



Diplomado virtual en
PROGRAMACIÓN EN PHP
Guía didáctica 3: Programación orientada a objetos [POO]



Formación Virtual

.....educación sin límites



Competencia específica

Se espera que, con los temas abordados en la guía didáctica del módulo 3: Programación orientada a objetos [POO], el estudiante logre la siguiente competencia específica:

- Conocer los conceptos básicos, componentes y características de la programación orientada a objetos en PHP.



Contenidos temáticos

El contenido temático, para desarrollar en la guía didáctica del módulo 3: Programación orientada a objetos [POO], es:

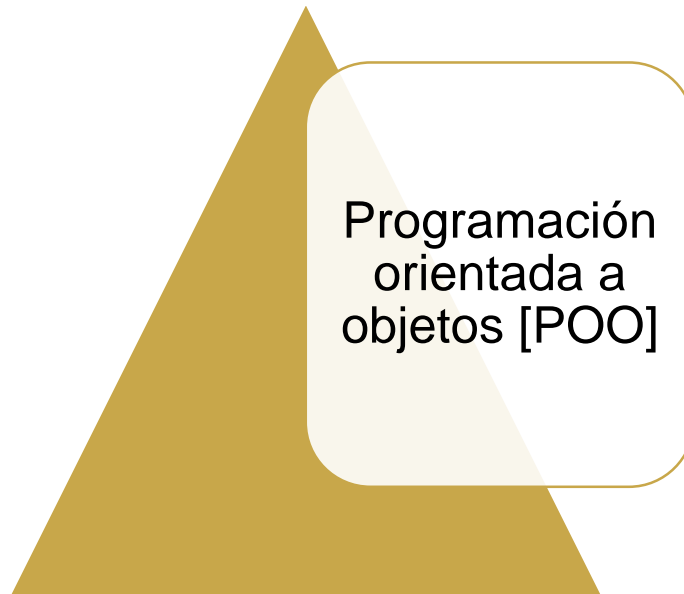


Ilustración 1: caracterización de la guía didáctica.

Fuente: autor¹.

¹ Todas las ilustraciones de esta guía son autoría propia del docente y tienen como función mostrar la aplicación y práctica del contenido que se desarrolla, por ello solo se enumerarán.

Tema 1: Programación Orientada a Objetos [POO]

La programación orientada a objetos es un paradigma de la programación, pero **¿qué es un paradigma en programación?** Es una propuesta tecnológica adoptada por una comunidad de programadores y desarrolladores, cuyo núcleo central es incuestionable en cuanto que únicamente trata de resolver uno o varios problemas claramente delimitados. Es decir, una forma que los programadores emplean para resolver problemas a partir de clases y objetos; es una forma especial de programar, más cercana a como se expresan las cosas en la vida real.

Se debe aclarar que la programación orientada a objetos, en adelante POO, no se trata de nuevas características que adquiere el lenguaje de programación. Con la POO se hace referencia a una nueva forma de pensar. Lo que se acostumbra a ver es la programación estructurada, donde se tiene un problema y se descompone en distintos subproblemas para poder llegar a tener soluciones más pequeñas y simples.

Cómo se mencionó anteriormente, la POO busca ver la programación como en la vida real. Donde los objetos son similares a los objetos en la vida real y de manera frecuente se escucharán los conceptos de «clases» y «objetos». Pero **¿qué es una clase y qué es un objeto?**

Clases

Las clases son declaraciones de objetos, también se podrían definir como abstracciones de objetos o moldes. Esto quiere decir que la definición de un objeto es una clase. Cuando se programa un objeto y se definen sus características y funcionalidades en realidad lo que se hace es programar una clase.

La definición básica de una clase comienza con la palabra reservada *class*, seguida de un nombre de clase, y continuando con un par de llaves que encierran las definiciones de las propiedades y métodos pertenecientes a dicha clase.

```

FOLDERS
  ▼ diplomado
    ► arrays
    ► ciclos
    ► condicionales
    ► holamundo
    ► operadores
    ► poo
    ► switchcase
    ► tiposdatos
    ► variables

clases.php
1  <?php
2
3  class Persona
4  {
5      $nombre = "Diego";
6      $apellido = "Palacio";
7      $edad = "22";
8      $genero = "Masculino";
9
10     public function cantar()
11     {
12         echo "Estoy cantando";
13     }
14
15     public function hablar()
16     {
17         echo "Voy hablar";
18     }
19
20     public function caminar()
21     {
22         echo "Estoy caminando";
23     }
24
25     public function bailar()
26     {
27         echo "¿Bailamos?";
28     }
29 }
30
31 ?>

```

Ilustración 2.

Las clases, en PHP, son básicamente una plantilla que sirve para crear un objeto. Si se imaginan las clases en el mundo en el que se vive, se podría decir que la clase **persona** es una plantilla sobre cómo es un humano. Juan, Andrés, Evelin y Nicol son objetos de la clase **persona**, ya que todos son personas.

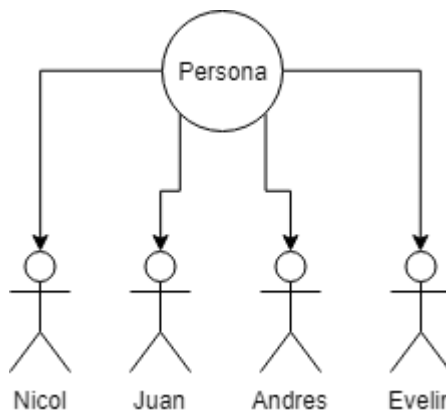
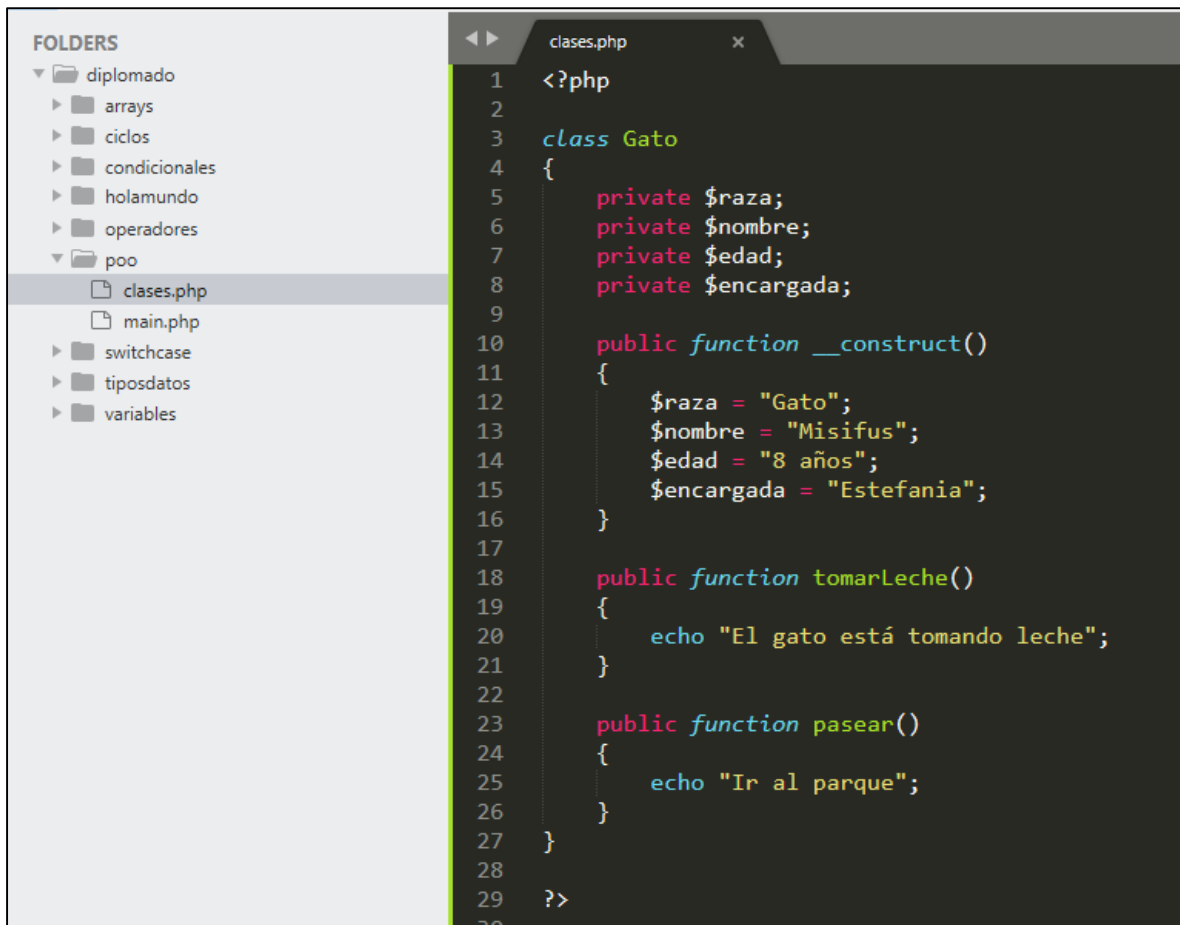


Ilustración 3.

La clase **persona** contiene la definición de un ser humano, mientras que cada ser humano es una instancia u objeto de dicha clase.

En el ejemplo de la clase **persona** hay varias características que se deben tener en cuenta al momento de trabajar con clases y programación orientada a objetos, dado que entran en juego términos como: atributos, métodos, objetos y constructores. Todos a partir de una clase.



```
1 <?php
2
3 class Gato
4 {
5     private $raza;
6     private $nombre;
7     private $edad;
8     private $encargada;
9
10    public function __construct()
11    {
12        $raza = "Gato";
13        $nombre = "Misifus";
14        $edad = "8 años";
15        $encargada = "Estefania";
16    }
17
18    public function tomarLeche()
19    {
20        echo "El gato está tomando leche";
21    }
22
23    public function pasear()
24    {
25        echo "Ir al parque";
26    }
27 }
28
29 ?>
```

Ilustración 4.

En este ejemplo, la clase **gato** está compuesta de 4 atributos: **raza**, **nombre**, **edad** y **encargada o dueña**; que son inicializados por medio del constructor de la clase **gato**, además de contener dos métodos o funciones que describen dos acciones: **tomar leche** y **pasear**.

Características de una clase

Existe una estructura fundamental para todas las clases definidas en PHP que se debe tener en cuenta al momento de construirlas:

- **Nombre:** identifica la clase de forma única en nuestro proyecto, debe ser claro y descriptivo.
- **Atributos:** referencia los campos y variables de la clase que permiten definir las características de una clase.

Los atributos hacen el papel de variables en las clases, tienen las mismas características y restricciones, pero en algunos casos cuentan con variantes y complementos.

En el ejemplo de la clase gato, los atributos que componen la clase son:

```
private $raza;  
private $nombre;  
private $edad;  
private $encargada;
```

Ilustración 5.

Las características que se recomiendan para la implementación de los atributos en Java son las siguientes:

- Una clase puede contener N atributos o ninguno.
- Se recomienda el uso de los modificadores de acceso para limitar el alcance de este, en especial *private*.
- Nombres claros en relación con la clase.
- En la mayoría de los casos no se inicializan los atributos.
- **Métodos:** los métodos en PHP permiten describir esas acciones que va a realizar la clase.

En el ejemplo de la clase gato, los métodos que describen las acciones que puede realizar esta clase son muy sencillos: tomarLeche() y ladrar().

Una clase puede contener N cantidad de métodos, con la salvedad de que deben contener nombres diferentes en su declaración, salvo en algunos casos especiales.

```
public function tomarLeche()
{
    echo "El gato está tomando leche";
}

public function pasear()
{
    echo "Ir al parque";
}
```

Ilustración 6.

- **Constructor:** el constructor es un método especial dentro de una clase, que se suele utilizar para darle valores a los atributos del objeto al crearlo.

Los constructores se crean a partir de la palabra reservada del lenguaje `__construct()`.

Es el primer método que se ejecuta al crear el objeto y se llama automáticamente al crearlo.

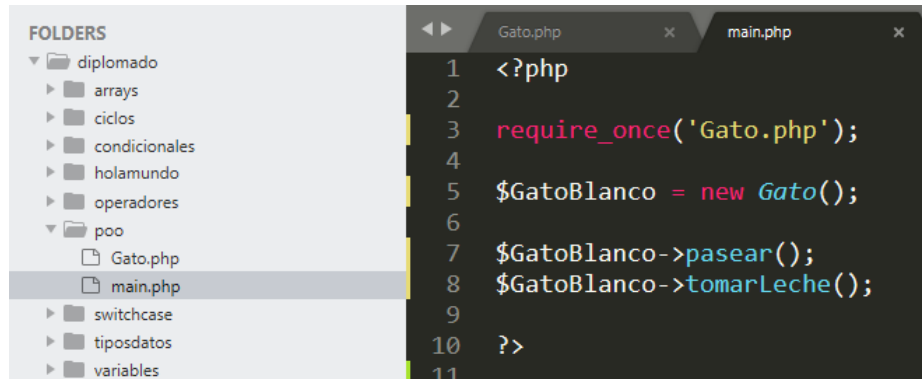
```
public function __construct()
{
    $nombre = "Misifus";
    $edad = "8 años";
    $encargada = "Estefania";
}
```

Ilustración 7.

Ahora de forma rápida y saltando algunos conceptos que serán vistos más adelante, veamos el uso de la clase gato en ejecución.

Para este ejemplo hay algunos cambios, el archivo que se llamaba clases dentro del proyecto y del directorio POO, lo vamos a cambiar a Gato.php.

Vamos a crear un nuevo archivo dentro del directorio POO, con nombre main.php, dado que las clases no se pueden ejecutar directamente dentro de sí, por lo que siempre debe haber un archivo «auxiliar» y la estructura de ese archivo será la siguiente para este caso.



Ilustraciones 8 y 9.

Getters y setters

Los *setters* y *getters* son métodos de acceso, generalmente son una interfaz pública para cambiar atributos privados de las clases, dado que cuando se determinan atributos privados no hay manera de acceder a ellos sin un método de acceso:

Setters: hace referencia a la acción de establecer, sirve para asignar un valor inicial a un atributo, pero de forma explícita, además el *setter* nunca retorna nada (siempre asigna) y solo permite dar acceso público a ciertos atributos que el usuario pueda modificar.

```
public function setNombre($nombre)
{
    $this->nombre = $nombre;
}
```

Ilustración 10.

Los métodos *setters* se componen de las siguientes características:

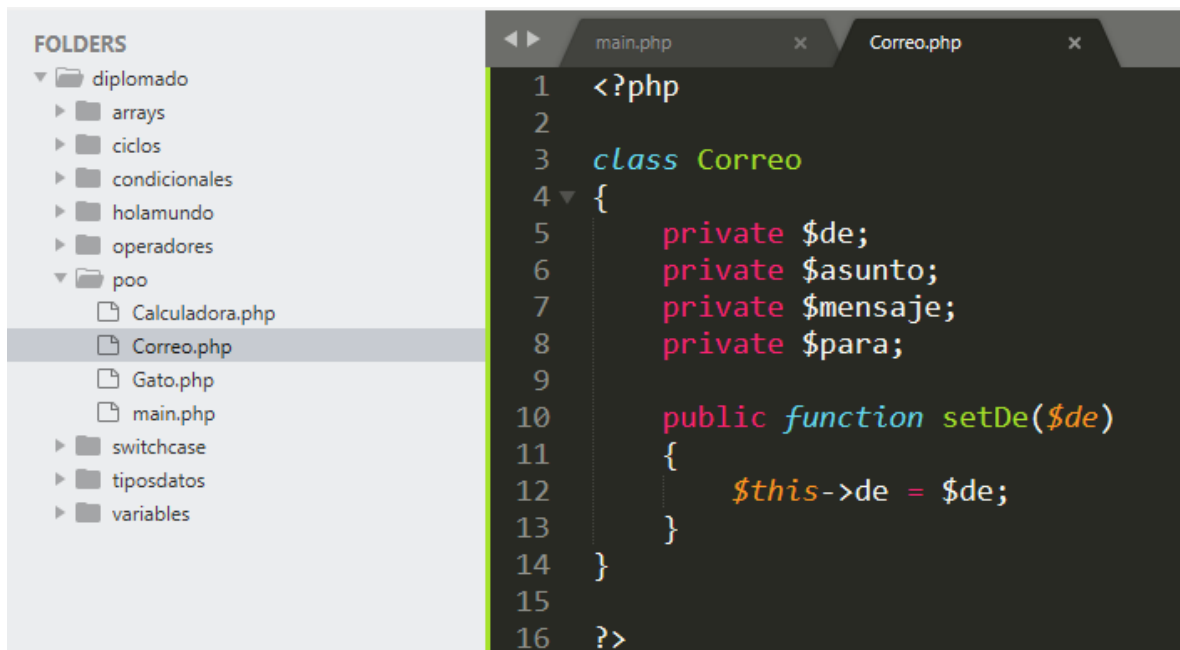
- Suelen ser públicos dado que dan acceso a los atributos privados.

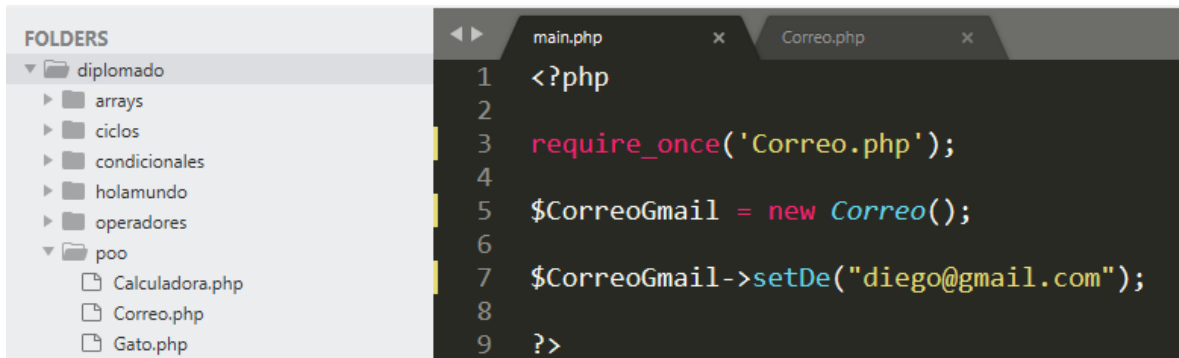
- El nombre suele estar compuesto anteponiendo la palabra *set* y el nombre del atributo.
- Contiene un parámetro que se pasa argumentado.
- La palabra *this* ayuda a diferenciar el atributo de la clase y el parámetro, dado que sin este se estaría operando únicamente con el parámetro, dejando a un lado los atributos.

```
public function setNombre($nombre)
{
    $nombre = $nombre;
}
```

Ilustración 11.

Un ejemplo de implementación sería el siguiente:





Ilustraciones 12 y13.

De esta forma el **atributo** de del **objeto** CorreoGmail toma el valor de “diego@gmail.com”.

Ahora, si ese valor se desea recuperar hay que hacer uso de get().

Getters: hace referencia a la acción de obtener, sirve para obtener (recuperar o acceder) el valor ya asignado a un atributo y ser utilizado.

```
public function getDe()  
{  
    return $this->de;  
}
```

Ilustración 14.

