variables) y **métodos** (funciones) para poder crear objetos de esa clase
  - Se instancian **objetos** de la clase: Al definir una variable como objeto de una clase, sus características son las mismas que las de la clase. Es como si utilizamos el molde, que es la clase, para hacer una galleta, que sería el objeto. Cuando se crea un objeto, se dice que se está "**instanciando**" la clase. Podríamos hacer tantos objetos como quisiéramos de una clase (muchas galletas).
  - Las clases se nombran en *mayúscula*, los objetos en *minúscula*. Por ejemplo, una clase llamada Perro (definición de la especie), no tiene objetos llamados Perro, sino objetos concretos, como toby, rocky, o en todo caso perro (uno más).
  - Siguiendo con el ejemplo, todo objeto de la clase Perro tendrá pelo. En cada instancia el color del pelo podrá cambiar. Así toby lo tendrá negro, rocky blanco, y perro marrón.
- **Visibilidad**: Tanto las propiedades (atributos) como los métodos pueden definirse con una visibilidad:
  - Privada (**private**): Sólo son visibles desde la propia clase
  - Protegida (**protected**): Sólo son visibles desde la propia clase o sus descendientes
  - Público (**public**): Son visibles desde cualquier otra clase.
- En PHP (a partir de la versión 5), *se recomienda* utilizar la programación orientada a objetos. Las librerías están evolucionando hacia esa orientación.

Diseño de clases

- Cuando un proyecto crece, las clases se diseñan mediante UML.

Nombre de la clase

Atributos:
"+" públicos
"-" privados
"#" protegidos

Métodos:
"+" públicos
"-" privados
"#" protegidos

Definición de clases

```php
<?php
class ClaseSencilla
{
    // Declaración de una propiedad
    public $var = 'un valor predeterminado';

    // Declaración de un método
    public function mostrarVar() {
        echo $this->var;
    }
}
?>
```

Importante:
- En PHP el nombre del fichero *no* tiene por qué coincidir con el nombre de la clase.
- Un fichero podrá incluir varias clases
- Se recomienda que el nombre sea significativo y adecuado a la funcionalidad del contenido

- **Clase**:
  - Para declarar una clase se utiliza la palabra **class**, seguido del nombre de la clase.
  - Las clases empiezan por *mayúsculas*
- **Atributos:**
  - En una clase podemos definir **atributos**. En el ejemplo vemos el atributo $var, al que se asigna un valor predeterminado.
  - Hay que indicar la *visibilidad* del atributo. Como PHP no es tipado, no es necesario declarar el tipo de dato.
  - Se recomienda que empiecen en minúsculas
- **Métodos:**
  - En una clase podemos definir **métodos** con la palabra **function**, indicando a la izquierda su visibilidad.
  - Como los métodos son funciones PHP, no requieren declaración de tipos, ni tampoco sus argumentos.
  - Se recomienda que empiecen en minúsculas

Instanciación y uso de objetos de una clase: new y ->

```php
<?php
1 reference | 0 implementations
class Persona {
    // Atributo
    2 references
    private $nombre = "A";
    2 references
    public $altura = "0";
    //Métodos
    1 reference | 0 overrides
    public function set_nombre($nuevo_nombre) {
        $this->nombre = $nuevo_nombre;
    }
    1 reference | 0 overrides
    public function mostrarPersona() {
        // echo $valor . "<br/>";
        echo "Nombre: " . $this->nombre . " Altura: " . $this->altura;
    }
}
$mi_amigo = new Persona;
$mi_amigo->set_nombre("Pedro");
$mi_amigo->altura = 180;
$mi_amigo->mostrarPersona();
?>
```

- Para instanciar un objeto de una clase se utiliza la palabra **new**.
- Para acceder a los atributos y a los métodos, tanto dentro como fuera de la clase se utiliza el operador **->**
- Si estoy dentro de la clase, utilizo **$this->**, si estoy fuera, utilizo **$miObjeto->**
- En el ejemplo de la izquierda, hemos declarado **público** el atributo $altura, por lo que podemos modificarlo directamente.
- En el ejemplo de la izquierda hemos definido **privado** el atributo $nombre, no se puede modificar directamente.
- Se han creado dos métodos:
  - Set_nombre: Permite modificar el nombre
  - MostrarPersona: Hace echo de los datos de la persona en la salida

Encapsulación

```php
class Persona {
  // Atributo
  2 references
  private $nombre = "A";
  2 references
  private $altura = "0";
  //Métodos
  1 reference | 0 overrides
  public function set_nombre($nuevo_nombre) {
    $this->nombre = $nuevo_nombre;
  }
  1 reference | 0 overrides
  public function set_altura($nueva_altura) {
    $this->altura = $nueva_altura;
  }
  0 references | 0 overrides
  public function get_nombre() {
    return $this.nombre;
  }
  0 references | 0 overrides
  public function get_altura() {
    return $this.altura;
  }
  1 reference | 0 overrides
  public function mostrarPersona() {
    echo "Nombre: " . $this->nombre . " Altura: " . $this->altura;
  }
}
```

- En el ejemplo anterior, nombre se había declarado como privado para que la clase controle la forma de modificarlo.
- Esto "encapsula" la información y la protege de manipulaciones no deseadas.
- En ese caso, se necesita un método para obtener o modificar ese dato. Por eso hemos añadido set_nombre.
- Se recomienda encapsular los atributos. Para ello se ponen como privados o protegidos, y se añaden los métodos para leerlos y modificarlos: Los setters y los getters
- A la izquierda hemos encapsulado la clase Persona de la página anterior, creando los getters y setters necesarios.

Constructor

```php
class Perro {
    // Atributo
    2 references
    private $nombre;
    1 reference | 0 overrides
    public function __construct ($nom) {
        $this->nombre = $nom;
    }
    1 reference | 0 overrides
    public function llama() {
        echo $this->nombre . ", ven aquí!!";
    }
}
$mascota = new Perro("Toby");
$mascota->llama();
?>
```

- Cuando hacemos new, se ejecuta el **constructor** de la clase. Si no hay ninguno definido por el usuario, se dejan todos los atributos a su valor por defecto.
- Si queremos que la clase se inicialice de una forma concreta, podremos crear un método `__construct($param,...)` que se ejecutará en el momento de hacer new.
- **__construct** podrá o no tener parámetros, pero sólo habrá una.
- En el ejemplo de la izquierda, cuando hagamos **new**, tendremos que indicar un parámetro. El constructor lo utilizará para inicializar el atributo nombre del objeto.

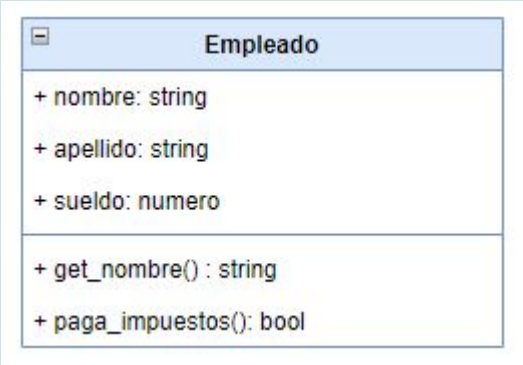
Constructores en PHP 8: Simplificación de atributos

- PHP 8 ha añadido la **simplificación de la declaración de los atributos**.
- Si vamos a inicializar un atributo en el constructor, no es necesario declararlo antes, queda declarado automáticamente.
- De esta forma, si tenemos muchos parámetros podemos no escribirlos dos veces.
- En el ejemplo de abajo, vemos cómo quedaría el ejemplo de la página anterior con simplificación de atributos

```php
class Perro {
    // private $nombre;
    1 reference | 0 overrides
    public function __construct ($nom) {
      $this->nombre = $nom; // Al estar en __construct, nombre es un atributo
    }
    1 reference | 0 overrides
    public function llama() {
      echo $this->nombre . ", ven aquí!!";
    }
}
$mascota = new Perro("Toby");
$mascota->llama();
?>
```

Ejercicio 1:

Encapsular



Boceto original: A modificar

1. Crea una clase empleado con su nombre, apellidos y sueldo, que se iniciarán en el constructor.
2. Encapsula las propiedades (atributos), mediante los getters-setters correspondientes.
3. Añade métodos para:
   - Obtener su nombre completo en un string
   - Que devuelva un booleano para ver si tiene que pagar impuestos o no. Los impuestos se pagan si su sueldo es superior a 3333€.

Clases estáticas

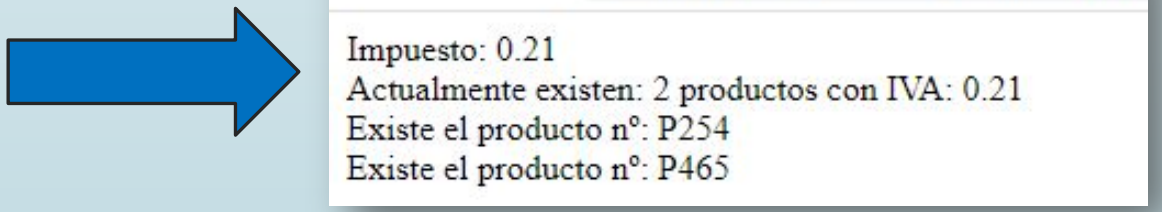
- Podemos definir **atributos estáticos**. Estos valores son compartidos por todos los objetos de una clase, y si son públicos, puede accederse desde todo el fichero. Son variables modificables. Se definen con la palabra **static**, y se accede a ellos con el operador **::**
- Podemos definir **atributos constantes**. Son compartidos por todos los objetos, y se ven en todo el fichero, pero no pueden modificarse. Se definen con la palabra **const**, y se accede a ellos con el operador **::**
- Podemos definir **métodos estáticos**, accesibles desde todo el fichero. Para acceder a ellos *no es necesario instanciar un objeto*. Se declaran con la palabra **static** y se accede a ellos con el operador **::**
- Para acceder a un método o a una variable estática desde dentro de la clase, utilizamos **self** en vez de this.
- Una clase que tenga atributos y/o métodos estáticos, se llama **clase estática**