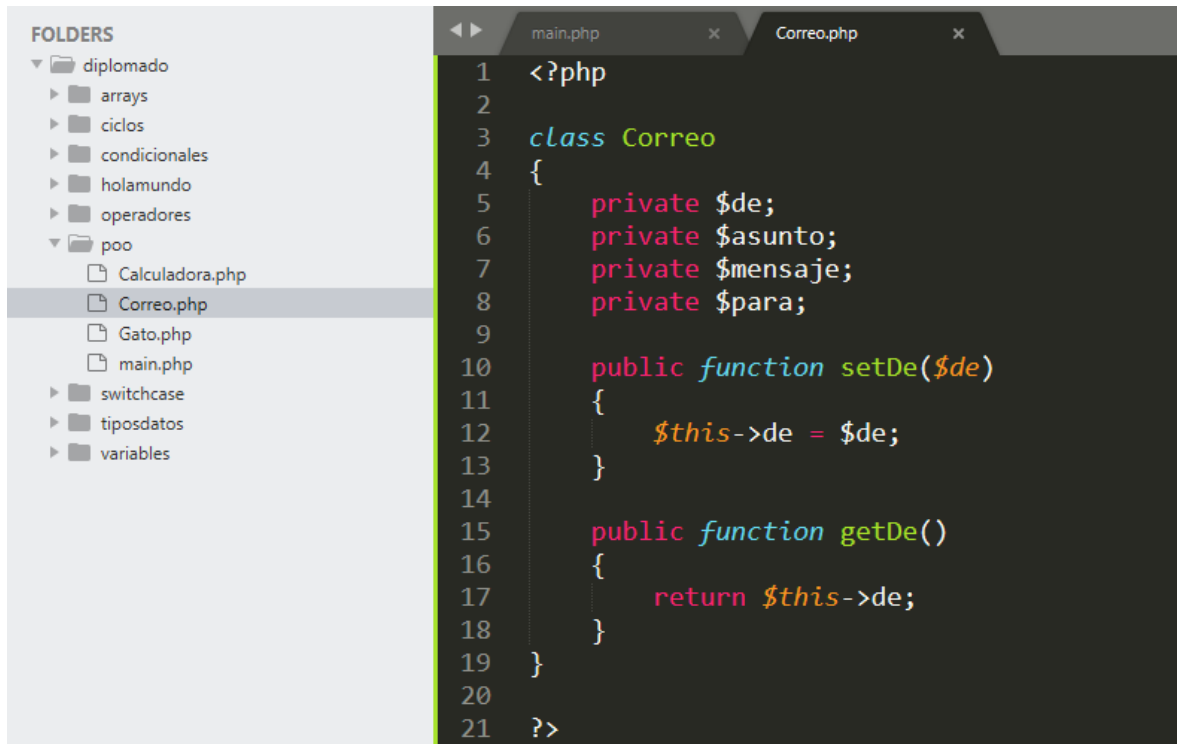
Los métodos *getters* se componen de las siguientes características:

- Suelen ser públicos, dado que dan acceso a los atributos privados.
- El nombre suele estar compuesto anteponiendo la palabra *get* y el nombre del atributo.
- No recibe parámetros dado que únicamente cumple la función de retornar valores ya establecidos en la clase.
- Retorna el valor del atributo que representa.
- Es necesario usar *this*, dado que se debe hacer referencia al atributo de la clase.
- Usa la palabra reservada **return** que, como su nombre lo dice, es retornar.

```
return $this->de;
```

Ilustración 15.

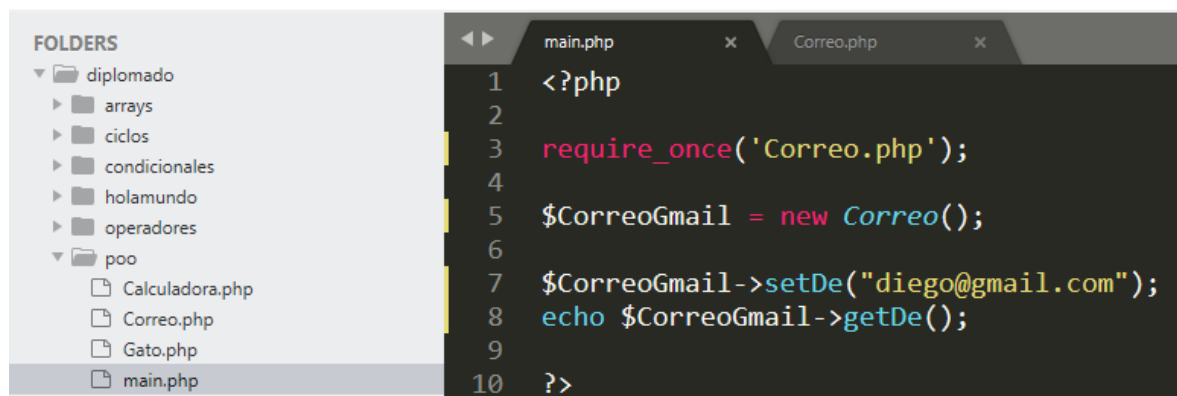
Un ejemplo de implementación sería el siguiente:



The screenshot shows a code editor with two tabs: 'main.php' and 'Correo.php'. The 'Correo.php' tab is active, displaying the following PHP code:

```
1 <?php
2
3 class Correo
4 {
5     private $de;
6     private $asunto;
7     private $mensaje;
8     private $para;
9
10    public function setDe($de)
11    {
12        $this->de = $de;
13    }
14
15    public function getDe()
16    {
17        return $this->de;
18    }
19 }
20
21 ?>
```

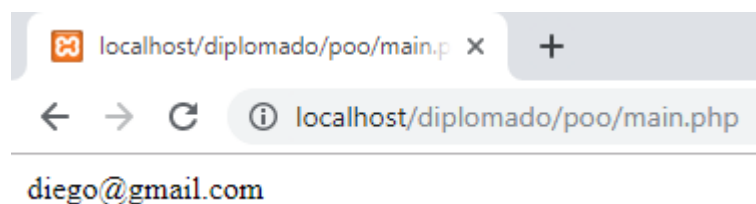
The left sidebar shows a file explorer with a folder structure: 'diplomado' (containing 'arrays', 'ciclos', 'condicionales', 'holamundo', 'operadores', and 'poo'), and 'poo' (containing 'Calculadora.php', 'Correo.php', 'Gato.php', and 'main.php'). 'Correo.php' is selected.



The screenshot shows a code editor with two tabs: 'main.php' and 'Correo.php'. The 'main.php' tab is active, displaying the following PHP code:

```
1 <?php
2
3 require_once('Correo.php');
4
5 $CorreoGmail = new Correo();
6
7 $CorreoGmail->setDe("diego@gmail.com");
8 echo $CorreoGmail->getDe();
9
10 ?>
```

The left sidebar shows the same file explorer as the previous screenshot, with 'main.php' selected.



Ilustraciones 16, 17 y 18.

De esta forma el **atributo** *de* del **objeto** CorreoGmail retornará el valor que fue seteado (set) previamente.

La clase correo tendría la siguiente estructura:

- Cuatro atributos
- Cuatro métodos *set*.
- Cuatro métodos *get*.

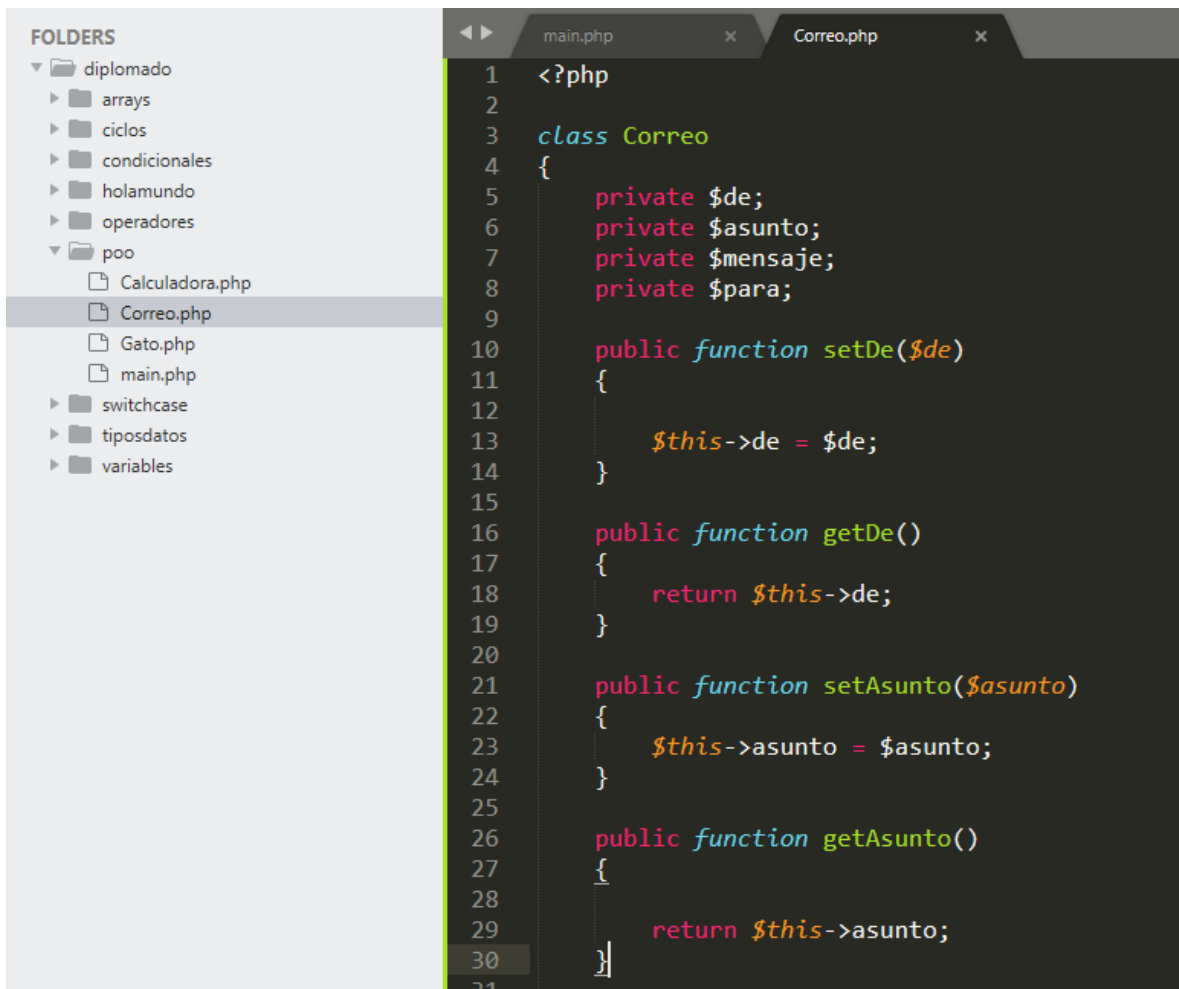
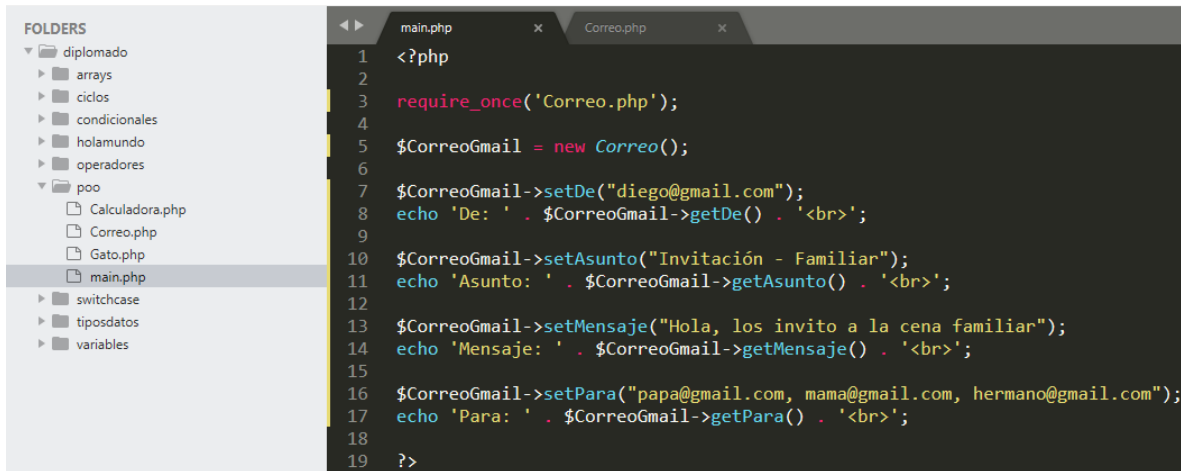


Ilustración 19.

El archivo de ejecución main.php tendría la siguiente estructura:

- El `require_once` de la clase correo.
- La creación del objeto CorreoGmail.
- La asignación de valores por medio de *set*.

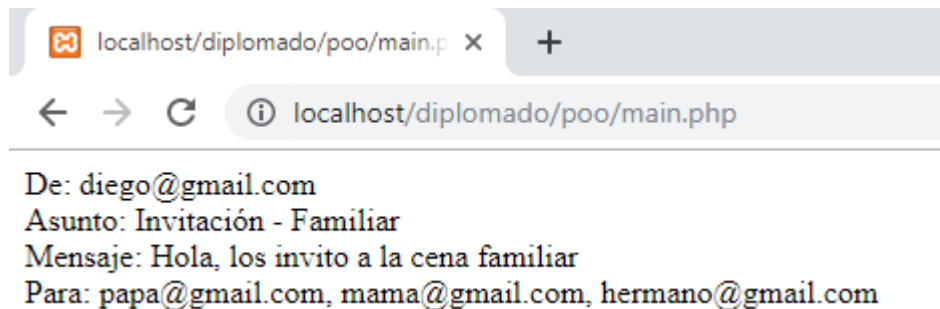
- La obtención de valores por medio de *get*.



```

1 <?php
2
3 require_once('Correo.php');
4
5 $CorreoGmail = new Correo();
6
7 $CorreoGmail->setDe("diego@gmail.com");
8 echo 'De: ' . $CorreoGmail->getDe() . '<br>';
9
10 $CorreoGmail->setAsunto("Invitación - Familiar");
11 echo 'Asunto: ' . $CorreoGmail->getAsunto() . '<br>';
12
13 $CorreoGmail->setMensaje("Hola, los invito a la cena familiar");
14 echo 'Mensaje: ' . $CorreoGmail->getMensaje() . '<br>';
15
16 $CorreoGmail->setPara("papa@gmail.com, mama@gmail.com, hermano@gmail.com");
17 echo 'Para: ' . $CorreoGmail->getPara() . '<br>';
18
19 ?>

```



Ilustraciones 20 y 21.

This - Static - Self

This

La pseudovariable *\$this* sirve para hacer referencia a un método, propiedad o atributo del objeto actual. Se utiliza principalmente cuando existe sobrecarga de nombres. La sobrecarga de nombres se da cuando hay una variable local de un método o constructor, o un parámetro formal de un método o constructor, con un nombre idéntico al que está presente en la clase al momento de relacionarse.

```
1  <?php
2
3  class Calculadora
4  {
5      private $numero1;
6      private $numero2;
7
8      public function __construct($numero1, $numero2)
9      {
10         $this->numero1 = $numero1;
11         $this->numero2 = $numero2;
12     }
```

Ilustración 22.

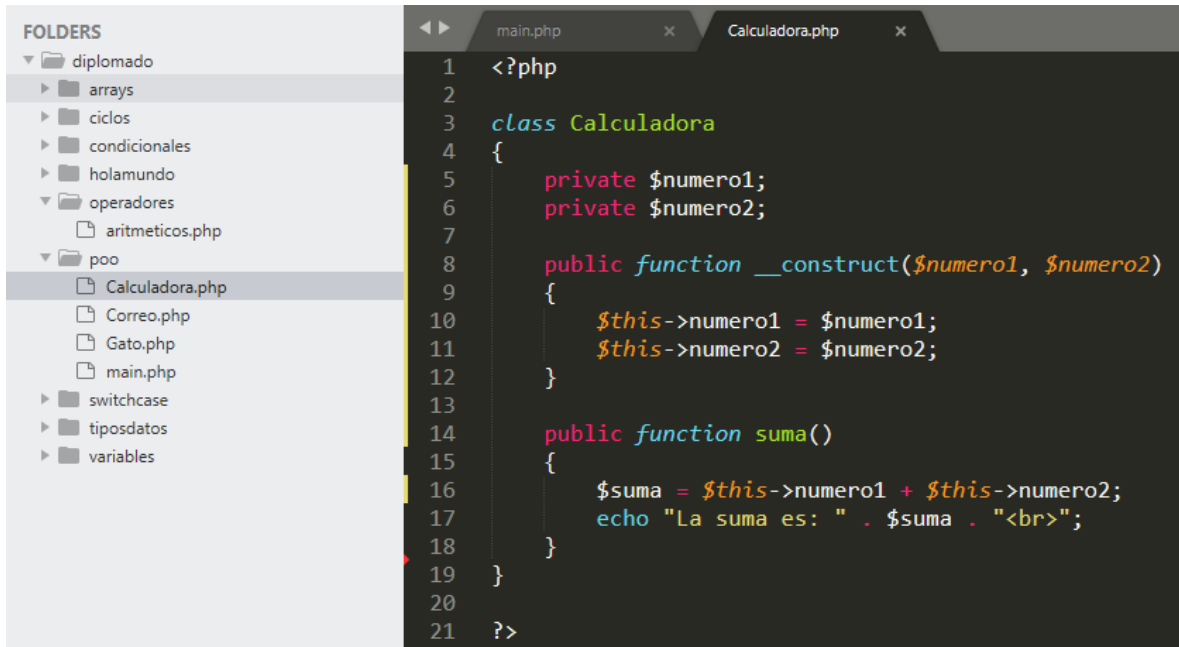
En el constructor de la clase calculadora, **this** ayuda a diferenciar los parámetros de los atributos, dado que estos contienen el mismo nombre.

Se hace uso en la clase calculadora dado que representa el objeto actual, al codificar **\$this->numero1** (sin el signo peso en el atributo), le estamos indicando a PHP que vamos a utilizar el atributo de la clase y no los parámetros del constructor.

```
1  <?php
2
3  class Calculadora
4  {
5      private $numero1;
6      private $numero2;
7
8      public function __construct($numero1, $numero2)
9      {
10         $this->numero1 = $numero1;
11         $this->numero2 = $numero2;
12     }
```

Ilustración 23.

Aquí se observa de mejor forma el funcionamiento de **this**, el color rojo representa los atributos y el color amarillo los parámetros. En ejecución el resultado sería el siguiente:



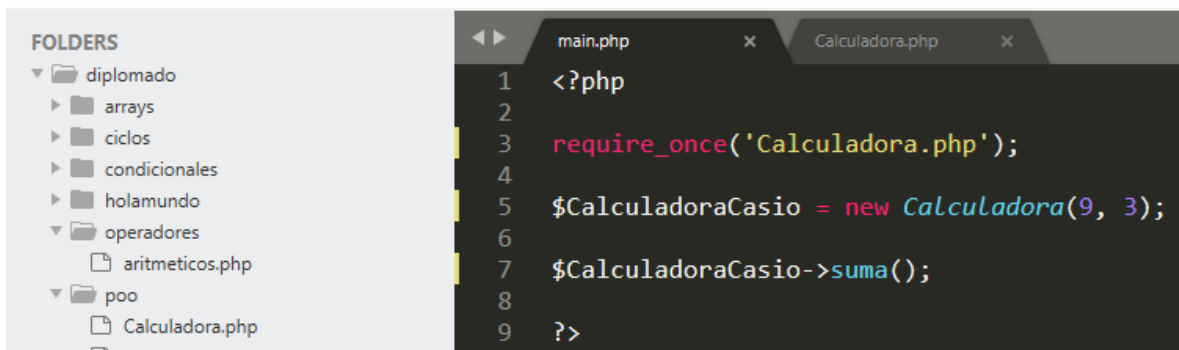
The screenshot shows a code editor with two tabs: 'main.php' and 'Calculadora.php'. The 'Calculadora.php' tab is active, displaying the following PHP code:

```

1  <?php
2
3  class Calculadora
4  {
5      private $numero1;
6      private $numero2;
7
8      public function __construct($numero1, $numero2)
9      {
10         $this->numero1 = $numero1;
11         $this->numero2 = $numero2;
12     }
13
14     public function suma()
15     {
16         $suma = $this->numero1 + $this->numero2;
17         echo "La suma es: " . $suma . "<br>";
18     }
19 }
20
21 ?>

```

The left sidebar shows a file explorer with a tree structure under 'diplomado' > 'poo', including 'Calculadora.php', 'Correo.php', 'Gato.php', and 'main.php'.