Clases estáticas: Ejemplo

```php
class Producto {
  4 references
  public const IVA = 0.21; // Iva a aplicar
  4 references
  private static $numProductos=0; // Productos creados
  8 references
  private $codigo; // Código de un  producto
  6 references | 1 override
  public function __construct ($cod) {
    self::$numProductos++;
    $this->codigo = $cod;
  }
  1 reference | 0 overrides
  public static function muestra_numproductos() {
    echo "Actualmente existen: " . self::$numProductos
    . " productos con IVA: ". self::IVA . "<br/>";
  }
  2 references | 0 overrides
  public function muestra_codigo() {
    echo "Existe el producto nº: " . $this->codigo . "<br/>";
  }
}
echo "Impuesto: " . Producto::IVA . "<br/>";
$producto1 = new Producto ("P254");
$producto2 = new Producto("P465");
Producto::muestra_numproductos();
$producto1->muestra_codigo();
$producto2->muestra_codigo();
```

En el ejemplo:

- IVA es una constante pública. La referencio desde fuera de la clase con **P**roducto**::**IVA
- $numProductos es un atributo estático. Cada vez que instancie un objeto de la clase Producto, se ejecuta el constructor, y se incrementa ese atributo.
- $codigo es un atributo normal, privado y accesible sólo desde la clase. Cada objeto tendrá el suyo
- **self** hace referencia a la clase, mientras que **this** hacía referencia al objeto instanciado. Como el atributo es estático, con self accedemos a él desde cualquier parte de la clase.
- muestra_numproductos es una función estática, podemos acceder desde fuera (es pública) sin instanciar objeto con **P**roducto**::**muestra_numproductos



Ejercicio 2:

Clases estáticas

1. En la clase del ejercicio 1 (Empleado), añade una variable estática salarioMinimo, y dos métodos estáticos para manejarla: Un getter y un setter estáticos. El valor inicial de salarioMinimo será de 1200.
2. Añade un constructor para iniciar los valores de los atributos durante la instanciación de un empleado.
3. Añade una restricción para que, al instanciar un empleado, éste tenga al menos el salario mínimo, independientemente de lo propuesto al constructor.

Funciones de PHP para trabajar con clases

- PHP ofrece varias funciones internas para trabajar con clases:
  - **instanceof**: Indica si un objeto es instancia de una clase dada
  - **get_class**: Indica cuál es la clase de un objeto dado
  - **get_declared_class**: Retorna un array con los nombres de todas las clases declaradas
  - **class_alias**: Crea un alias para una clase.
  - **class_exists**, **method_exists**, **property_exists**: Indica si existe una clase/método/propiedad indicados
  - **get_class_methods**, **get_class_vars**: Retorna un array con los métodos/propiedades de la clase indicada

Asignación de objetos y clonación

- Si asigno un objeto a otro, NO estoy duplicándolo en el nuevo, es una referencia, es decir, ambos nombres son el mismo objeto, y si cambio propiedades en uno, quedan modificadas en el otro.
- Para duplicar un objeto, es necesario utilizar la función clone:

```
$nuevoObjeto = clone($objeto);
```

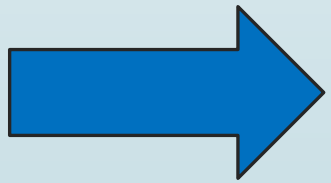
- El nuevo objeto tendrá las propiedades iguales que el primero, pero no será el mismo objeto, sería un clon (un gemelo idéntico), y con el tiempo podrían ser diferentes.
- Si queremos que al clonar un objeto haya alguna propiedad diferente, podemos utilizar en la clase un método **__clone()**, que se ejecutará al hacer el clone de un objeto. Si un objeto ha sido creado por clonación, se ejecutará primero el __construct y luego el __clone.

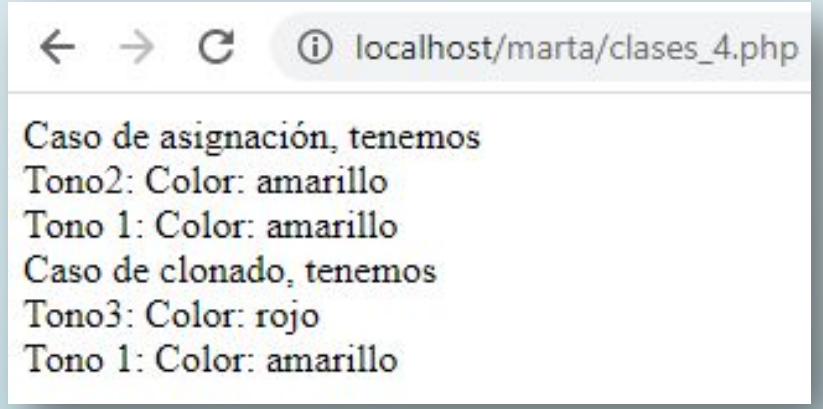
Ejemplos de asignación y clonación

```php
class Pintura {
    3 references
    private $color;
    1 reference | 0 overrides
    public function __construct($miColor) {
        $this->color = $miColor;
    }
    2 references | 0 overrides
    public function set_color($miColor) {
        $this->color = $miColor;
    }
    4 references | 0 overrides
    public function indica_color() {
        echo "Color: " . $this->color;
    }
}
$tono1 = new Pintura("Azul");
$tono2 = $tono1;
$tono2->set_color("amarillo");
echo "Caso de asignación, tenemos";
echo "<br>Tono2: ";$tono2->indica_color();
echo "<br>Tono 1: ";$tono1->indica_color();
$tono3 = clone($tono1);
$tono3->set_color("rojo");
echo "<br/>Caso de clonado, tenemos";
echo "<br>Tono3: ";$tono3->indica_color();
echo "<br>Tono 1: ";$tono1->indica_color();
```

En el ejemplo vamos a intentar *crear tres objetos*, con new, con asignación, y con clonado:

- $tono1 es un objeto creado en color azul con **new**
- **Asigno** a $tono2 el objeto $tono1. Ahora, ambos son el mismo objeto!!. Es como si creamos un alias.
- Si cambio el color de $tono2 a amarillo, $tono1 también es amarillo (pues es el mismo objeto!!)
- Creo un nuevo objeto $tono3, mediante el **clonado** de $tono1.
- $tono3 es un nuevo objeto!! Al principio sus propiedades coinciden con las de $tono1.
- Si cambio el color de $tono3 a rojo, cambia $tono3, pero no cambia $tono1, puesto que son objetos diferentes.





Herencia

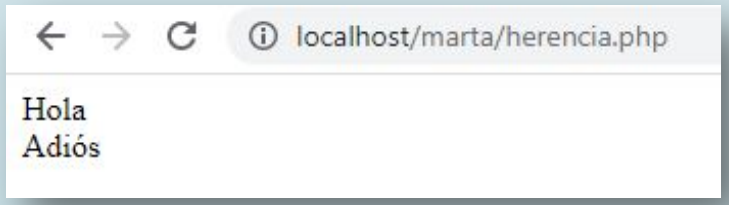
- Mediante una relación de **herencia**, disponible en los lenguajes orientados a objetos, podemos reutilizar una clase existente y heredar todas las características de la clase existente y añadir algunas nuevas.
- Es una forma de reutilización de código: Declaramos en la clase madre unas características, y las reutilizamos en clases que hereden de la misma.
- En la relación de herencia tenemos:
  - **Clase madre**: Define una serie de características (atributos, métodos,...)
  - **Clase hija** o **subclase**: Hereda las características de la madre, y añade algunas nuevas
- Las subclases son una versión *especializada* de la clase madre
- PHP permite **herencia simple** (una clase sólo podrá heredar de otra, no de varias)
- En PHP la relación de herencia se indica con la palabra **extends** (extiende, es decir, amplía)
- Los atributos y métodos de la clase madre serán visibles en la clase hija sólo si son públicos o protegidos. Los privados no pueden verse desde la clase hija.

Herencia: Ejemplo

```php
1 reference | 1 implementation
class Saludo {
    5 references
    protected $idioma;
    0 references | 0 overrides
    public function __construct($idioma) {
        $this->idioma = $idioma;
    }
    1 reference | 0 overrides
    public function saluda() {
        if ($this->idioma == "en") echo "Hello";
        else if ($this->idioma == "fr") echo "Bon jour";
        else echo "Hola";
    }
}
1 reference | 0 implementations
class Social extends Saludo {
    1 reference | 0 overrides
    public function despide() {
        if ($this->idioma == "en") echo "Good Bye";
        else if ($this->idioma == "fr") echo "Au revoir";
        else echo "Adiós";
    }
}
$amable = new Social("es");
echo $amable->saluda() . "<br/>";
echo $amable->despide() . "<br/>";
```

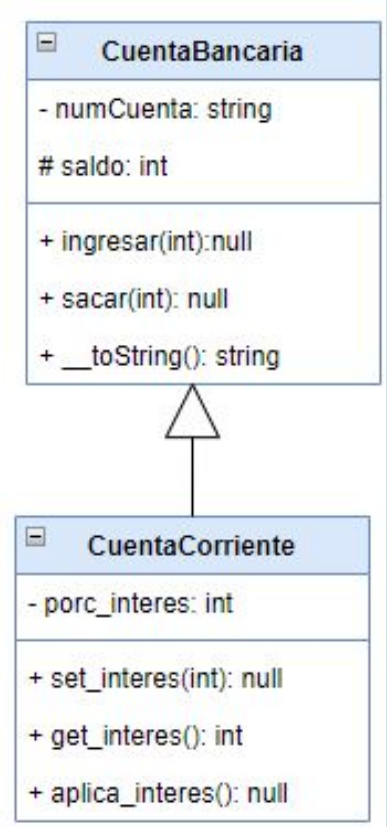
En el ejemplo hemos definido la clase Social como subclase (heredera) de la clase Saludo.