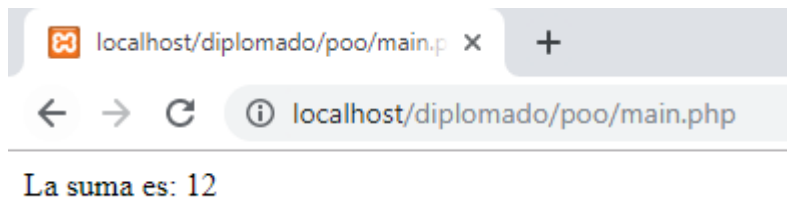
The screenshot shows the same code editor with the 'main.php' tab active. The code in 'main.php' is as follows:

```

1  <?php
2
3  require_once('Calculadora.php');
4
5  $CalculadoraCasio = new Calculadora(9, 3);
6
7  $CalculadoraCasio->suma();
8
9  ?>

```

The left sidebar shows the same file explorer structure, with 'Calculadora.php' highlighted under 'poo'.



Ilustraciones 24, 25 y 26.

En este caso por medio del constructor de la clase calculadora se inicializan los atributos por medio de *this*, que estarán disponibles en toda la clase y objeto actual.

```
public function __construct($numero1, $numero2)
{
    $this->numero1 = $numero1;
    $this->numero2 = $numero2;
}
```

Ilustración 27.

Y en el método suma, simplemente se debe realizar un llamado al mismo. Dado que internamente, gracias a *this*, el método llama a los atributos que fueron pasados como parámetros en el constructor.

```
7 $CalculadoraCasio->suma();
```

```
public function suma()
{
    $suma = $this->numero1 + $this->numero2;
    echo "La suma es: " . $suma . "<br>";
}
```

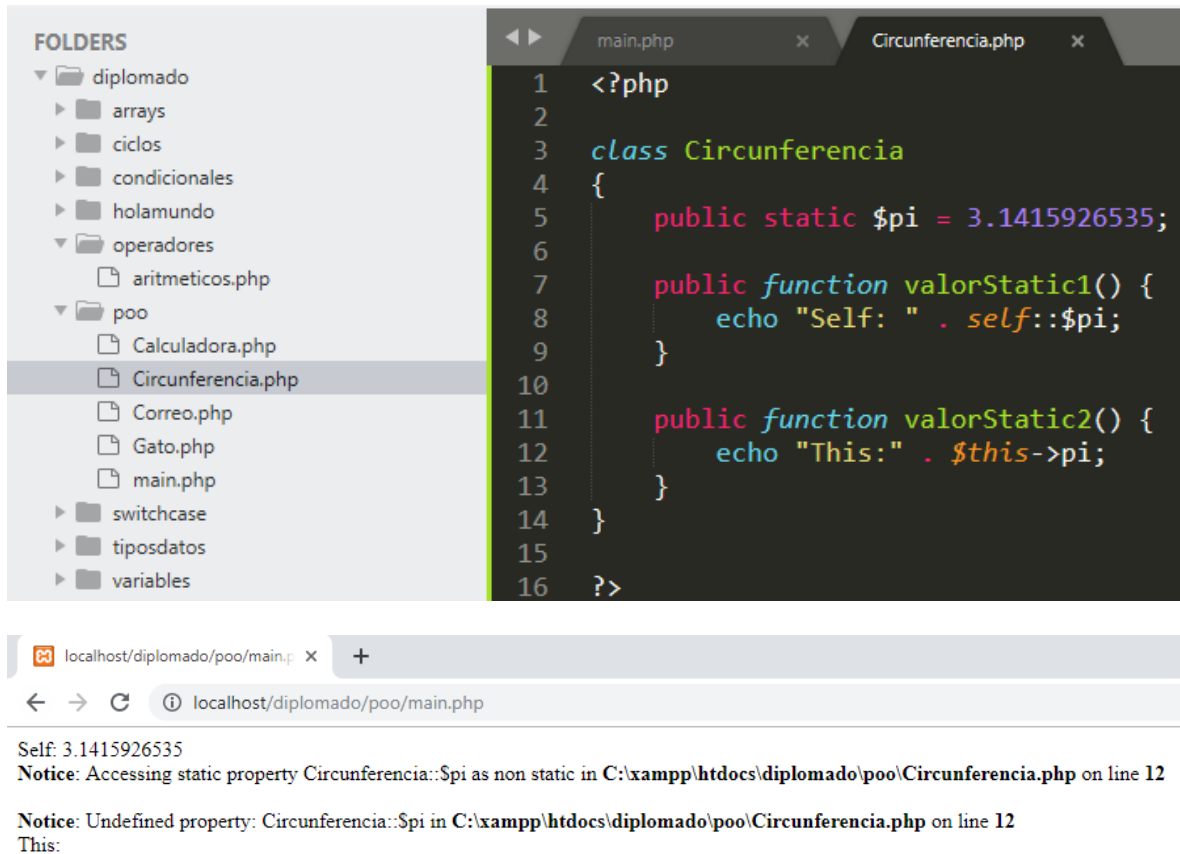
Ilustraciones 28 y 29.

Static

Los elementos estáticos (o miembros de clase) son aquellos que pertenecen a la clase, en lugar de pertenecer a un objeto en particular.

Declarar propiedades o métodos de clases como estáticos los hacen accesibles sin la necesidad de instanciar la clase. Una propiedad declarada como *static* no puede ser accedida con un objeto de clase instanciado (aunque un método estático sí lo puede hacer).

Debido a que los métodos estáticos se pueden invocar sin tener creada una instancia del objeto, la pseudovariable *\$this* no está disponible dentro de los métodos declarados como estáticos.



The screenshot shows a code editor with two tabs: 'main.php' and 'Circunferencia.php'. The 'Circunferencia.php' tab is active, displaying the following PHP code:

```
1 <?php
2
3 class Circunferencia
4 {
5     public static $pi = 3.1415926535;
6
7     public function valorStatic1() {
8         echo "Self: " . self::$pi;
9     }
10
11     public function valorStatic2() {
12         echo "This:" . $this->pi;
13     }
14 }
15
16 ?>
```

Below the code editor, a browser window shows the output of the script. The output is 'Self: 3.1415926535'. Below the output, there are two error messages:

```
Notice: Accessing static property Circunferencia::$pi as non static in C:\xampp\htdocs\diplomado\poo\Circunferencia.php on line 12
Notice: Undefined property: Circunferencia::$pi in C:\xampp\htdocs\diplomado\poo\Circunferencia.php on line 12
This:
```

Ilustraciones 30 y 31.

Como se observa en el ejemplo, el uso de **this** para acceder a propiedades estáticas está prohibido, en esos casos se hace el uso de **self**. Pero ¿qué es **self** y en qué se diferencia de **this**?

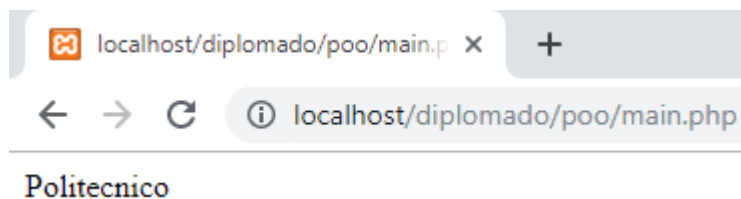
Self

En PHP, **self** y **this** se utilizan para referir miembros de clase dentro del alcance de una clase. Los miembros de la clase pueden ser variables o métodos. Estas palabras claves de PHP difieren con el comportamiento estático de los miembros de la clase.

Cuando queramos acceder a una constante o método estático o un atributo desde dentro de la clase, se usa la palabra reservada: **self**.

```
1 <?php
2
3 class Palabra
4 {
5     private static $palabra;
6
7     public function __construct($palabra)
8     {
9         self::$palabra = $palabra;
10    }
11
12    public function mostrarPalabra()
13    {
14        echo self::$palabra;
15    }
16 }
17
18 ?>
```

```
1 <?php
2
3 require_once('Palabras.php');
4
5 $Palabrita = new Palabra("Politecnico");
6 $Palabrita->mostrarPalabra();
7
8 ?>
```



Ilustraciones 32, 33 y 34.

Parámetros - Argumentos

Los parámetros o argumentos son una forma de intercambiar información con el método. Pueden servir para introducir datos para ejecutar el método (entrada) o para obtener o modificar datos tras su ejecución (salida).

- **Parámetros:** uso en la declaración del método, son los valores que un método recibe desde un objeto.

```
public function mostrarInformacion($nombre, $edad, $correo)
{
    echo "Nombre: " . $nombre . " - Edad: " . $edad . " - Correo: " . $correo;
}
```

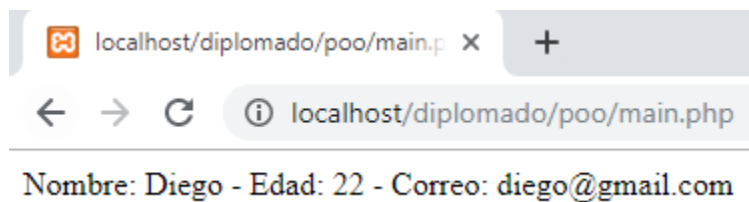
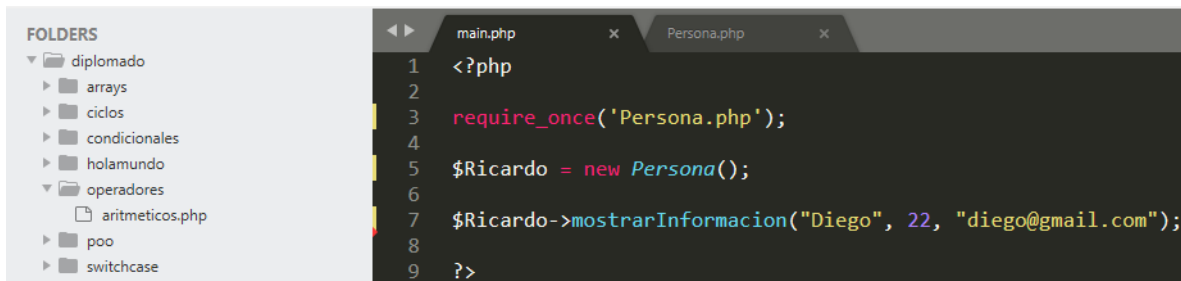
Ilustración 35.

- **Argumentos:** uso en el paso de datos a métodos desde un objeto. Son los valores que un objeto recibe para operar un método.

```
1 <?php
2
3 require_once('Persona.php');
4
5 $Ricardo = new Persona();
6
7 $Ricardo->mostrarInformacion("Diego", 22, "diego@gmail.com");
8
9 ?>
```

Ilustración 36.

Paso por valor. Son argumentos que contienen el valor exacto de estos.



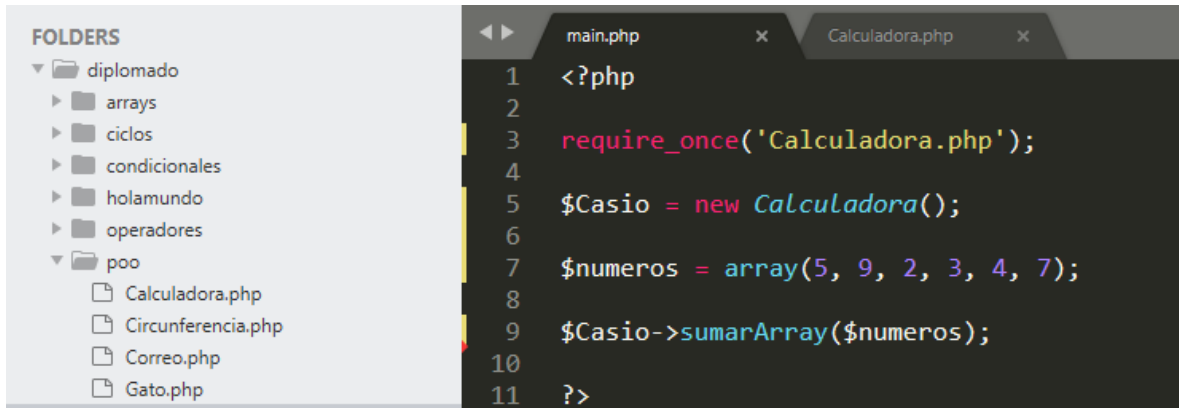
Ilustraciones 37 y 38.

El paso por valor se caracteriza por el uso de tipos de datos clásicos que permiten el paso de valores exactos a los métodos. Ejemplos:

- Números.

- Cadenas.

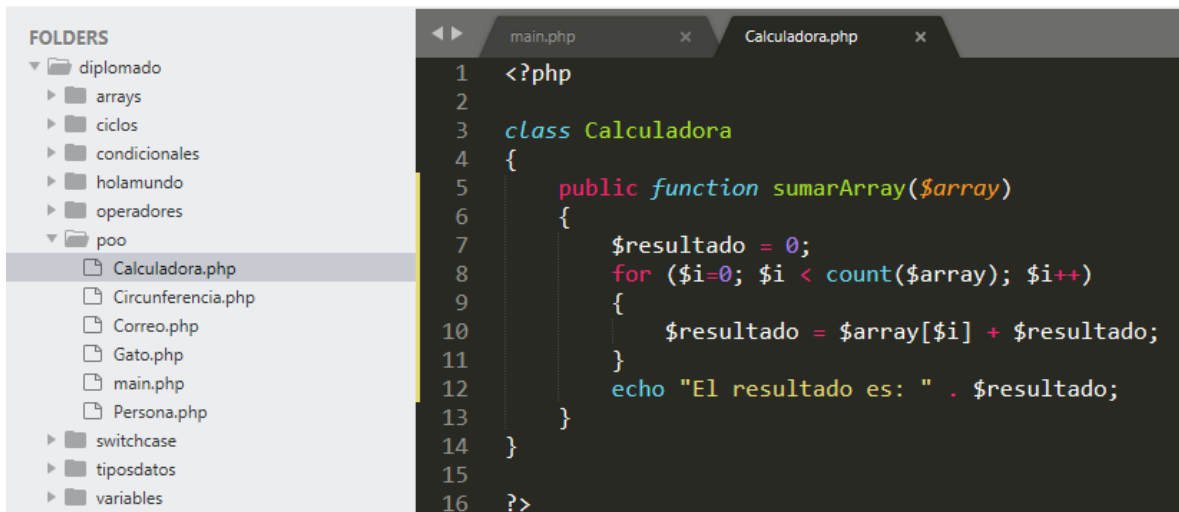
Paso por referencia. Son argumentos de tipo objeto, *array*, entre otros, que no contienen los valores de estos sino su respectiva dirección en memoria.



```

1 <?php
2
3 require_once('Calculadora.php');
4
5 $Casio = new Calculadora();
6
7 $numeros = array(5, 9, 2, 3, 4, 7);
8
9 $Casio->sumarArray($numeros);
10
11 ?>

```



```

1 <?php
2
3 class Calculadora
4 {
5     public function sumarArray($array)
6     {
7         $resultado = 0;
8         for ($i=0; $i < count($array); $i++)
9         {
10             $resultado = $array[$i] + $resultado;
11         }
12         echo "El resultado es: " . $resultado;
13     }
14 }
15
16 ?>

```

Ilustraciones 39 y 40.

El paso por referencia se caracteriza por el uso de tipos de datos por referencia que permiten el paso de la dirección de memoria del valor original, más no la copia del valor.

Constructores

Es un método que contiene las acciones que se realizarán por defecto al crear un objeto, en la mayoría de los casos se inicializan los valores de los atributos en el constructor.

- No es obligatoria su creación.
- Se utiliza la palabra reservada `__construct`.
- No retorna ningún valor.
- Se recomienda el uso de modificadores de accesos.

```
public function __construct($raza, $nombre, $edad, $encargada)
{
    $this->raza = $raza;
    $this->nombre = $nombre;
    $this->edad = $edad;
    $this->encargada = $encargada;
}
```

Ilustración 41.

El constructor recibe todos los atributos de la clase en su inicialización. Desde la declaración de la instancia u objeto, sería de la siguiente forma:

```
<?php

require_once('Gato.php');

$Anahi = new Gato("Gato", "Anahi", 12, "Estefania");

?>
```

Ilustración 42.

El objeto debe recibir cuatro argumentos, en el mismo orden conforme se declara dentro del constructor, la instrucción **new Gato(..)** se encarga de enviar los argumentos al constructor para ser inicializados los atributos.

Para el uso correcto de los constructores debemos entender el uso y aplicación de los **argumentos** y **parámetros** que ayudan a inicializar esos valores. Véase:

```
1  <?php
2
3  class Calculadora
4  {
5      private $numero1;
6      private $operador;
7      private $numero2;
8
9      public function __construct($numero1, $operador, $numero2)
10     {
11         $this->numero1 = $numero1;
12         $this->operador = $operador;
13         $this->numero2 = $numero2;
14     }
15
16     public function dibujarOperacion()
17     {
18         switch ($this->operador)
19         {
20             case "+":
21                 $this->dibujarSuma();
22                 break;
23             case "-":
24                 $this->dibujarResta();
25                 break;
26             case "/":
27                 $this->dibujarDivision();
28                 break;
29             case "*":
30                 $this->dibujarMultiplicacion();
31                 break;
```

```
32 ▼      default:
33          echo "Error de operador";
34          break;
35      }
36  }
37
38  public function dibujarSuma()
39  ▼  {
40      $suma = $this->numero1 + $this->numero2;
41      echo $this->numero1 . $this->operador . $this->numero2 . " = " . $suma;
42  }
43
44  public function dibujarResta()
45  ▼  {
46      $resta = $this->numero1 - $this->numero2;
47      echo $this->numero1 . $this->operador . $this->numero2 . " = " . $resta;
48  }
49
50  public function dibujarDivision()
51  ▼  {
52      $division = $this->numero1 / $this->numero2;
53      echo $this->numero1 . $this->operador . $this->numero2 . " = " . $division;
54  }
55
56  public function dibujarMultiplicacion()
57  ▼  {
58      $multiplicacion = $this->numero1 * $this->numero2;
59      echo $this->numero1 . $this->operador . $this->numero2 . " = " . $multiplicacion;
60  }
61  }
62
```

Ilustraciones 43 y 44.

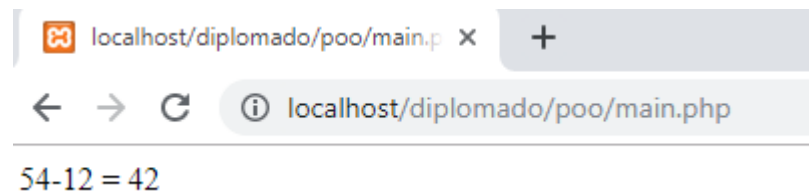
La clase calculadora es un ejemplo básico que permitirá describir adecuadamente el funcionamiento del constructor. La clase cuenta con tres atributos, todos privados y sin inicializar sus respectivos valores, dado que eso se hará dentro del constructor.

Este, para ser un constructor, debe cumplir con unas características básicas para el correcto funcionamiento:

- Es público.
- El nombre es la palabra reservada `__construct`.
- Puede o no recibir parámetros.
- No cuenta con retorno.

Entiéndase el funcionamiento dentro del constructor, este recibe unos parámetros desde la instancia u objeto que lo invoca que permiten asignar a los atributos privados unos valores en tiempo de ejecución. Véase los parámetros y argumentos.

```
1  <?php
2
3  require_once('Calculadora.php');
4
5  $Casio = new Calculadora(54, "-", 12);
6
7  $Casio->dibujarOperacion();
8
9  ?>
```



Ilustraciones 45 y 46.

En la archivo Main se creará las respectiva instancia de la clase calculadora, por lo que se deben pasar respectivamente los tres argumentos.

Recuerda que los argumentos son los valores que se le asignan al método llamado por el objeto, y los parámetros son los valores que recibe el método en su declaración. Más adelante se profundizará.