- En la clase madre Saludo, tenemos:
  - El atributo protegido $idioma, visible en la subclase
  - El método público saluda, visible en la subclase
- En la subclase Social, tenemos:
  - Un nuevo método definido, despide, que es público
- En el código definimos el objeto $amable con new Social:
  - La clase Social hereda el constructor de la clase Saludo. El objeto de tipo Social se crea con ese constructor.
  - La clase Social hereda el atributo $idioma. El objeto de tipo Social se crea con ese atributo, en este caso igualado a "es"
- En el código hemos utilizado dos métodos del objeto $amable, de tipo Social:
  - El método saluda, heredado de la clase madre Saludo
  - El método despide, añadido en la subclase Social



Ejercicio 3:

Herencia



1. Crea una clase CuentaBancaria con
   - Dos atributos (el número de cuenta privado, y el saldo protegido), iniciados en el constructor
   - Dos métodos públicos: ingresar dinero y sacar dinero.
   - Un método __toString que retornará un string con la información de la clase. Este método es invocado por PHP cuando intentemos convertir un objeto en string (por ejemplo al ejecutar la sentencia "echo $objeto")
2. Crea otra clase CuentaCorriente, que herede de CuentaBancaria.
   - Tendrá un atributo privado adicional, el porcentaje de interés, con sus getters y setters.
   - También tendrá un método público adicional, aplica_interes, sin parámetros, que calculará el interés y lo ingresará en la cuenta
3. Instancia una cuenta corriente y prueba todos los métodos de ambas clases

Herencia: Sobreescribir métodos y constructores

- La relación de herencia se utiliza para ampliar nuevos atributos y métodos
- También podemos utilizarla para modificar un método existente en la clase madre, substituyendo el código en la clase hija. A esto se le llama **sobreescribir** el método.
- Para sobreescribir un método, basta con escribir un método en la clase hija, que se llame igual que el método en la clase madre. En este caso, en vez de ejecutarse el código de la madre, se ejecutará el código de la hija.
- Es habitual que al sobreescribir un método, queramos ejecutar también el código de la madre. Para ello es necesario indicar que ejecute el de la madre dentro del método que sobreescribe con: **parent::nombreMetodo()**
- Si no se define un constructor en la clase hija, por defecto se ejecuta el constructor de la madre.
- Si escribimos un constructor para la clase hija, y queremos también que se ejecute el de la madre, tendremos que invocar al constructor de la madre con: **parent::__construct()**
- Podemos sobreescribir la conversión a string en PHP, mediante el método __toString, que retornará un string con el contenido que nos interese.

Herencia: Ejemplo de sobreescritura de métodos

```php
class Producto {
    2 references
    public string $codigo;
    3 references | 1 override
    public function __construct($codigo) {
        $this->codigo = $codigo;
    }
    2 references | 1 override
    public function mostrarResumen() {
        echo "<p>Prod:" .$this->codigo."</p>";
    }
}
1 reference | 0 implementations
class Tele extends Producto {
    2 references
    public $pulgadas;
    2 references
    public $tecnologia;
    2 references | 0 overrides | prototype
    public function __construct($codigo,$pulgadas,$tecnologia) {
        parent::__construct($codigo);
        $this->pulgadas = $pulgadas;
        $this->tecnologia=$tecnologia;
    }
    2 references | 0 overrides | prototype
    public function mostrarResumen() {
        parent::mostrarResumen();
        echo "<p>TV: " . $this->tecnologia." de ".$this->pulgadas." pulgadas</p>";
    }
}
$miTele = new Tele("232342423",27,"Plasma");
$miTele->mostrarResumen();
```

En el ejemplo hemos definido la clase Producto y una subclase Tele.

- Producto tiene el atributo $codigo, un constructor y el método mostrarResumen
- Tele extiende la clase Producto(hereda lo anterior), con dos atributos adicionales y, además:
  - Sobreescribe el constructor de Producto
  - Sobreescribe el método mostrarResumen
- Como se quiere reutilizar el código escrito en Producto para ambos métodos sobreescritos:
  - En el nuevo constructor, se pide que se ejecute también el código del constructor de la clase madre Producto con parent::__construct
  - En el nuevo método mostrarResumen, se pide que se ejecute también el código del método mostrarResumen de la clase madre Producto con parent::mostrarResumen
- Finalmente, creamos un objeto miTele, instancia de la clase Tele, y se pide mostrarResumen. Como constructor y método están sobreescritos, se ejecutarán los de la hija.

Prod:232342423

TV: Plasma de 27 pulgadas

Clases abstractas

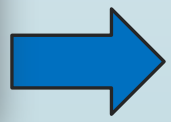
- Una clase abstracta es una especie de plantilla para otras clases, no se define para la instanciación de objetos (ni se pueden instanciar!!)
- Si defino una clase abstracta, tendré que definir otras clases no abstractas que hereden de ella, sino no podré instanciar objetos.
- Los atributos de una clase abstracta no son abstractos
- Una clase abstracta puede tener dos tipos de métodos:
  - Abstractos: Deberán, obligatoriamente, ser implementados por la clase heredera. La declaración de un método abstracto sólo admite nombre y parámetros, no puede escribirse código.
  - No abstractos: Podrán o no ser sobreescritos por la clase heredera.
- Podemos definir una **clase abstracta** como aquella que tiene **al menos** un método abstracto.
- Si intentamos instanciar un objeto de una clase abstracta, PHP nos dará un error fatal, indicando que no se puede instanciar una clase abstracta.

Clases abstractas: Ejemplo

```php
abstract class Producto {
    4 references
    private $codigo;
    3 references | 1 override
    public function __construct($codigo) {
        $this->codigo = $codigo;
    }
    1 reference | 0 overrides
    public function getCodigo() {
        return $this->codigo;
    }
    3 references | 2 overrides
    abstract public function mostrarResumen();
}
2 references | 0 implementations
class Tele extends Producto {
    4 references
    public $pulgadas;
    4 references
    public $tecnologia;
    3 references | 0 overrides | prototype
    public function mostrarResumen() {
        echo "Código: " . $this->getCodigo() . "<br/>";
        echo "TV: ". $this->tecnologia . " de ".  $this->pulgadas . " pulgadas";
    }
}
$miTele = new Tele("543553435");
$miTele->pulgadas = "35";
$miTele->tecnologia = "Plasma";
$miTele->mostrarResumen();
```

En el ejemplo hemos definido la clase Producto como abstracta, su método mostrarResumen, abstracto, deberá implementarse obligatoriamente en subclases.

- Los métodos no abstractos getCodigo y __construct se heredan
- El atributo privado $codigo se hereda, pero no es visible. La subclase tendrá que manipularlo mediante getCodigo.
- El método abstracto mostrarResumen deberá escribirse obligatoriamente en la subclase Tele, y así se ha hecho.
- Al instanciar Tele, el objeto miTele se crea con el constructor (no abstracto) definido en Producto y heredado por Tele.
- El objeto miTele utiliza el método mostrarResumen escrito en la clase Tele, que sobreescribe el método abstracto de la clase Producto.



Prod:232342423

TV: Plasma de 27 pulgadas

Clases y métodos finales

```php
class Producto {
    6 references
    private $codigo;

    1 reference | 0 overrides
    public function getCodigo() : string {
        return $this->codigo;
    }

    3 references
    final public function mostrarResumen() : string {
        return "Producto ".$this->codigo;
    }
}

// Nadie puede heredar de Microondas
0 references
final class Microondas extends Producto {
    1 reference
    private $potencia;

    0 references
    public function getPotencia() : int {
        return $this->potencia;
    }

    // No se puede implementar mostrarResumen()
}
```

- Una **clase final** es lo contrario de una abstracta. Ninguna clase puede heredar de una clase final. Son hijos sin posibilidad de descendencia.
- Un **método final** es aquel que no puede ser sobreescrito en las subclases.
- Las clases finales están pensadas para instanciación de objetos, y deben estar completamente definidas.
- Sus métodos no podrán ser sobreescritos ya que no pueden tener subclases para hacerlo.
- Se definen indicando la palabra **final** antes de la declaración.
- En el ejemplo, la clase Producto no es final, y Microondas es una subclase suya.
- En el ejemplo, Microondas no puede sobreescribir el método mostrarResumen porque es final en la clase madre.
- En el ejemplo, la clase Microondas no puede ser extendida (tener subclases), porque es final.

Interfaces en PHP

- **Un interfaz** es parecido a una clase abstracta, con las particularidades:
  - Sólo declara métodos vacíos (nombre y parámetros) con el objetivo de que las subclases los implementen. Todos los métodos de un interfaz son abstractos y públicos, aunque no se pone abstract delante.
  - No puede tener atributos, pero sí constantes
  - Un interfaz puede heredar de otro u otros interfaces, mientras no escriba el código de ningún método. Para ello se utiliza extends inter1, inter2, … Sería una especie de herencia múltiple de constantes y métodos.

```
interface Interfaz1 extends InterfazA, InterfazB, InterfazC
```

  - En el ejemplo anterior, interfaz1 recopilará todos los métodos declarados en él mismo y en los otros tres que extiende.
- Es necesario definir una clase para implementar el código de los métodos de un interfaz. Las clases que implementen un interfaz, obligatoriamente tendrán que escribir sus métodos.
- **Una clase**:
  - Sólo puede heredar con **extends** de otra clase, y sólo de una:

```
class MiClase extends OtraClase
```