Estos argumentos son transformados en parámetros en el método de forma de objeto y el constructor de la clase cuenta ahora con los siguientes valores:

```
private $numero1; // 54
private $operador; // -
private $numero2; // 12
```

Ilustración 47.

Puesto que el constructor le asignó los valores a los atributos del presente objeto por medio de la siguiente operación:

```
public function __construct($numero1, $operador, $numero2)
{
    //Atributos      //Parametros
    $this->numero1    = $numero1;
    $this->operador    = $operador;
    $this->numero2    = $numero2;
}
```

Ilustración 48.

En esta operación de asignación convergen los argumentos y los párametros, dado que los valores de los párametros los deben tomar dichos argumentos que se desean inicializar.

this cumple el papel de diferenciar un atributo de un pámetro, dado que por convención es ideal mantener los nombres, *this* permite hacer la referencia al objeto actual (es decir la clase).

En el ejercicio anterior el constructor recibía todos los pámetros, ¿cómo funcionan los constructores vacíos?

```
private $numero1;
private $operador;
private $numero2;

public function __construct()
{

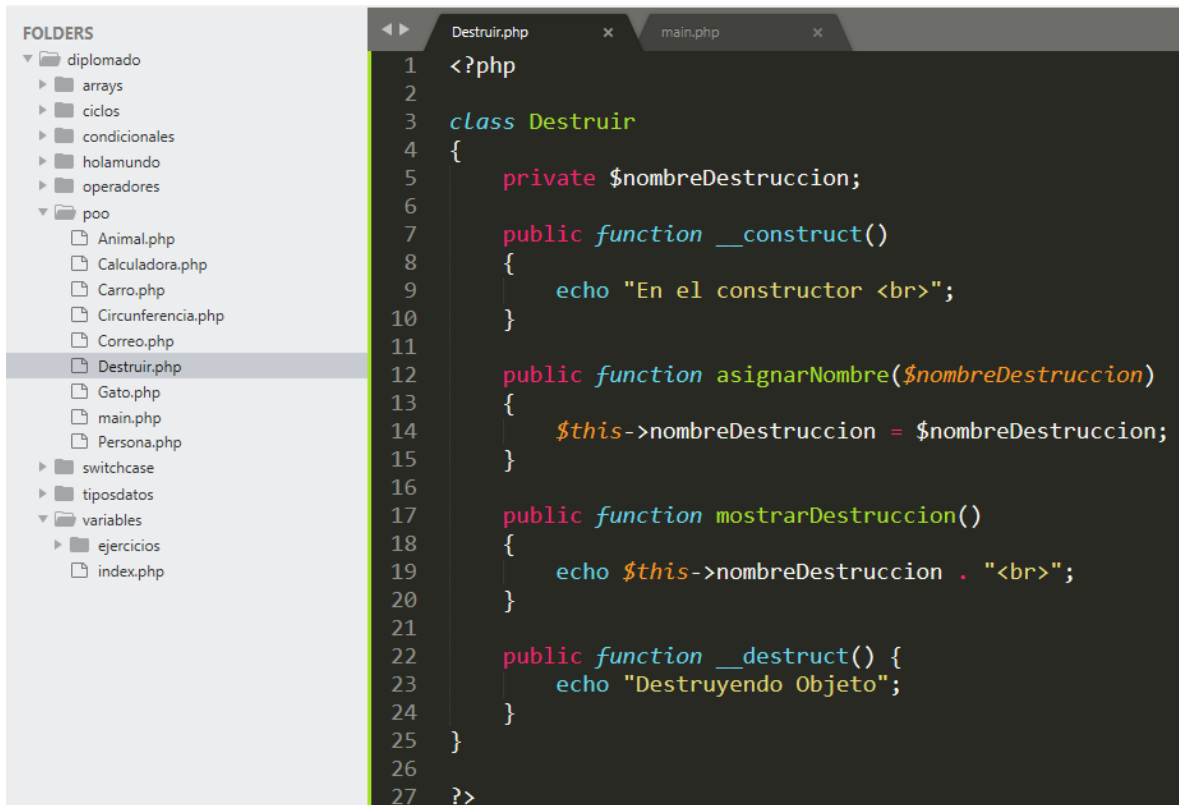
}
```

Ilustración 49.

Es admisible la declaración de un constructor vacío, pero ¿cómo se inicializarán los valores? Aquí vuelven a aplicar los *getters* y *setters*, intenta aplicarlos al ejercicios de la calculadora.

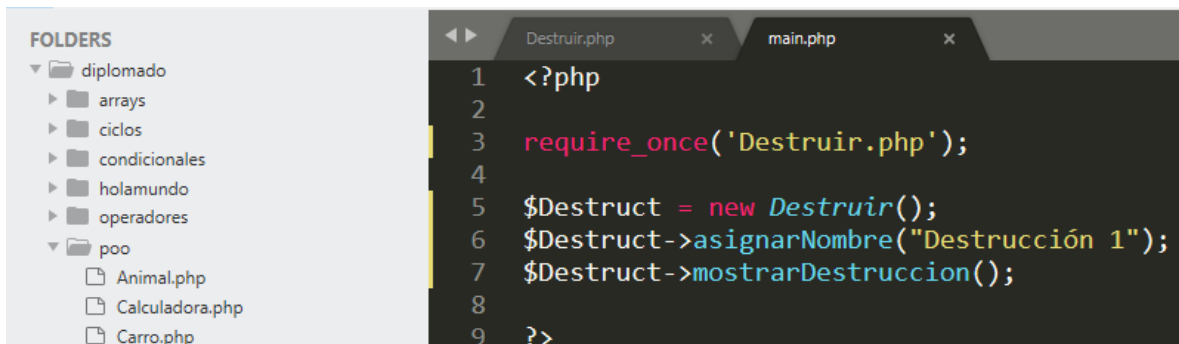
Destructores

El método destructor será llamado tan pronto como no haya otras referencias a un objeto determinado, o en cualquier otra circunstancia de finalización.



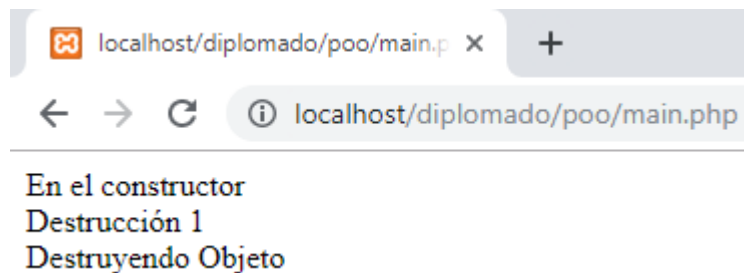
```

1 <?php
2
3 class Destruir
4 {
5     private $nombreDestruccion;
6
7     public function __construct()
8     {
9         echo "En el constructor <br>";
10    }
11
12    public function asignarNombre($nombreDestruccion)
13    {
14        $this->nombreDestruccion = $nombreDestruccion;
15    }
16
17    public function mostrarDestruccion()
18    {
19        echo $this->nombreDestruccion . "<br>";
20    }
21
22    public function __destruct() {
23        echo "Destruyendo Objeto";
24    }
25 }
26
27 ?>
  
```



```

1 <?php
2
3 require_once('Destruir.php');
4
5 $Destruct = new Destruir();
6 $Destruct->asignarNombre("Destrucción 1");
7 $Destruct->mostrarDestruccion();
8
9 ?>
  
```



Ilustraciones 50, 51 y 52.

Las características de este método son:

- El objetivo principal es liberar recursos que solicitó el objeto (conexión a la base de datos, creación de imágenes dinámicas, etc.),
- Es el último método que se ejecuta de la clase.
- Se ejecuta en forma automática, es decir, no tenemos que llamarlo.
- Debe llamarse `__destruct`.
- No retorna datos.
- Es menos común su uso que el constructor, ya que PHP gestiona bastante bien la liberación de recursos en forma automática.

Métodos

Son procesos o acciones disponibles para el objeto, creados a partir de una clase.

Un método es una abstracción de una operación que puede hacer o realizarse con un objeto. Una clase puede declarar cualquier número de métodos que lleven a cabo operaciones de todo tipo con los objetos.

Algunas características que se encuentran dentro de los métodos son:

- Permiten reutilizar código.
- Pueden o no retornar valores, los métodos que no retornan valores son conocidos como métodos de tipo *void*, a diferencia de los métodos que sí retorna y son de un tipo en específico: *int*, *string*, entre otros, más adelante ahondaremos en este tipo de métodos.
- Pueden existir N cantidades de métodos dentro de una clase.
- Puede contener o no parámetros.
- Un método puede contener N parámetros, aunque se recomienda no sobrecargar los métodos.
- Los métodos deben retornar un tipo de dato del mismo tipo que está diseñado el método, es decir, un método de tipo entero debe retornar un entero.
- Los nombres deben ser muy claros y descriptivos con la acción que van a realizar.
- Se recomienda el uso de los modificadores de acceso, en especial diseñar los métodos con el modificador de acceso *public*.

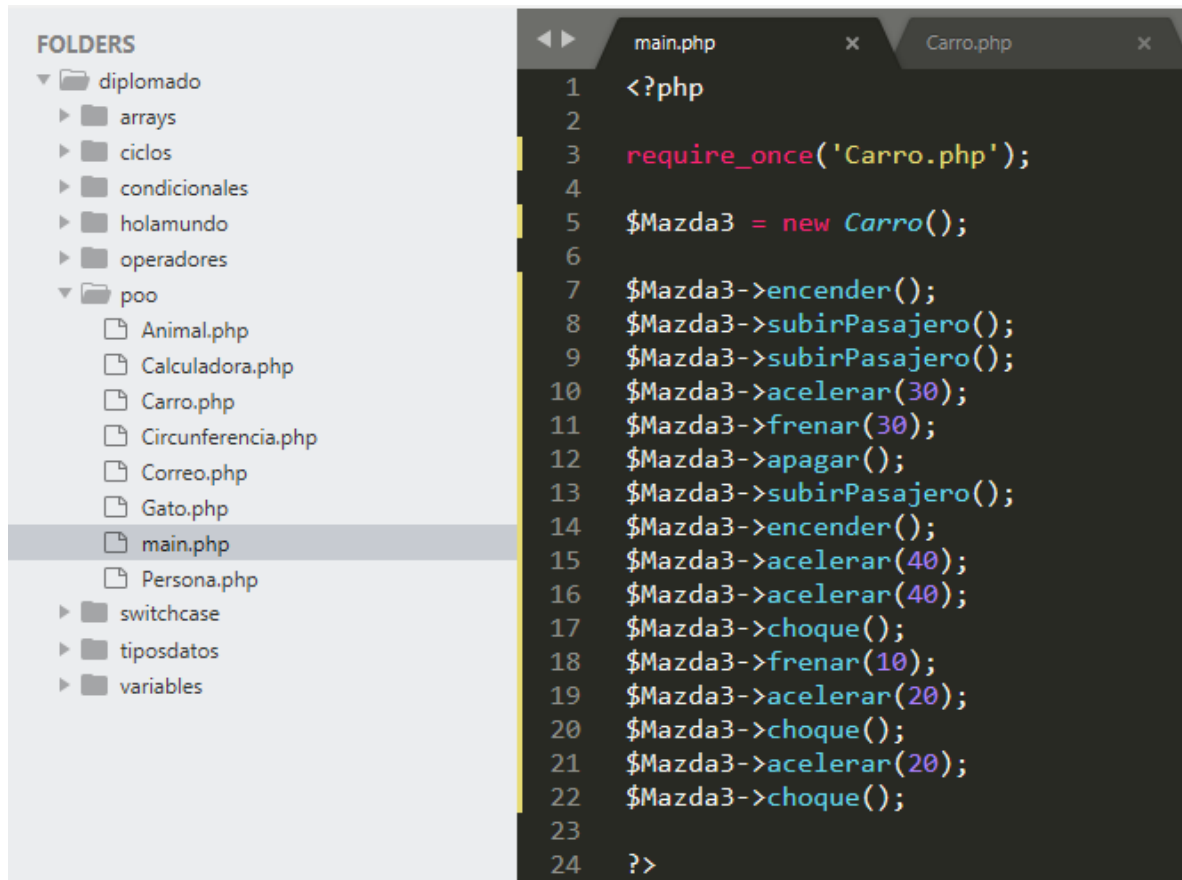
La estructura general para implementar los métodos dentro de las clases es la siguiente:

- Modificador de acceso.
- Palabra reservada *function*.
- Nombre del método.
- Parámetros.

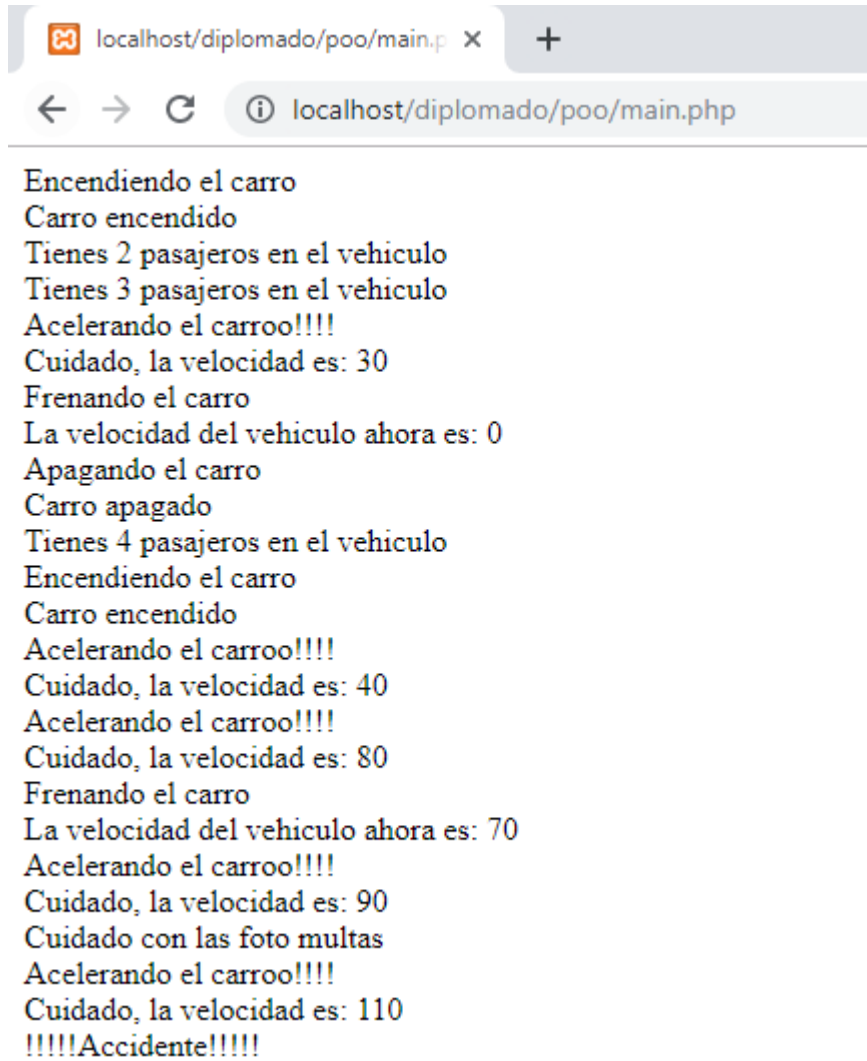
En el siguiente ejemplo vamos a simular el funcionamiento de un carro a partir de métodos, teniendo en cuenta el estado del carro, los pasajeros y la velocidad, además, podremos simular algunas funciones del carro, como frenar, apagar, encender, etc.

```
1  <?php
2
3  class Carro
4  {
5      private $velocidad = 0;
6      private $estadoCarro = "apagado";
7      private $personas = 1;
8
9      public function encender()
10     {
11         echo "Encendiendo el carro </br>";
12         $this->estadoCarro = "encendido";
13         echo "Carro encendido </br>";
14     }
15
16     public function frenar($rapidez)
17     {
18         echo "Frenando el carro </br>";
19         $this->velocidad = $this->velocidad - $rapidez;
20         echo "La velocidad del vehiculo ahora es: " . $this->velocidad . " </br>";
21     }
22
23     public function apagar()
24     {
25         echo "Apagando el carro </br>";
26         $this->estadoCarro = "apagado";
27         echo "Carro apagado </br>";
28     }
29 }
```

```
main.php x Carro.php x
23 public function apagar()
24 {
25     echo "Apagando el carro </br>";
26     $this->estadoCarro = "apagado";
27     echo "Carro apagado </br>";
28 }
29
30 public function acelerar($rapidez)
31 {
32     echo "Acelerando el carroo!!!! </br>";
33     $this->velocidad = $this->velocidad + $rapidez;
34     echo "Cuidado, la velocidad es: " . $this->velocidad . " </br>";
35 }
36
37 public function subirPasajero()
38 {
39     $this->personas++;
40     echo "Tienes " . $this->personas . " pasajeros en el vehiculo </br>";
41 }
42
43 public function choque()
44 {
45     if($this->velocidad >= 100)
46     {
47         echo "!!!!!!Accidente!!!!!! </br>";
48     }
49     else if($this->velocidad > 80)
50     {
51         echo "Cuidado con las foto multas </br>";
52     }
53 }
54 }
55
```



```
1 <?php
2
3 require_once('Carro.php');
4
5 $Mazda3 = new Carro();
6
7 $Mazda3->encender();
8 $Mazda3->subirPasajero();
9 $Mazda3->subirPasajero();
10 $Mazda3->acelerar(30);
11 $Mazda3->frenar(30);
12 $Mazda3->apagar();
13 $Mazda3->subirPasajero();
14 $Mazda3->encender();
15 $Mazda3->acelerar(40);
16 $Mazda3->acelerar(40);
17 $Mazda3->choque();
18 $Mazda3->frenar(10);
19 $Mazda3->acelerar(20);
20 $Mazda3->choque();
21 $Mazda3->acelerar(20);
22 $Mazda3->choque();
23
24 ?>
```



```
Encendiendo el carro
Carro encendido
Tienes 2 pasajeros en el vehiculo
Tienes 3 pasajeros en el vehiculo
Acelerando el carroo!!!!
Cuidado, la velocidad es: 30
Frenando el carro
La velocidad del vehiculo ahora es: 0
Apagando el carro
Carro apagado
Tienes 4 pasajeros en el vehiculo
Encendiendo el carro
Carro encendido
Acelerando el carroo!!!!
Cuidado, la velocidad es: 40
Acelerando el carroo!!!!
Cuidado, la velocidad es: 80
Frenando el carro
La velocidad del vehiculo ahora es: 70
Acelerando el carroo!!!!
Cuidado, la velocidad es: 90
Cuidado con las foto multas
Acelerando el carroo!!!!
Cuidado, la velocidad es: 110
!!!!Accidente!!!!
```

Ilustraciones 53, 54, 55 y 56.

Otro ejemplo de los métodos en acción es el siguiente ejemplo de la clase **adivinar**, hay muchos conceptos y es interesante prestar atención a su funcionamiento.

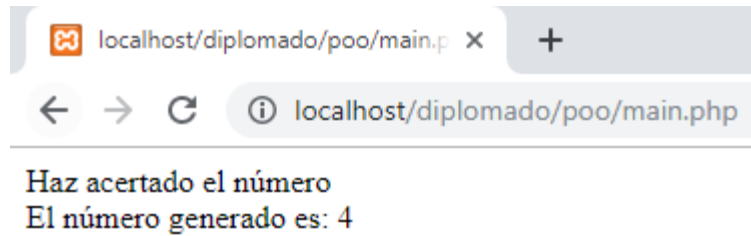
Suponga que se debe generar una lotería a partir de un límite inferior y superior, y acercar el número aleatorio que se genere. Hay que implementar una clase y un método que realice todo el proceso.

```
main.php x Adivina.php x
1  <?php
2
3  class Adivina
4  {
5      private $numeroGenerado;
6      private $numeroElegido;
7      private $min;
8      private $max;
9
10     public function __construct($min, $max, $numeroElegido)
11     {
12         $this->min = $min;
13         $this->max = $max;
14         $this->numeroElegido = $numeroElegido;
15         $this->verificar();
16     }
17
18     public function generarNumero()
19     {
20         return $this->numeroGenerado = rand($this->min, $this->max);
21     }
22
23     public function verificar()
24     {
25         if($this->numeroElegido > $this->max)
26         {
27             echo "El número es mayor al límite br>";
28         }
29         elseif ($this->numeroElegido < $this->min)
30         {
31             echo "El número es menor al límite br>";
32         }
33         elseif($this->generarNumero() == $this->numeroElegido)
34         {
35             echo "Haz acertado el número <br>";
36         }
37         $this->mostrarNumeroGenerado();
38     }
39
```

```
39
40     public function mostrarNumeroGenerado()
41     {
42         echo "El número generado es: " . $this->numeroGenerado;
43     }
44
45 }
46
47 ?>
```



```
1  <?php
2
3  require_once('Adivina.php');
4
5  $Juego = new Adivina(1,5,4);
6
7  ?>
```



Ilustraciones 57, 58, 59 y 60.