  - Puede implementar los métodos de uno o varios interfaces. Se indica con la palabra **implements**

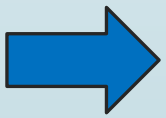
```
class MiClase implements Interfaz1, Interfaz2, Interfaz3
```

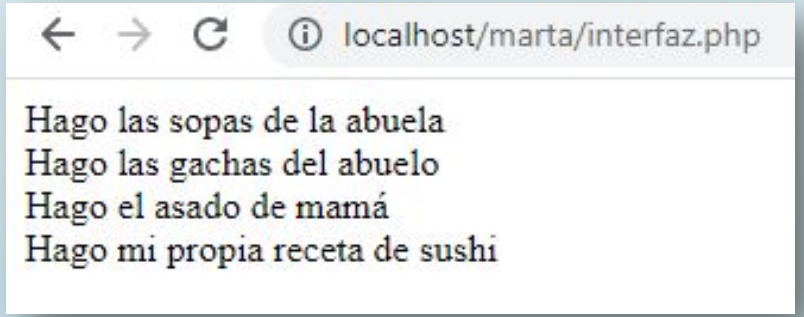
Interfaces: Ejemplo

```php
<?php
interface I_abuelo
{
    public function gachas();
}
interface I_abuela {
  public function sopas();
}
interface I_madre extends I_abuelo, I_abuela {
  public function asado();
}
class C_hija implements I_madre {
  public function gachas() {
    echo "Hago las gachas del abuelo<br/>";
  }
  public function sopas() {
    echo "Hago las sopas de la abuela</br>";
  }
  public function asado() {
    echo "Hago el asado de mamá<br/>";
  }
  public function sushi() {
    echo "Hago mi propia receta de sushi<br/>";
  }
}
$marta = new C_hija;
$marta->sopas();
$marta->gachas();
$marta->asado();
$marta->sushi();
?>
```

En el ejemplo hemos definido tres interfaces y una clase:

- Los **interfaces** I_abuela e I_abuelo declaran los métodos sopas y gachas.
- El **interfaz** I_madre, hereda (con **extends**) esas declaraciones de los anteriores (sopas y gachas), y añade otra nueva (asado)
- La **clase** C_hija implementa (con **implements**) el interfaz I_madre. **Obligatoriamente** tiene que implementar el código de las tres funciones del interfaz a implementar (sopas, gachas y asado). Además, podrá añadir nuevas funciones (caso de sushi).





Métodos mágicos

- Todas las clases de PHP, ofrecen la posibilidad de sobreescribir ciertos comportamientos del lenguaje. Estos comportamientos se denominan métodos mágicos.
- En el manual de PHP pueden verse todos los métodos mágicos. Todos los métodos mágicos empiezan por **__**
- Citaremos aquí los siguientes:
  - **__construct**: Si sobreescribimos este método, cuando hacemos new, PHP crea el objeto y ejecuta el __construct que hemos sobreescrito.
  - **__destruct**: Si termina el programa, PHP llama al recolector de basura y destruye los objetos. Si hemos sobreescrito __destruct, ejecutará el código incluido antes de eliminar el objeto.
  - **__toString**: Si PHP va a convertir en string algún objeto, que es lo que sucede cuando hacemos echo del objeto, se ejecutará el código del `__toString`. Esta función debe retornar un string para que no de error.

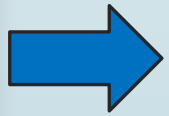
Métodos mágicos: Ejemplo de __construct y __toString

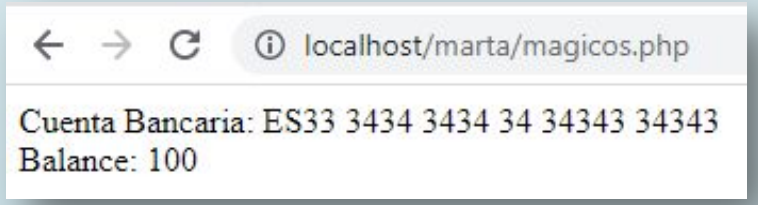
```php
<?php
1 reference | 0 implementations
class CuentaBancaria {
    2 references
    private $numCuenta;
    2 references
    private $balance;
    1 reference | 0 overrides
    public function __construct ($cuenta, $balance) {
        $this -> numCuenta = $cuenta;
        $this -> balance = $balance;
    }

    0 references | 0 overrides
    public function __toString() {
        return "Cuenta Bancaria: " . $this->numCuenta
                . "<br> Balance: " . $this->balance . "<br/>";
    }
}
$miCuenta = new CuentaBancaria("ES33 3434 3434 34 34343 34343","100");
echo $miCuenta;
?>
```

En el ejemplo, la clase CuentaBancaria tiene dos atributos privados, y dos métodos mágicos, que sobreescriben comportamiento de pHP:

- **__construct:** El sobreescrito pide dos parámetros, nº de cuenta y balance.
- **__toString**: Convierte la clase en el string del return cuando PHP quiera hacer la conversión del objeto a string
- Al crear $miCuenta, se pasan los parámetros indicados en el __construct
- Al hacer echo de $miCuenta, PHP tiene que convertir el objeto a string para poder mostrarlo. Como hemos sobreescrito el __toString, se convierte a un string como el que ponemos en el return.