Métodos *void*. La utilidad de los métodos *void* radica en que son métodos que no cuentan con ningún tipo de retorno.

Algunas características de este tipo de método son:

- Se centran en realizar acciones que no requieren retornar un valor en específico, también suele ser usado para mostrar mensajes.
- Se caracterizan por no tener un tipo de dato asociado.
- El modificador de acceso más común es *public*.
- Puede o no recibir parámetros.
- Los nombres deben ser muy claros y descriptivos con la acción que van a realizar.

Observa algunos ejemplos de métodos *void*.

El método `encender()` cumple la función de asignar un valor al atributo `estadoCarro`, de la clase `carro`, a partir del objeto.

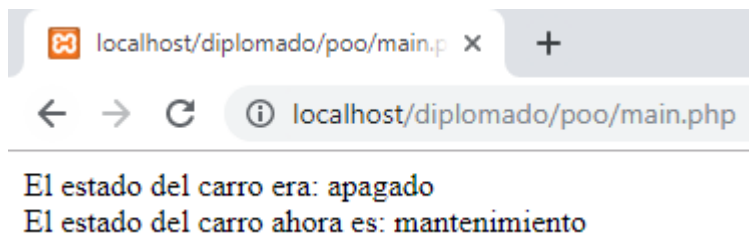
```
public function encender()
{
    echo "Encendiendo el carro </br>";
    $this->estadoCarro = "encendido";
    echo "Carro encendido </br>";
}
```

```
public function cambiarEstado($estadoCarro)
{
    echo "El estado del carro era: " . $this->estadoCarro . " </br>";
    $this->estadoCarro = $estadoCarro;
    echo "El estado del carro ahora es: " . $this->estadoCarro . " </br>";
}
```

Ilustraciones 61 y 62.

El método `cambiarEstado ()` cumple la función de asignar un nuevo valor al atributo `estadoCarro` de la clase `carro` por medio del parámetro que recibe (recordar el uso de *this*).

```
4
5  $Mazda3 = new Carro();
6
7  $Mazda3->cambiarEstado("mantenimiento");
8
```



localhost/diplomado/poo/main.p x +
localhost/diplomado/poo/main.php
El estado del carro era: apagado
El estado del carro ahora es: mantenimiento

Ilustraciones 63 y 64.

Métodos de tipo. La utilidad de los métodos de tipo radica en que son métodos que cuentan un retorno en base al tipo de dato que fue declarado.

Algunas características de este tipo de método son:

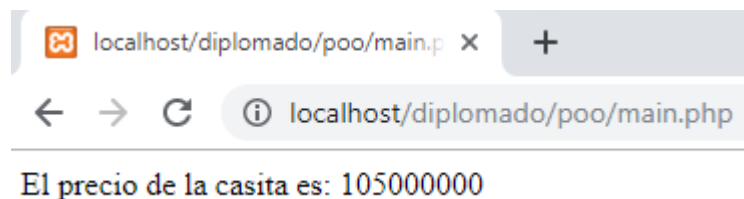
- Se centran en realizar acciones que deben contar con un retorno obligatorio con base al tipo de dato definido.
- Se caracterizan por tener un tipo de dato asociado.
- El modificador de acceso más común es *public*.
- Puede o no recibir parámetros.
- Se debe tener en cuenta la correcta identificación de los nombres de los métodos.

- Los nombres deben ser muy claros y descriptivos con la acción que van a realizar.

Observa algunos ejemplos de métodos de tipo.

```
1  <?php
2
3  class Casa
4  {
5      private $precio = 0;
6
7      public function __construct($precio)
8      {
9          $this->precio = $precio;
10     }
11
12     public function aumentarPrecio($precio)
13     {
14         return $this->precio = $this->precio + $precio;
15     }
16 }
17
18 ?>
```

```
1  <?php
2
3  require_once('Casa.php');
4
5  $miCasita = new Casa(85000000);
6  $precioCasita = $miCasita->aumentarPrecio(20000000);
7
8  echo "El precio de la casita es: " . $precioCasita;
9  ?>
```



Ilustraciones 65, 66 y 67.

El método `aumentarPrecio ()` cumple la función de incrementar y retornar el atributo del precio de la clase `Casa` con base al parámetro que recibe. En estos momentos no se está realizando una tipificación estricta, donde se le obligue al método a devolver un tipo estrictamente definido.

PHP 7 añade soporte para las declaraciones de tipo de devolución. De forma similar a las declaraciones de tipo de argumento, las declaraciones de tipo de devolución especifican el tipo del valor que serán devuelto desde una función. Están disponibles los mismos tipos para las declaraciones de tipo de devolución que para las declaraciones de tipo de argumento.

Los tipos de datos que se pueden usar para los métodos y para los argumentos son los siguientes:

- *String* - *int* - *bool* - *float* - *array* - *Self* - *objeto de clase* - *callable*

Vamos a ondear principalmente en los tipos de:

- *String*
- *Int*
- *Bool*
- *Float*
- *Array*

Siguiendo el ejemplo de la clase casa y el método aumentarPrecio, realicemos ahora una tipificación estricta.

```
public function aumentarPrecio($precio) : int  
{  
    return $this->precio = $this->precio + $precio;  
}
```

Ilustración 68.

Por medio de la declaración *int*, PHP obliga al método a que el valor de retorno sea sí o sí un número entero.

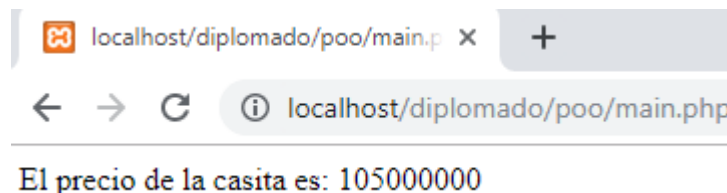
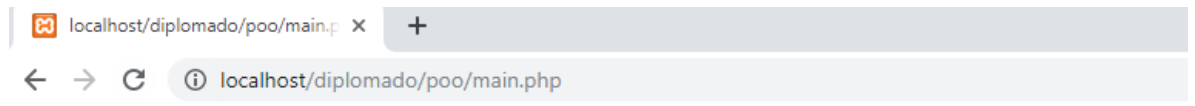


Ilustración 69.

Este sería el resultado esperado, ahora, por ejemplo, realicemos un cambio en el retorno del método.



```
public function aumentarPrecio((int $precio) : int
{
    return $this->precio = $this->precio + $precio;
}
```

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

$miCasita = new Casa(85000000);
$precioCasita = $miCasita->aumentarPrecio("Diego");
```

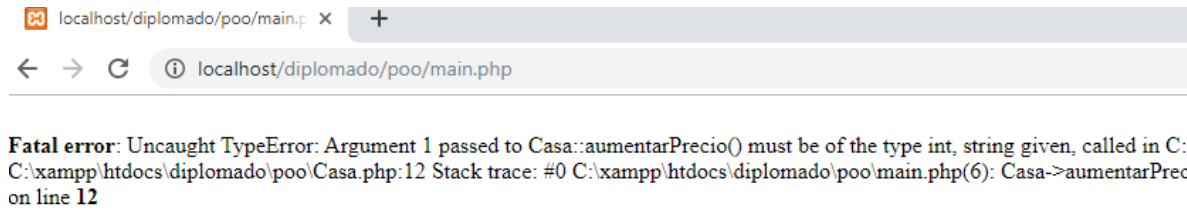


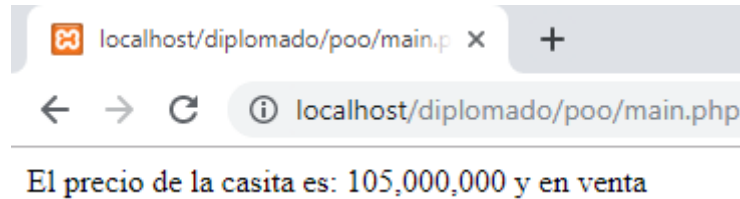
Ilustración 74.

El tipo de error es `TypeError` con el argumento que ha sido pasado al parámetro `$precio` en el método `aumentarPrecio` y los tipos *int* a *string*.

Veamos ahora algunos ejemplos con otros tipos de datos y otros métodos haciendo uso de varios conceptos, en especial las dos tipificaciones que acabamos de ver.

```
1  <?php
2
3  class Casa
4  {
5      private $precio = 0;
6      private $propietario;
7      private $estadoVenta;
8
9      public function __construct(int $precio, string $propietario, bool $estadoVenta)
10     {
11         $this->precio = $precio;
12         $this->propietario = $propietario;
13         $this->estadoVenta = $estadoVenta;
14     }
15
16     public function aumentarPrecio(int $precio) : int
17     {
18         return $this->precio = $this->precio + $precio;
19     }
20
21     public function cambiarPropietario(string $propietario) : void
22     {
23         $this->propietario = $propietario;
24     }
25
26     public function estadoCasa() : string
27     {
28         return $this->estadoVenta ? "en venta" : "no venta";
29     }
30 }
31
32 ?>
```

```
1 <?php
2
3 require_once('Casa.php');
4
5 $miCasita = new Casa(85000000, "Diego Palacio", true);
6 $precioCasita = $miCasita->aumentarPrecio(20000000);
7 $miCasita->cambiarPropietario("Carlos Palacio");
8 $estadoCasita = $miCasita->estadoCasa();
9
10 echo "El precio de la casita es: " . number_format($precioCasita) . " y " . $estadoCasita;
11
12 ?>
```



Ilustraciones 75, 76 y 77.

Modificadores de acceso

Los modificadores de acceso introducen el concepto de encapsulamiento. El encapsulamiento busca controlar el acceso a los datos que conforman un objeto o instancia; de este modo una clase, y por ende sus objetos que hacen uso de modificadores de acceso (especialmente privados), son objetos encapsulados.

Los modificadores de acceso permiten dar un nivel de seguridad mayor restringiendo el acceso a diferentes atributos, métodos y constructores, asegurando que el usuario deba seguir una «ruta» especificada para acceder a la información.

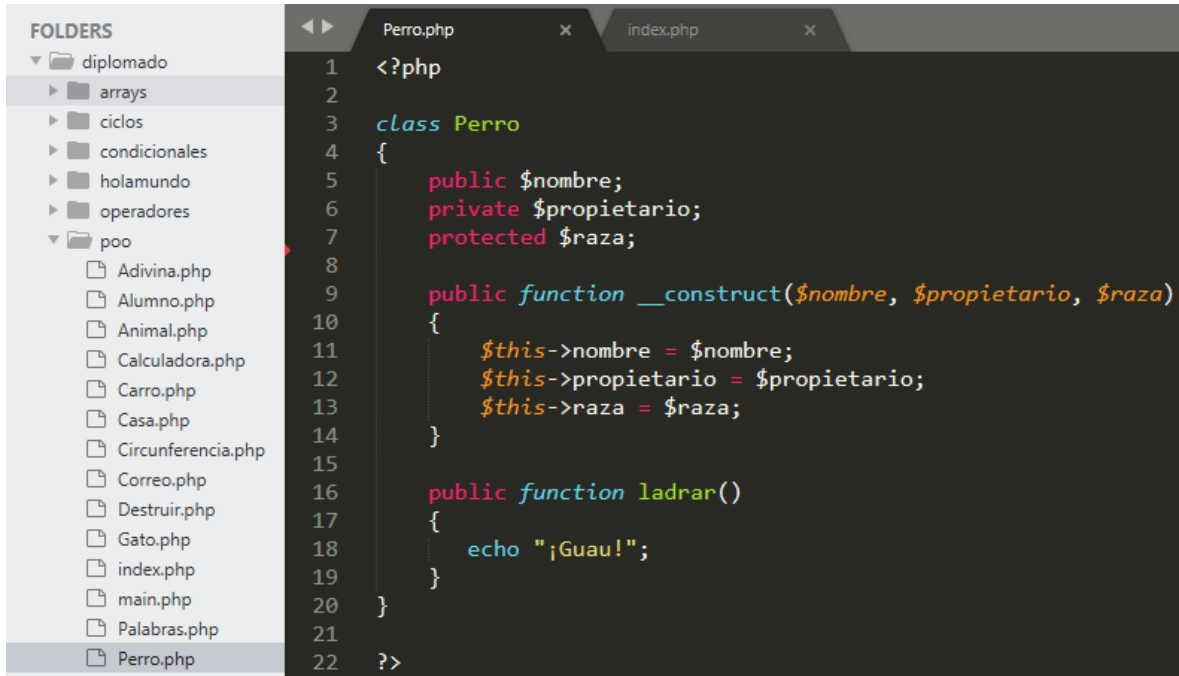
Implementando el uso de los modificadores de acceso se podrá asegurar que un valor no será modificado incorrectamente. Generalmente el acceso a los atributos se consigue por medio de los métodos *get* y *set*, es estrictamente necesario que los atributos de una clase sean privados.

Existen cuatro modificadores de acceso:

Public - default: la propiedad o método podrá usarse en cualquier parte del *script*.

Private: la propiedad o método solo podrá usarse en la clase a la que pertenece.

Protected: la propiedad o método se podrá usar por la clase a la que pertenece y por sus descendientes.



```

1  <?php
2
3  class Perro
4  {
5      public $nombre;
6      private $propietario;
7      protected $raza;
8
9      public function __construct($nombre, $propietario, $raza)
10     {
11         $this->nombre = $nombre;
12         $this->propietario = $propietario;
13         $this->raza = $raza;
14     }
15
16     public function ladrar()
17     {
18         echo "¡Guau!";
19     }
20 }
21
22 ?>

```

Ilustración 78.

Métodos mágicos

Los métodos mágicos en PHP permiten realizar acciones en objetos cuando suceden determinados eventos que activan dichos métodos. Estos, denominados con doble barra baja `__nombreMetodo()`, determinan cómo reaccionará el objeto.

Los métodos mágicos disponibles en PHP son los siguientes:

Métodos	
<code>__construct</code>	<code>__sleep</code>
<code>__destruct</code>	<code>__call</code>
<code>__get</code>	<code>__callStatic</code>
<code>__set</code>	<code>__clone</code>
<code>__isset</code>	<code>__invoke</code>
<code>__unset</code>	

Ilustración 79.

Suponemos que tenemos una aplicación en la que queremos extraer tweets de Twitter. Extraemos los tweets del usuario y queremos transformar cada tweet en un objeto de PHP con el que trabajar. Hagamos uso de algunos métodos (Lázaro, 2018).

`construct()` y `destruct()`. El método mágico más utilizado en PHP es `__construct()`, un método que es llamado automáticamente cuando se crea el objeto. Permite adjuntar parámetros para construir el objeto.

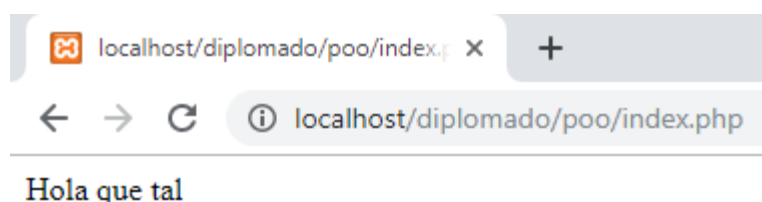
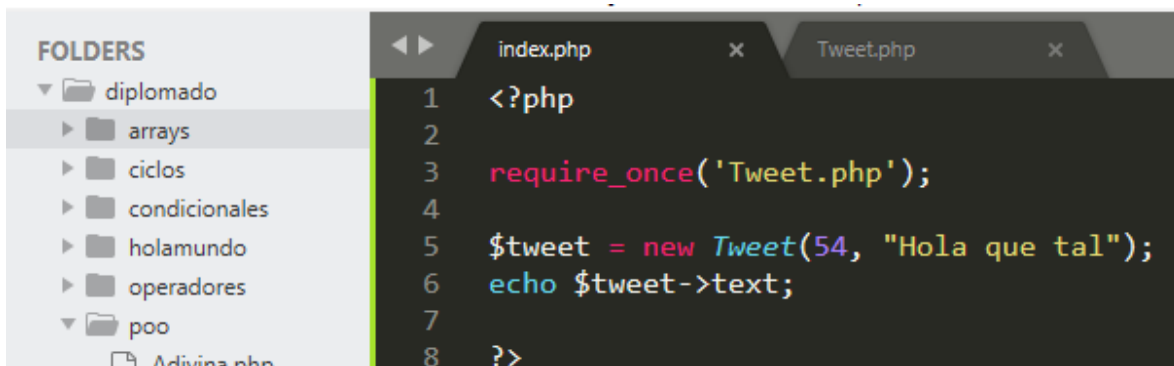
Cuando creamos una nueva instancia del objeto `tweet` podemos pasarle parámetros que se iniciarán en el método `__construct()`. No se puede llamar al método ya que es llamado automáticamente al crear el objeto.

Al igual que podemos construir cosas cuando instanciamos un objeto, también podemos destruirlas cuando el objeto se destruye. Si por ejemplo tenemos una conexión a la base de datos en el objeto y queremos asegurarnos de que se cierra al destruir el objeto, podemos hacer lo siguiente:

```
public function __destruct()
{
    echo "Objeto destruido";
}
```

Ilustración 80.

`__get()` y `__set()`. Cuando se trabaja con objetos, las propiedades pueden ser accedidas de esta forma si son *public*:



Ilustraciones 81 y 82.

Pero en la práctica lo normal y aconsejable es que las propiedades sean *protected* o *private*, por lo que no podemos acceder a ellas de esa forma.

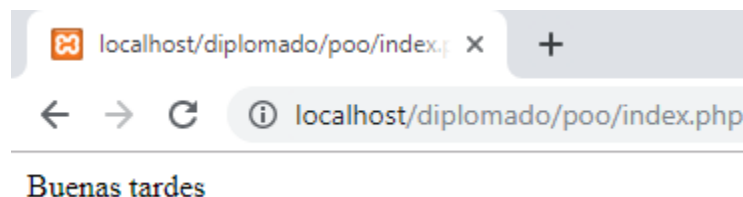
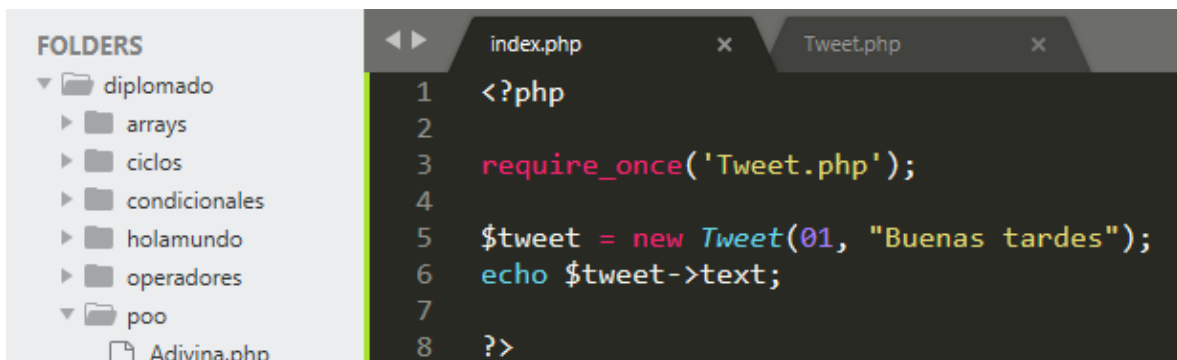
El método mágico `__get()` permite poder acceder a estas propiedades:

```
public function __get($property)
{
    return property_exists($this, $property) ? $this->$property : "No existe";
}
```

Ilustración 83.

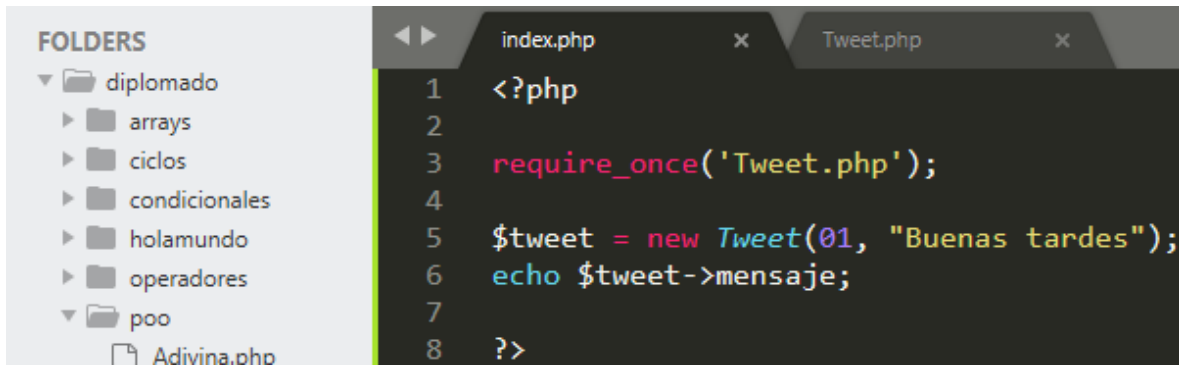
El método acepta el nombre de la propiedad que estabas buscando como argumento. No es necesario llamar al método `__get()`, ya que PHP lo llamará automáticamente cuando trates de acceder a una propiedad que no es *public*. También lo llamará cuando no existe una propiedad.

Ahora, con el método mágico y los atributos en privado, o *protected*, se puede realizar el proceso normal para el llamado a los atributos.



Ilustraciones 84 y 85.

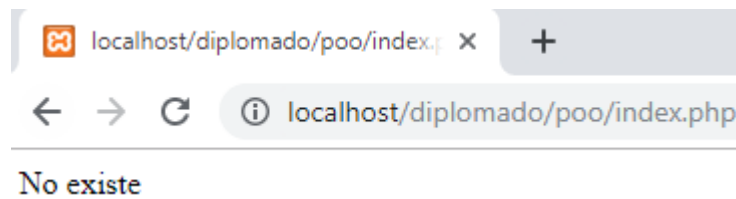
Funciona perfectamente para las propiedades declaradas, en caso de que no exista opera de la siguiente forma:



```

1 <?php
2
3 require_once('Tweet.php');
4
5 $tweet = new Tweet(01, "Buenas tardes");
6 echo $tweet->mensaje;
7
8 ?>

```



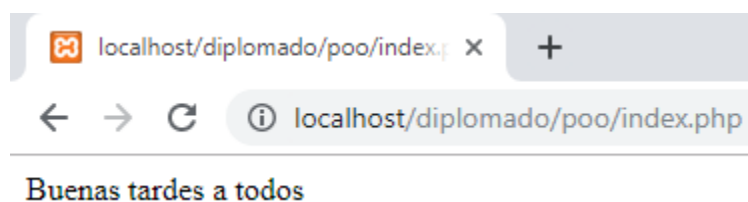
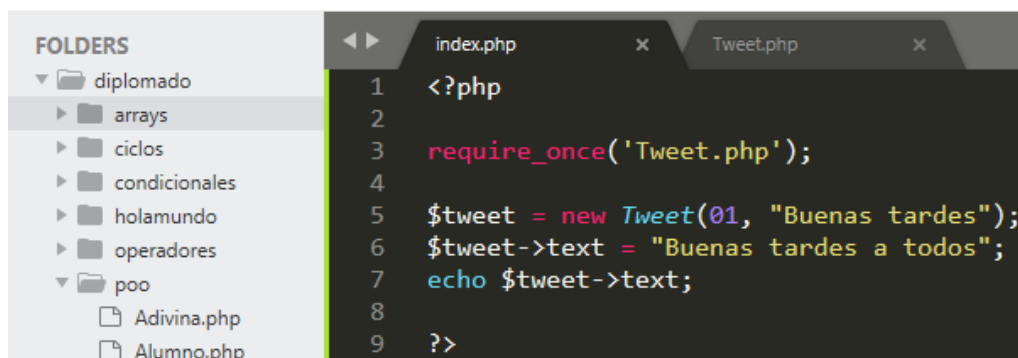
Ilustraciones 86 y 87.

Si lo que quieres es establecer un valor a un atributo del objeto actual que no es accesible, puedes insertar un método `__set()`. Este método toma el atributo al que intentabas acceder y el valor que intentabas establecer como sus dos argumentos.

```

public function __set($property, $value)
{
    return property_exists($this, $property) ? $this->$property = $value : "En blanco";
}

```

```

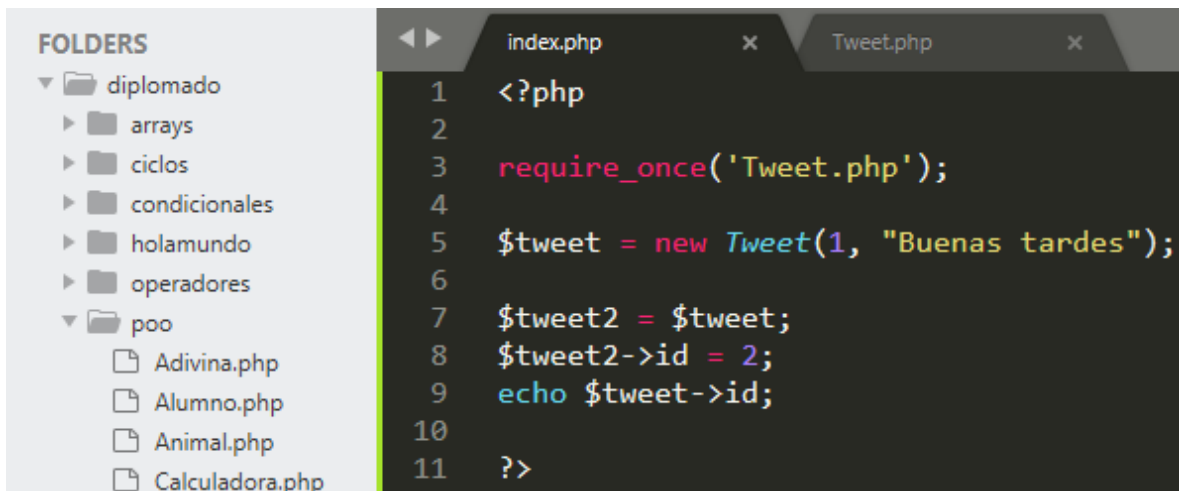
1 <?php
2
3 require_once('Tweet.php');
4
5 $tweet = new Tweet(01, "Buenas tardes");
6 $tweet->text = "Buenas tardes a todos";
7 echo $tweet->text;
8
9 ?>

```

Ilustraciones 88, 89 y 90.

En este último ejemplo, por medio del constructor inicializamos el objeto y todos sus atributos, seguidamente establecimos un nuevo valor al atributo texto por medio de set y obtuvimos el valor gracias a *get*.

___**clone()**__. Cuando se hace una copia de un objeto en PHP mediante su asignación a otra variable, esta variable sigue haciendo referencia al mismo objeto. Esto significa que si cambias cualquier cosa en uno se cambiará en ambos dos:



The screenshot shows a code editor with two files: `index.php` and `Tweet.php`. The `index.php` file contains the following PHP code:

```

1  <?php
2
3  require_once('Tweet.php');
4
5  $tweet = new Tweet(1, "Buenas tardes");
6
7  $tweet2 = $tweet;
8  $tweet2->id = 2;
9  echo $tweet->id;
10
11 ?>

```

The `Tweet.php` file is also visible but its content is not shown in the screenshot.

Ilustración 91.

En el ejemplo anterior se crea un objeto de la clase `tweet` llamado «`tweet`» y una variable llamada «`tweet2`», que contendrá el objeto anteriormente creado («`tweet`») y con la asignación de valor a «`tweet2`» también cambiará «`tweet`» por la referencia existente.

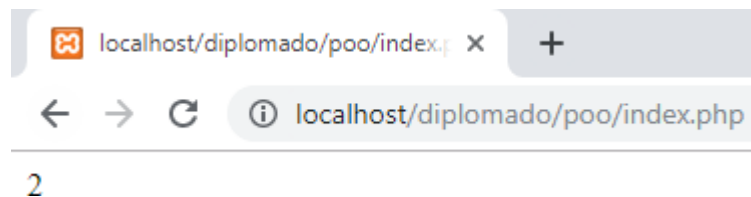
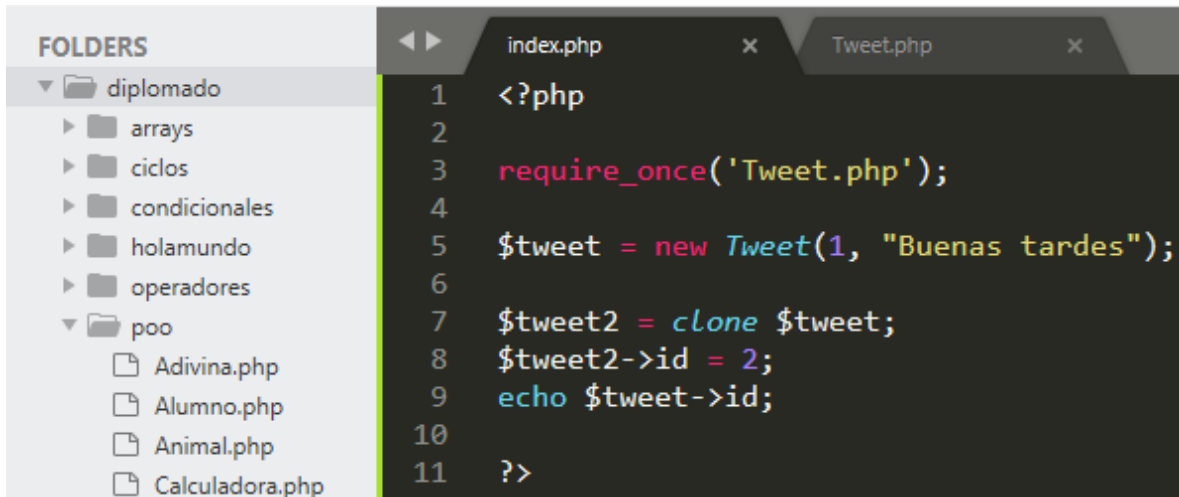


Ilustración 92.

Para crear una copia del objeto, independiente uno de otro, se usa `clone`:



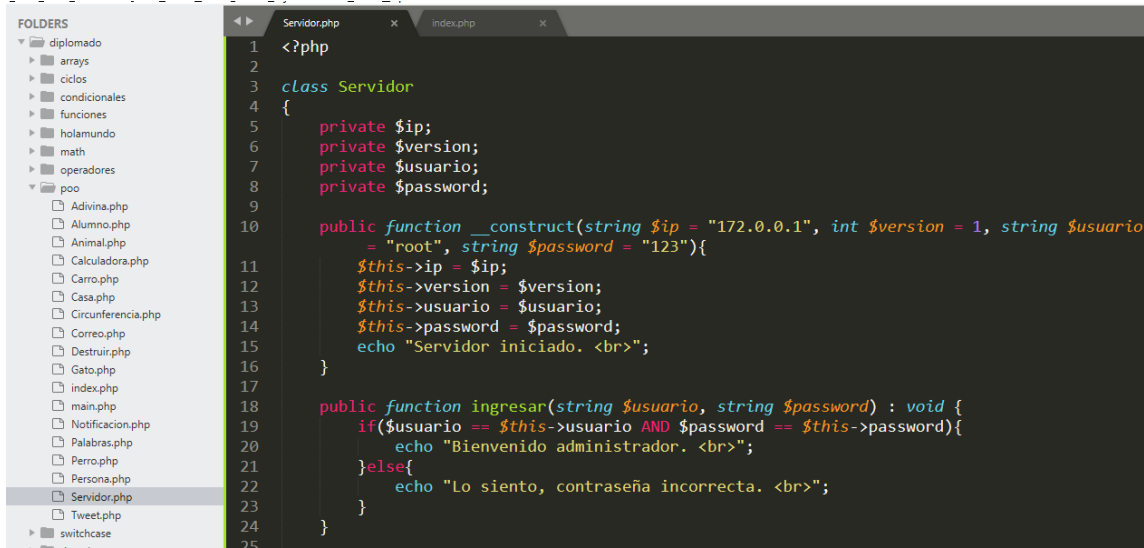
```
1 <?php
2
3 require_once('Tweet.php');
4
5 $tweet = new Tweet(1, "Buenas tardes");
6
7 $tweet2 = clone $tweet;
8 $tweet2->id = 2;
9 echo $tweet->id;
10
11 ?>
```

Ilustración 93.

Estos son algunos de los métodos mágicos más comunes, aunque principalmente destacan *get*, *set* y *construct*.

Return. La palabra reservada *return* permite retornar valores dentro de los métodos, además de detener la ejecución de este.

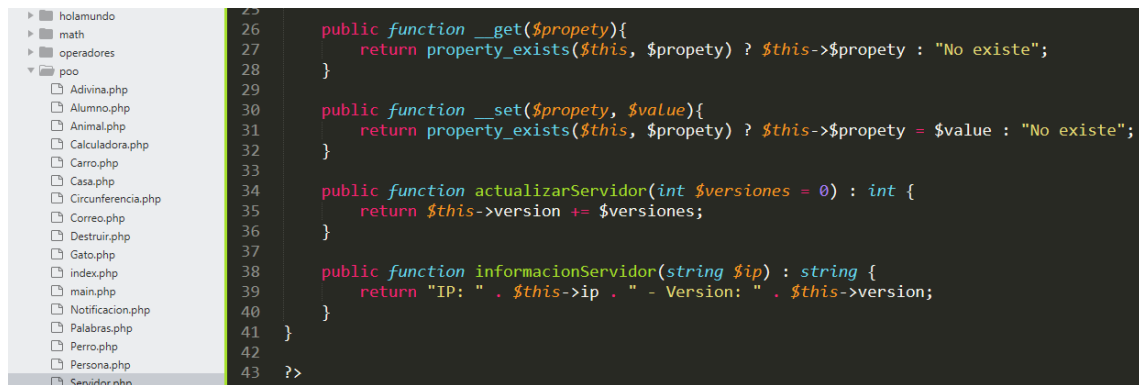
- Cualquier instrucción que se encuentre después de la ejecución de *return* NO será ejecutada. Es común encontrar métodos con múltiples sentencias *return* al interior de condicionales, pero una vez que el código ejecuta una sentencia *return* lo que haya de allí hacia abajo no se ejecutará.
- Si un método con tipo estricto va a retornar, el tipo del valor que se retorna debe coincidir con el del tipo estricto declarado del método, es decir, si se declara *int* el método, el valor retornado debe ser un número entero.
- En el caso de los métodos *void* (sin retorno) se puede usar la sentencia *return*, pero sin ningún tipo de valor, solo se usaría como una manera de terminar la ejecución del método.



```

1 <?php
2
3 class Servidor
4 {
5     private $ip;
6     private $version;
7     private $usuario;
8     private $password;
9
10    public function __construct(string $ip = "172.0.0.1", int $version = 1, string $usuario
        = "root", string $password = "123"){
11        $this->ip = $ip;
12        $this->version = $version;
13        $this->usuario = $usuario;
14        $this->password = $password;
15        echo "Servidor iniciado. <br>";
16    }
17
18    public function ingresar(string $usuario, string $password) : void {
19        if($usuario == $this->usuario AND $password == $this->password){
20            echo "Bienvenido administrador. <br>";
21        }else{
22            echo "Lo siento, contraseña incorrecta. <br>";
23        }
24    }
25

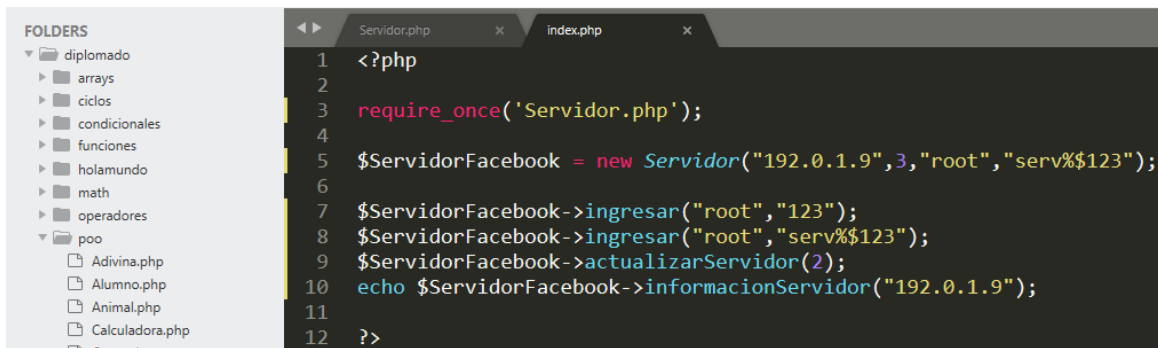
```



```

26    public function __get($property){
27        return property_exists($this, $property) ? $this->$property : "No existe";
28    }
29
30    public function __set($property, $value){
31        return property_exists($this, $property) ? $this->$property = $value : "No existe";
32    }
33
34    public function actualizarServidor(int $versiones = 0) : int {
35        return $this->version += $versiones;
36    }
37
38    public function informacionServidor(string $ip) : string {
39        return "IP: " . $this->ip . " - Version: " . $this->version;
40    }
41 }
42
43 ?>

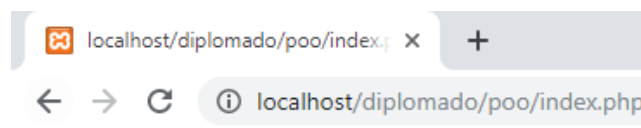
```



```

1 <?php
2
3 require_once('Servidor.php');
4
5 $ServidorFacebook = new Servidor("192.0.1.9",3,"root","serv%123");
6
7 $ServidorFacebook->ingresar("root","123");
8 $ServidorFacebook->ingresar("root","serv%123");
9 $ServidorFacebook->actualizarServidor(2);
10 echo $ServidorFacebook->informacionServidor("192.0.1.9");
11
12 ?>

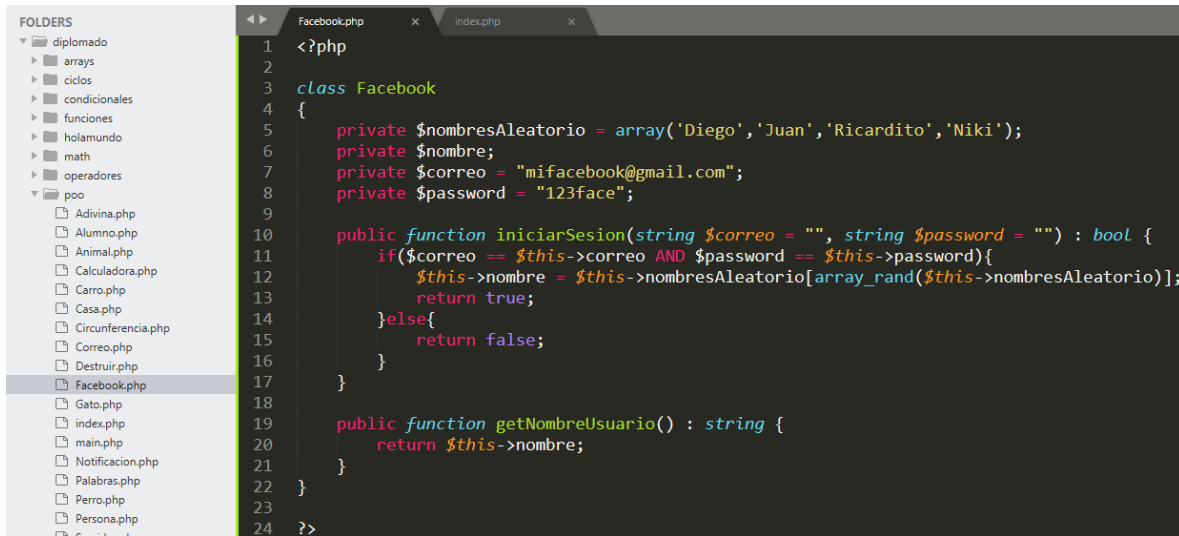
```



Ilustraciones 94, 95, 96 y 97.

Objetos

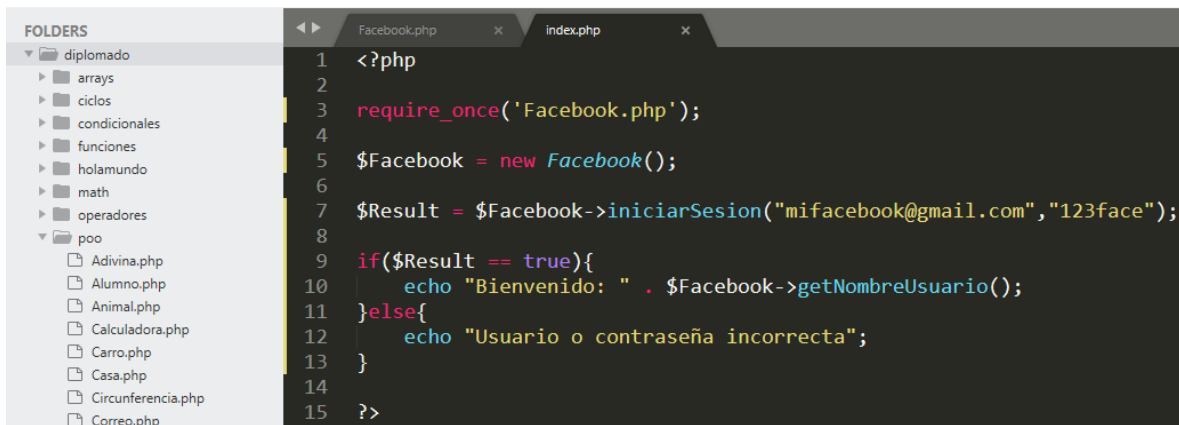
Los objetos son ejemplares de una clase. Cuando se crea un objeto se debe especificar la clase a partir de la cual se creará. Esta acción de crear un objeto a partir de una clase se llama **instanciar**.



```

1 <?php
2
3 class Facebook
4 {
5     private $nombresAleatorio = array('Diego','Juan','Ricardito','Niki');
6     private $nombre;
7     private $correo = "mifacebook@gmail.com";
8     private $password = "123face";
9
10    public function iniciarSesion(string $correo = "", string $password = "") : bool {
11        if($correo == $this->correo AND $password == $this->password){
12            $this->nombre = $this->nombresAleatorio[array_rand($this->nombresAleatorio)];
13            return true;
14        }else{
15            return false;
16        }
17    }
18
19    public function getNombreUsuario() : string {
20        return $this->nombre;
21    }
22 }
23
24 ?>

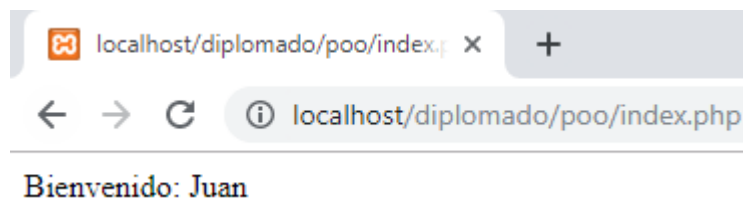
```



```

1 <?php
2
3 require_once('Facebook.php');
4
5 $Facebook = new Facebook();
6
7 $Result = $Facebook->iniciarSesion("mifacebook@gmail.com","123face");
8
9 if($Result == true){
10     echo "Bienvenido: " . $Facebook->getNombreUsuario();
11 }else{
12     echo "Usuario o contraseña incorrecta";
13 }
14
15 ?>

```



Ilustraciones 98, 99 y 100.

Los objetos tienen características fundamentales que nos permiten conocerlos mediante la observación, identificación y el estudio posterior de su comportamiento; estas características son:

Identidad. La identidad es la propiedad que permite diferenciar a un objeto y distinguirse de otros. Generalmente esta propiedad es tal que da nombre al objeto. Por ejemplo, el «verde» como un objeto concreto de una clase color; la propiedad que da identidad única a este objeto es precisamente su «color», verde. Tanto es así que no tiene sentido usar otro nombre para el objeto que no sea el valor de la propiedad que lo identifica.

En programación, la identidad de todos los objetos sirve para comparar si dos objetos son iguales o no. No es raro encontrar que en muchos lenguajes de programación la identidad de un objeto esté determinada por la dirección de memoria de la computadora en la que se encuentra el objeto, pero este comportamiento puede ser variado redefiniendo la identidad del objeto a otra propiedad.

Comportamiento. El comportamiento de un objeto está directamente relacionado con su funcionalidad y determina las operaciones que este puede realizar. La funcionalidad de un objeto está determinada, primariamente, por su responsabilidad. Una de las ventajas fundamentales de la POO es la reusabilidad del código; un objeto es más fácil de reutilizarse en tanto su responsabilidad sea mejor definida y más concreta.

Una tarea fundamental a la hora de diseñar una aplicación informática es definir el comportamiento que tendrán los objetos de las clases involucradas en la aplicación, asociando la funcionalidad requerida por la aplicación a las clases adecuadas.

Estado. El estado de un objeto se refiere al conjunto de atributos y sus valores en un instante de tiempo dado. El comportamiento de un objeto puede modificar el estado de este.

Para declarar un objeto de una clase específica el proceso es el siguiente:

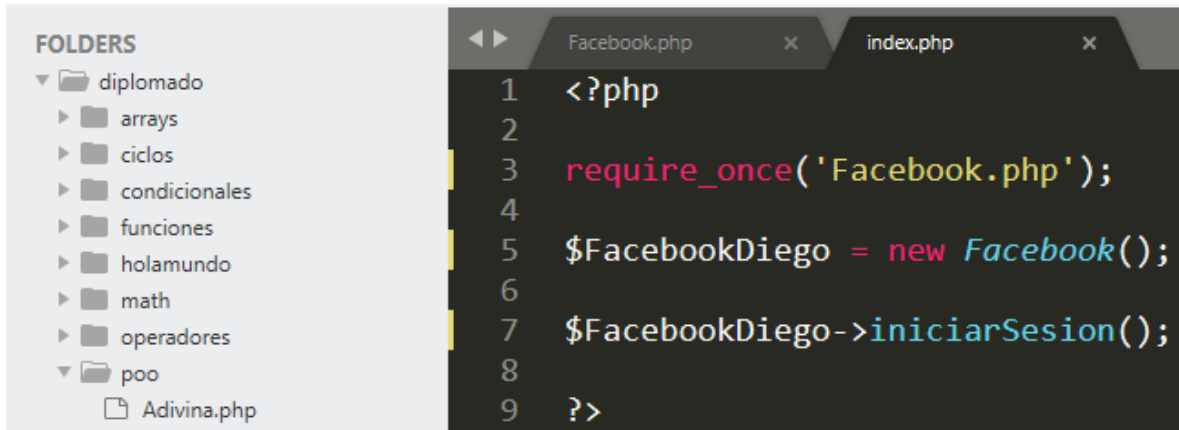


Ilustración 101.

Con la declaración **FacebookDiego** se está reservando un espacio de memoria para almacenar una referencia (dirección de memoria) a un objeto de la clase **Facebook**. Al respecto, es importante comprender que **FacebookDiego** no es un objeto, sino una variable que almacenará la referencia a un objeto (de la clase **Facebook**) que todavía no existe. Seguidamente, mediante la sentencia **FacebookDiego = new Facebook ()**, el operador *new* creará un objeto de la clase **Facebook**, reservando memoria para guardar sus atributos. Finalmente, con el operador de asignación (=), la dirección de memoria donde esté creado el objeto es asignada a **FacebookDiego**.

Hay otros aspectos para tener en cuenta en POO con los objetos:

- Se pueden instanciar N cantidad de objetos de una clase, siempre y cuando tengan nombres diferentes.
- Un objeto puede acceder a todos métodos definidos en la clase de la cual está generada la instancia a partir de los modificadores de acceso.

En programación orientada a objetos (POO), una instancia de programa (por ejemplo, un programa ejecutándose en una computadora) es tratada como un conjunto dinámico de objetos interactuando entre sí.

Los objetos en la POO extienden la noción más general descrita en secciones anteriores para modelar un tipo muy específico que está definido fundamentalmente por:

- **Atributos:** representan los datos asociados al objeto o, lo que es lo mismo,

sus propiedades o características. Los atributos y sus valores, en un momento dado, determinan el estado de un objeto.

- **Métodos:** acceden a los atributos de una manera predefinida e implementan el comportamiento del objeto.

Los atributos y métodos de un objeto están definidos por su clase, aunque una instancia puede poseer atributos que no fueron definidos en su clase. Algo similar ocurre con los métodos: una instancia puede contener métodos que no estén definidos en su clase de la misma manera que una clase puede declarar ciertos métodos como «métodos de clase», y estos (en dependencia del lenguaje) podrán estar o no presentes en la instancia.

En el caso de la mayoría de los objetos, los atributos solo pueden ser accedidos a través de los métodos; de esta manera es más fácil garantizar que los datos permanecerán siempre en un estado bien definido.

En un lenguaje en el que cada objeto es creado a partir de una clase, un objeto es llamado una instancia de esa clase. Cada objeto pertenece a un tipo y dos objetos que pertenezcan a la misma clase tendrán el mismo tipo de dato.

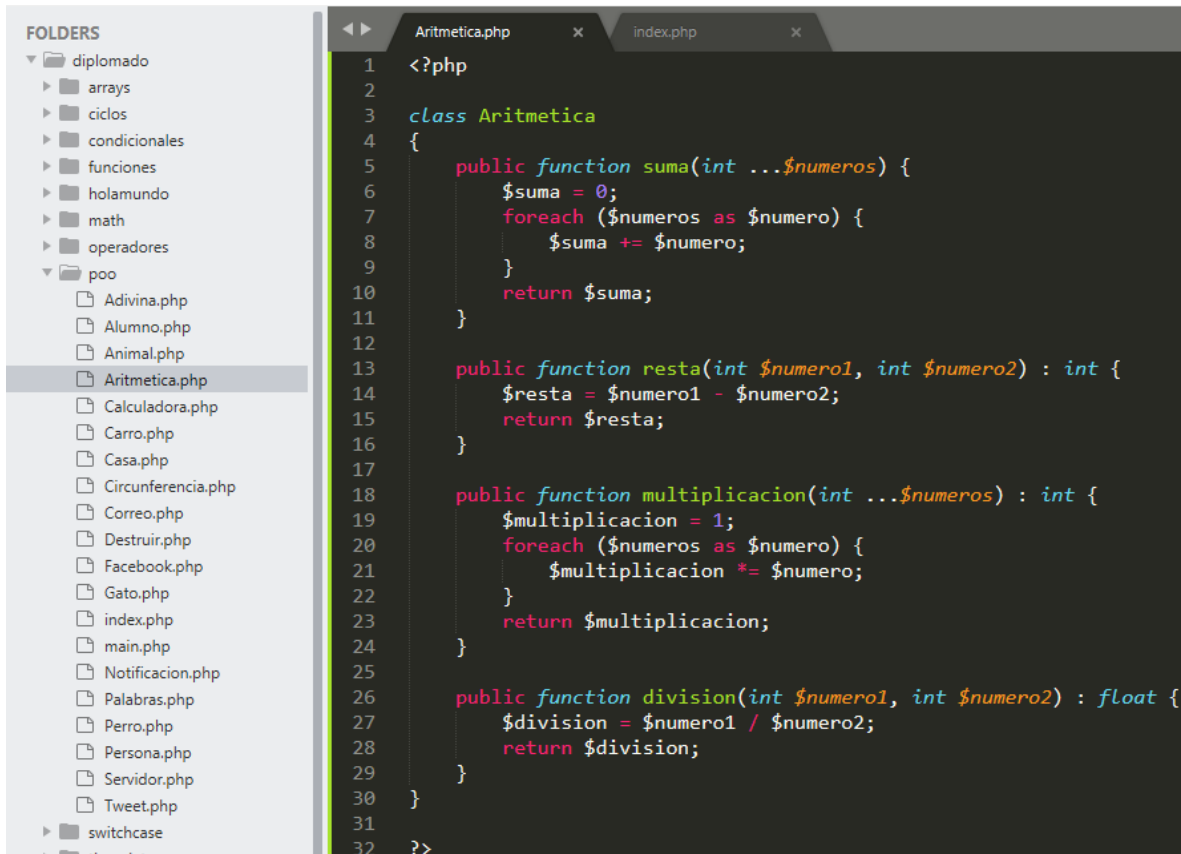
Crear una instancia de una clase es entonces referido como instanciar la clase.

En PHP, el operador «->» es usado para referirse o «llamar» a un método particular de un objeto.

Considere como ejemplo una clase aritmética llamada Aritmética. Esta clase contiene métodos como «sumar», «restar», »multiplicar», «dividir», etc., que calculan el resultado de realizar estas operaciones sobre dos números.

Un objeto de esta clase puede ser utilizado para calcular el producto de dos números, pero primeramente sería necesario definir dicha clase y crear un objeto.

Declaración de la clase **aritmética**:



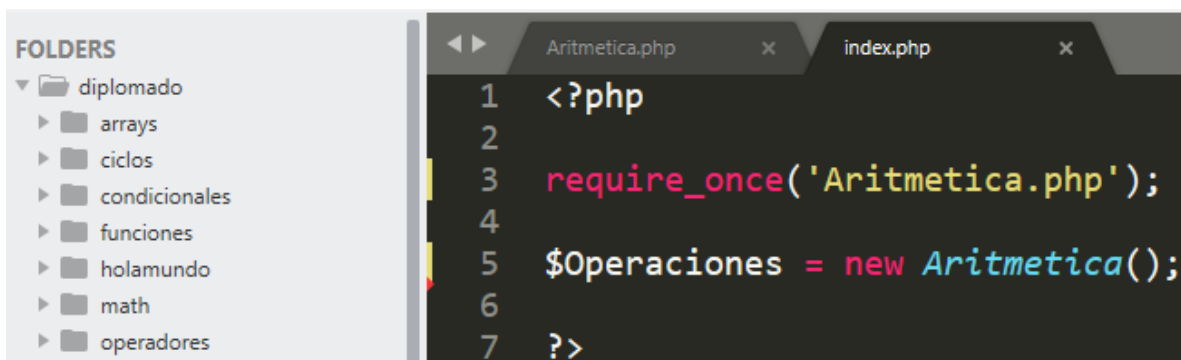
```

1  <?php
2
3  class Aritmetica
4  {
5      public function suma(int ...$numeros) {
6          $suma = 0;
7          foreach ($numeros as $numero) {
8              $suma += $numero;
9          }
10         return $suma;
11     }
12
13     public function resta(int $numero1, int $numero2) : int {
14         $resta = $numero1 - $numero2;
15         return $resta;
16     }
17
18     public function multiplicacion(int ...$numeros) : int {
19         $multiplicacion = 1;
20         foreach ($numeros as $numero) {
21             $multiplicacion *= $numero;
22         }
23         return $multiplicacion;
24     }
25
26     public function division(int $numero1, int $numero2) : float {
27         $division = $numero1 / $numero2;
28         return $division;
29     }
30 }
31
32 ?>

```

Ilustración 102.

Creación del objeto **operaciones**:



```

1  <?php
2
3  require_once('Aritmetica.php');
4
5  $Operaciones = new Aritmetica();
6
7  ?>

```

Ilustración 103.

Ejecución de los métodos de la clase **aritmética** a partir del objeto **operaciones**:



Ilustración 104.

Resultados de la ejecución de los métodos con base en los argumentos.

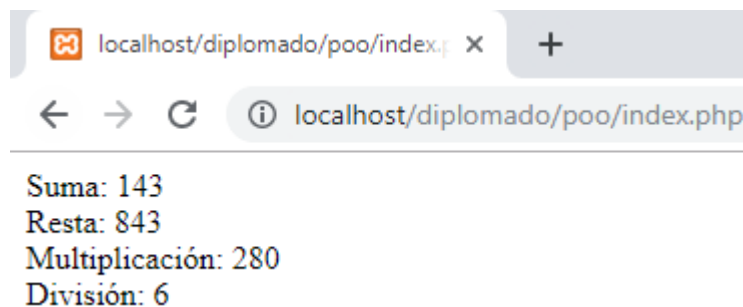


Ilustración 105.

Herencia

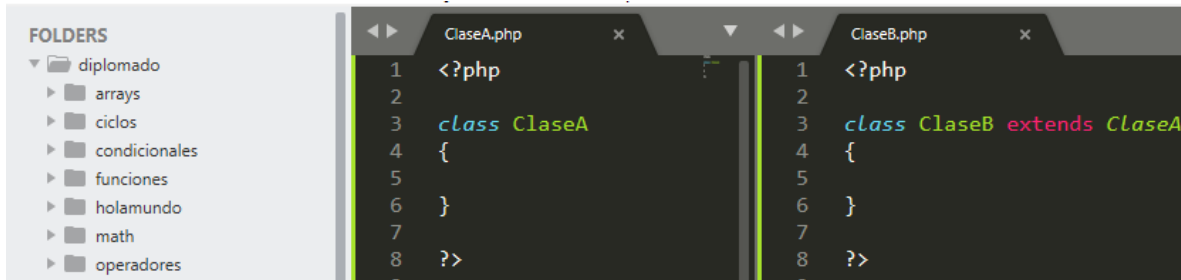
La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases y objetos.

Esto proporciona una de las ventajas principales de la programación orientada a objetos: la reutilización de código previamente desarrollado, ya que permite a una clase más específica incorporar la estructura y comportamiento de una clase más general.

Cuando una clase B se construye a partir de otra A mediante la herencia, la clase B hereda todos los atributos, métodos y clases internas de la clase A. Además, la clase B puede redefinir los componentes heredados y añadir atributos, métodos y clases internas específicas.

Para indicar que la clase B (clase descendiente, derivada, hija o subclase) hereda de la clase A (clase ascendiente, heredada, padre, base o superclase) se

emplea la palabra reservada *extends* en la cabecera de la declaración de la clase descendiente. La sintaxis es la siguiente:



```

FOLDERS
└─ diplomado
  └─ arrays
  └─ ciclos
  └─ condicionales
  └─ funciones
  └─ holamundo
  └─ math
  └─ operadores

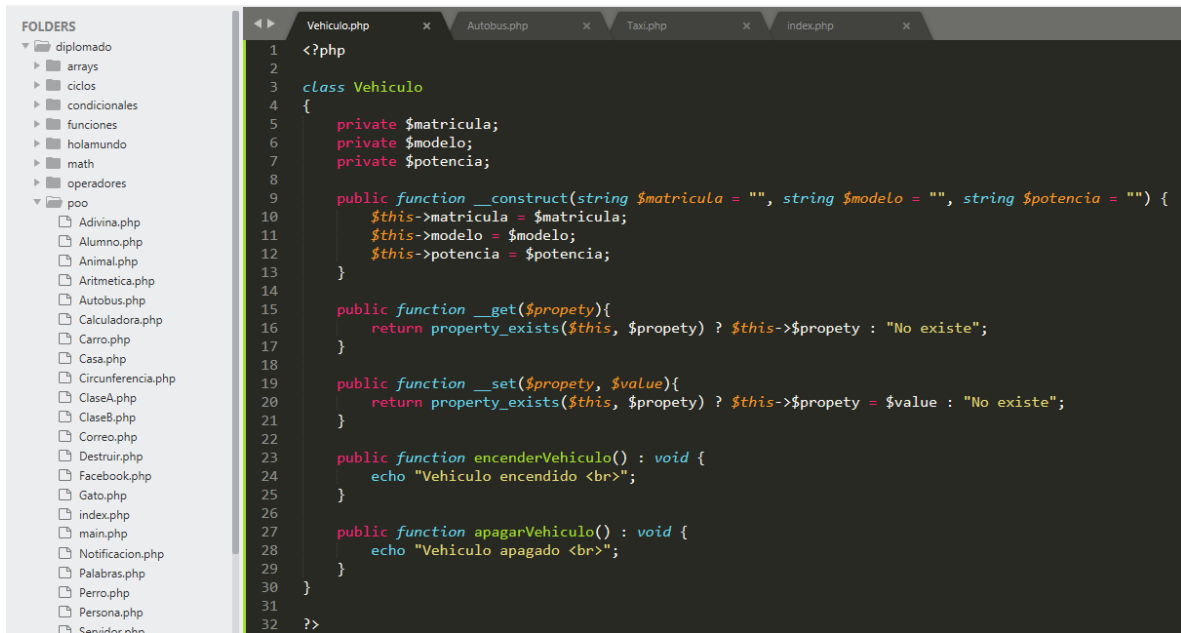
ClaseA.php
1 <?php
2
3 class ClaseA
4 {
5
6 }
7
8 ?>

ClaseB.php
1 <?php
2
3 class ClaseB extends ClaseA
4 {
5
6 }
7
8 ?>
  
```

Ilustración 106.

Un ejemplo más claro de la aplicación de la herencia sería el siguiente:

Un programa donde se encuentre una clase **taxi** y **autobús** con características similares, como encendido, apagado, matrícula, modelo y potencia. Podría resumirse esos elementos en común en una clase padre (**vehículo**) y las clases hijos (**taxi** y **autobús**) heredar de la clase padre.