Métodos mágicos: Ejemplos de creación y destrucción

```php
<?php
2 references | 0 implementations
class UnaClase {
    5 references
    public $nombre;

    2 references | 0 overrides
    public function __construct($nombre) {
        $this->nombre = $nombre;
        echo "<p>Se ejecuta el __construct de " . $this->nombre . "</p>";
    }
    0 references | 0 overrides
    public function __destruct (){
        echo "<p>Se ejecuta el __destruct de " . $this->nombre . "</p>";
    }
    0 references | 0 overrides
    public function __toString (){
        echo "<p>Se ejecuta el __toString de " . $this->nombre . "</p>";
        return "<p>Mi nombre es: " . $this->nombre . "</p>";
    }

}
$marta = new UnaClase("Marta"); // Se ejecuta el __construct
echo $marta; // Se ejecuta el __toString, pues echo requiere un string y PHP convierte
unset($marta); // PHP elimina el objeto
$pedro = new UnaClase("pedro"); // Se ejecuta el __construct
?>
```

En este ejemplo utilizo los métodos mágicos que complementan las acciones de PHP:
1.- Al hacer new, se ejecuta __construct
2.- Al hacer echo $objeto, PHP intenta convertir $objeto a string, por tanto se ejecuta __toString
3.- Al hacer unset($objeto), se ejecuta el __destruct
4.- Al terminar de ejecutar el script, PHP destruye los objetos con el recolector de basura (garbage collector), y cuando le llega el turno a los objetos existentes de tipo UnaClase, ejecutará el __destruct para cada objeto a destruir.
-- $marta ya estaba destruido con el unset, por tanto no existe al finalizar el script
-- $pedro existe al finalizar el script, por tanto PHP lo destruye, y como hay un __destruct, lo ejecuta.

Funciones PHP: Si/No orientadas a objetos

- Antes de PHP 5, no existía la orientación a objetos en este lenguaje. Por eso, muchas de las funciones internas no están orientadas a objetos. Se definían con un nombre doble, para organizar por funcionalidad.
- Por ejemplo, si miramos en el manual las funciones de fecha, vemos que todas empiezan por la palabra date (fecha en inglés), y luego "_", y finalmente el nombre de la función.

  https://www.php.net/manual/en/ref.datetime.php

- Como buena práctica de programación, los diseñadores de PHP pusieron un prefijo a todos los nombres para organizar las funciones en grupos: date_, string_, array_, etc
- Esto que hemos visto es una organización del espacio de nombres de las librerías.
- Con la POO nos ahorramos eso, ya que englobamos en una caja (la Clase) todas las funciones relativas a un mismo espacio de nombres.
- Por ejemplo, en la clase DateTime de PHP, ya conviven todas las funciones de fecha, por lo que no hace falta poner date_. Al estar en un objeto de ese tipo se sobreentiende que actúan sobre él. Así, si instancio un objeto "miFecha", para sumar bastaría poner miFecha.add(), no tendríamos que indicar miFecha.fecha_add(), ya sé que es una fecha!!

  https://www.php.net/manual/en/class.datetime.php

Funciones PHP: Ejemplo con/sin orientación a objetos

```php
<?php
  // Sin orientación a objetos
  echo "<h3>No orientado a objetos</h3>";
  date_default_timezone_set('Europe/Madrid'); // Ver manual https://www.php.net/

  $proximaSemana = time() + (7 * 24 * 60 * 60 ); // Datos en segundos
  echo "Ahora: ". date("d-m-Y") . "<br/>";
  echo "Próxima semana: " . date("d-m-Y", $proximaSemana) . "<br/>";

  // Con orientación a objetos
  echo "<h3>Orientado a objetos</h3>";
  $ahora = new DateTime(); // Por defecto pone now: Día y hora de creación
  $proximaSemana = new DateTime(); // Pone now
  $proximaSemana->add(new DateInterval("P7D")); // Le sumamos una semana
  echo "Ahora: ". $ahora->format("d-m-Y") . "<br/>";
  echo "Próxima semana: " . $proximaSemana->format("d-m-Y") . "<br/>";
?>
```

Ejercicio 4:

Clases vs funciones.

Fechas en PHP

1. Accede al manual de PHP e investiga cómo manejar una fecha y hora en este lenguaje:

   https://www.php.net/manual/es/book.datetime.php

2. Responde:
   1. ¿Qué Clase se utiliza para implementar y manejar objetos de tipo fecha y hora?
   2. ¿Qué interfaz implementa esa clase?
   3. ¿Qué otras clases implementan ese interfaz?
   4. ¿Podemos manejar fechas y horas sin utilizar clases?

https://www.php.net/docs.php