```

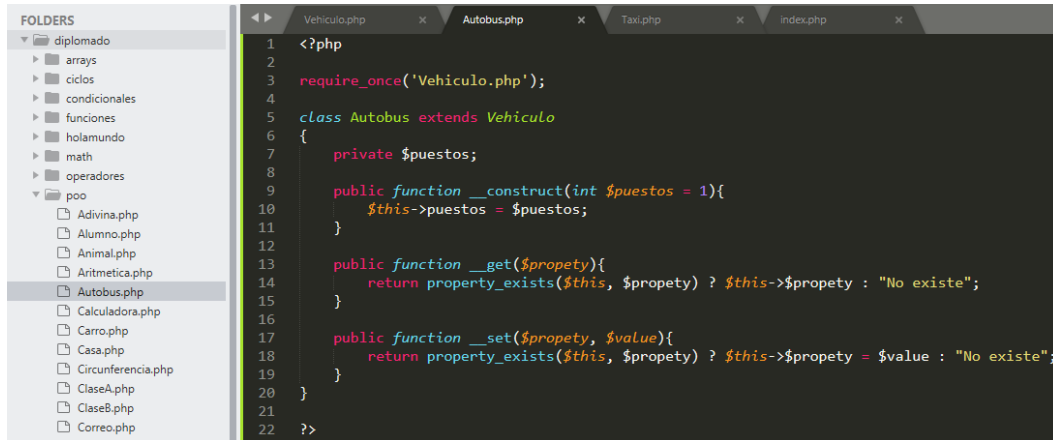
FOLDERS
└─ diplomado
  └─ arrays
  └─ ciclos
  └─ condicionales
  └─ funciones
  └─ holamundo
  └─ math
  └─ operadores
  └─ poo
    └─ Adivina.php
    └─ Alumno.php
    └─ Animal.php
    └─ Arimetica.php
    └─ Autobus.php
    └─ Calculadora.php
    └─ Carro.php
    └─ Casa.php
    └─ Circunferencia.php
    └─ ClaseA.php
    └─ ClaseB.php
    └─ Correo.php
    └─ Destruir.php
    └─ Facebook.php
    └─ Gato.php
    └─ index.php
    └─ main.php
    └─ Notificacion.php
    └─ Palabras.php
    └─ Perro.php
    └─ Persona.php
    └─ Servidor.php

Vehiculo.php
1 <?php
2
3 class Vehiculo
4 {
5     private $matricula;
6     private $modelo;
7     private $potencia;
8
9     public function __construct(string $matricula = "", string $modelo = "", string $potencia = "") {
10         $this->matricula = $matricula;
11         $this->modelo = $modelo;
12         $this->potencia = $potencia;
13     }
14
15     public function __get($property){
16         return property_exists($this, $property) ? $this->$property : "No existe";
17     }
18
19     public function __set($property, $value){
20         return property_exists($this, $property) ? $this->$property = $value : "No existe";
21     }
22
23     public function encenderVehiculo() : void {
24         echo "Vehiculo encendido <br>";
25     }
26
27     public function apagarVehiculo() : void {
28         echo "Vehiculo apagado <br>";
29     }
30 }
31
32 ?>
  
```

Ilustración 107.

La clase padre se encarga de englobar todas las características que ambas clases hijas tengan en común para su posterior herencia. Las clases hijas tendrían la siguiente estructura:

Estructura de la clase autobús:



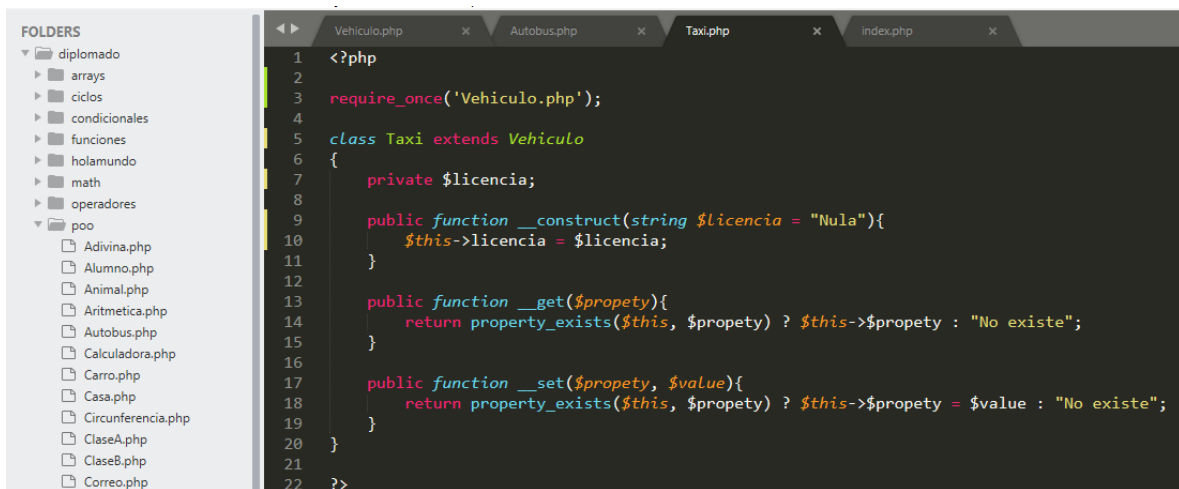
```

1 <?php
2
3 require_once('Vehiculo.php');
4
5 class Autobus extends Vehiculo
6 {
7     private $puestos;
8
9     public function __construct(int $puestos = 1){
10         $this->puestos = $puestos;
11     }
12
13     public function __get($property){
14         return property_exists($this, $property) ? $this->$property : "No existe";
15     }
16
17     public function __set($property, $value){
18         return property_exists($this, $property) ? $this->$property = $value : "No existe";
19     }
20 }
21
22 ?>

```

Ilustración 108.

Estructura de la clase taxi:



```

1 <?php
2
3 require_once('Vehiculo.php');
4
5 class Taxi extends Vehiculo
6 {
7     private $licencia;
8
9     public function __construct(string $licencia = "Nula"){
10         $this->licencia = $licencia;
11     }
12
13     public function __get($property){
14         return property_exists($this, $property) ? $this->$property : "No existe";
15     }
16
17     public function __set($property, $value){
18         return property_exists($this, $property) ? $this->$property = $value : "No existe";
19     }
20 }
21
22 ?>

```

Ilustración 109.

Con la estructura jerárquica de las tres clases descritas, ahora se puede operar entre ellas de la manera correcta y reutilizando código.

FOLDERS

- diplomado
 - arrays
 - ciclos
 - condicionales
 - funciones
 - holamundo
 - math
 - operadores
 - poo
 - Adivina.php
 - Alumno.php
 - Animal.php
 - Aritmetica.php
 - Autobus.php

Vehiculo.php

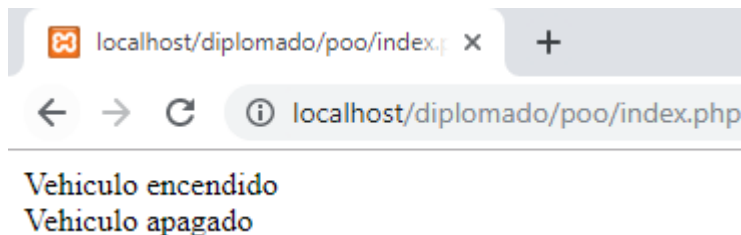
Autobus.php

Taxi.php

```

1 <?php
2
3 require_once('Taxi.php');
4
5 $TaxiAmarillito = new Taxi("LC015841");
6
7 $TaxiAmarillito->matricula = "USR921";
8 $TaxiAmarillito->modelo = "2005";
9 $TaxiAmarillito->potencia = "300 CV";
10
11 $TaxiAmarillito->encenderVehiculo();
12 $TaxiAmarillito->apagarVehiculo();
13
14 ?>

```



FOLDERS

- diplomado
 - arrays
 - ciclos
 - condicionales
 - funciones
 - holamundo
 - math
 - operadores
 - poo
 - Adivina.php
 - Alumno.php
 - Animal.php
 - Aritmetica.php
 - Autobus.php

Vehiculo.php

Autobus.php

Taxi.php

```

1 <?php
2
3 require_once('Autobus.php');
4
5 $Trasconor = new Autobus(22);
6
7 $Trasconor->matricula = "TRS021";
8 $Trasconor->modelo = "2010";
9 $Trasconor->potencia = "500 CV";
10
11 $Trasconor->encenderVehiculo();
12 $Trasconor->apagarVehiculo();
13
14 ?>

```

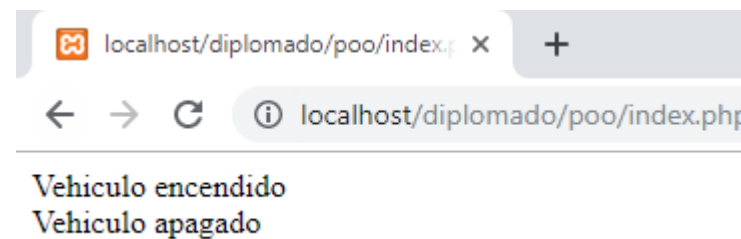


Ilustración 110, 111, 112 y 113.

Por medio de la clase padre no hay necesidad de crear métodos y atributos independientes en cada clase, dado que la herencia permite englobar aquellas características que se comparten en cuanto a la estructura y simplificarlas para ahorrar código posteriormente.

PHP permite múltiples niveles de herencia, pero no la herencia *múltiple*, es decir, una clase solo puede heredar directamente de una clase ascendiente. Por otro lado, una clase puede ser ascendiente de tantas clases descendiente como se desee (*un único padre, multitud de hijos*). En la siguiente figura se muestra gráficamente un ejemplo de jerarquía entre diferentes clases relacionadas mediante la herencia.

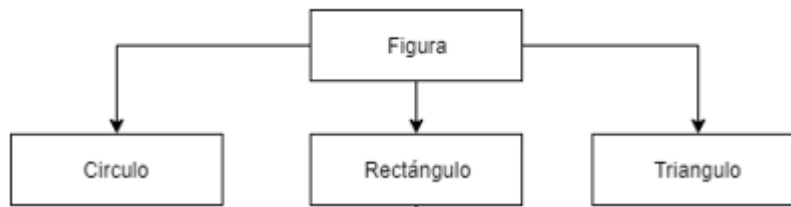


Ilustración 114.

Como se ha comentado anteriormente, la clase descendiente puede añadir sus propios atributos y métodos, pero también puede sustituir u ocultar los heredados. En concreto:

Se puede declarar un nuevo método de instancia con la misma cabecera que el de la clase ascendiente, lo que supone su sobrescritura. Por lo tanto, la sobrescritura o redefinición consiste en que métodos adicionales declarados en la clase descendiente con el mismo nombre, tipo de dato devuelto y número y tipo de parámetros sustituyen a los heredados.


```
class Vehiculo
{
    private $matricula;
    private $modelo;
    private $potencia;

    public function __construct(string $matricula = "", string $modelo = "", string $potencia = "") {
        $this->matricula = $matricula;
        $this->modelo = $modelo;
        $this->potencia = $potencia;
    }

    public function __get($property){
        return property_exists($this, $property) ? $this->$property : "No existe";
    }

    public function __set($property, $value){
        return property_exists($this, $property) ? $this->$property = $value : "No existe";
    }

    public function encenderVehiculo() : void {
        echo "Vehiculo encendido <br>";
    }

    public function apagarVehiculo() : void {
        echo "Vehiculo apagado <br>";
    }
}
```

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

class Autobus extends Vehiculo
{
    private $puestos;

    public function encenderVehiculo() : void {
        echo "Autobus encendido <br>";
    }

    public function __construct(int $puestos = 1){
        $this->puestos = $puestos;
    }

    public function __get($property){
        return property_exists($this, $property) ? $this->$property : "No existe";
    }

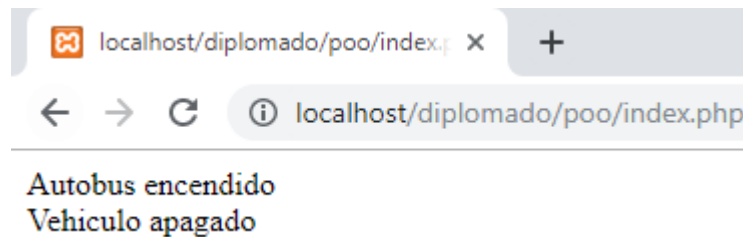
    public function __set($property, $value){
        return property_exists($this, $property) ? $this->$property = $value : "No existe";
    }
}
```

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

$Trasconor = new Autobus(22);

$Trasconor->matricula = "TRS021";
$Trasconor->modelo = "2010";
$Trasconor->potencia = "500 CV";

$Trasconor->encenderVehiculo();
$Trasconor->apagarVehiculo();
```



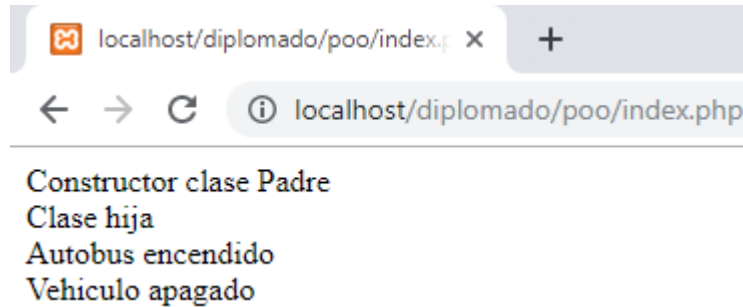
Ilustraciones 115, 116, 117 y 118.

De esta forma, la herencia se pierde al sobrescribir los métodos, en este caso `encenderVehiculo`, que ignora el componente de la herencia y se dirige al ámbito de la clase.

Se puede declarar un constructor de la subclase que llame al de la superclase de forma implícita o mediante la palabra reservada `parent::__construct`. Con `parent` hacemos el llamado a la clase padre y especificamos el método, en este caso el constructor.

```
public function __construct(string $matricula = "", string $modelo = "", string $potencia = "") {
    $this->matricula = $matricula;
    $this->modelo = $modelo;
    $this->potencia = $potencia;
    echo "Constructor clase Padre <br>";
}
```

```
public function __construct(int $puestos = 1){
    $this->puestos = $puestos;
    parent::__construct();
    echo "Clase hija <br>";
}
```



Ilustraciones 119, 120 y 121.

En general puede accederse a los métodos de la clase ascendiente que han sido redefinidos empleando la palabra reservada *parent* delante del método. Este mecanismo solo permite acceder al método perteneciente a la clase en el nivel inmediatamente superior de la jerarquía de clases.

Parent

Puede encontrarse escribiendo código que se refiera a variables y funciones en clases base (padres). En lugar de usar el nombre literal de la clase base en su código, debe usar la palabra reservada ***parent***, que se refiere al nombre de su clase base como se indica en la declaración extendida de su clase. Al hacer esto, evita usar el nombre de su clase base en más de un lugar. Si su árbol de herencia cambia durante la implementación, el cambio se realiza fácilmente al cambiar la declaración extendida de su clase.

De forma más clara, *parent* es una representación de una clase padre en herencia, que, por la funcionalidad de heredar, permite realizar llamados a métodos o variables de la clase padre sin usar el nombre la misma, de la siguiente forma:

```
public function __construct(int $puestos = 1){  
    $this->puestos = $puestos;  
    parent::__construct();  
    echo "Clase hija <br>";  
}
```

Ilustración 122.

En esta instrucción se realiza un llamado al constructor de la clase padre, pero si no se usase *parent*:

```
public function __construct(int $puestos = 1){  
    $this->puestos = $puestos;  
    Vehiculo::__construct();  
    echo "Clase hija <br>";  
}
```

Ilustración 123.

El resultado es igual. Pero en caso de que la clase cambiara de nombre en algún momento del proceso de codear, se tendría que cambiar en todos los lugares donde se realicen llamados a las clases padres.

Sobrescritura de métodos - Polimorfismo

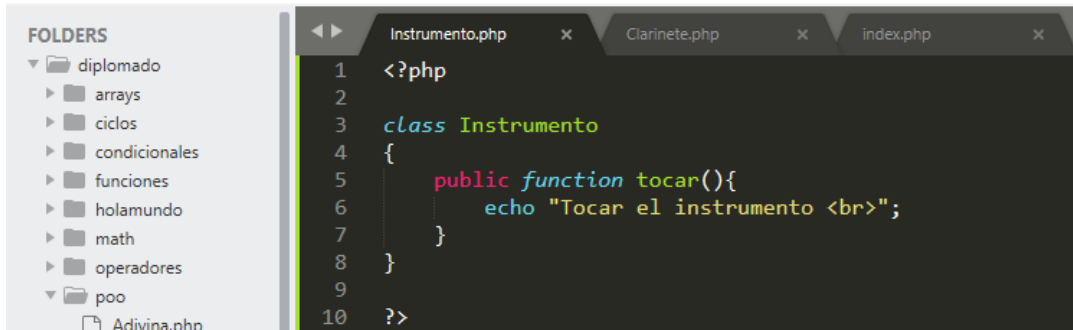
Es la forma por la cual una clase que hereda puede redefinir los métodos de su clase padre, de esta manera puede crear nuevos métodos con el mismo nombre de su superclase. Es decir, si existe una clase padre con el método *saludar ()*, se puede crear en la clase hija un método que también se llame *saludar ()*, pero con la implementación según la necesidad.

La característica principal que se debe tener muy presente cuando se trabaja con sobrescritura de métodos es:

- La estructura de los métodos debe ser igual en ambas clases: mismos parámetros, mismo tipo de retorno e implementación del mismo modificador de acceso.

Obsérvese el siguiente ejemplo para entender el funcionamiento de la sobrescritura de métodos:

Clase padre instrumento:



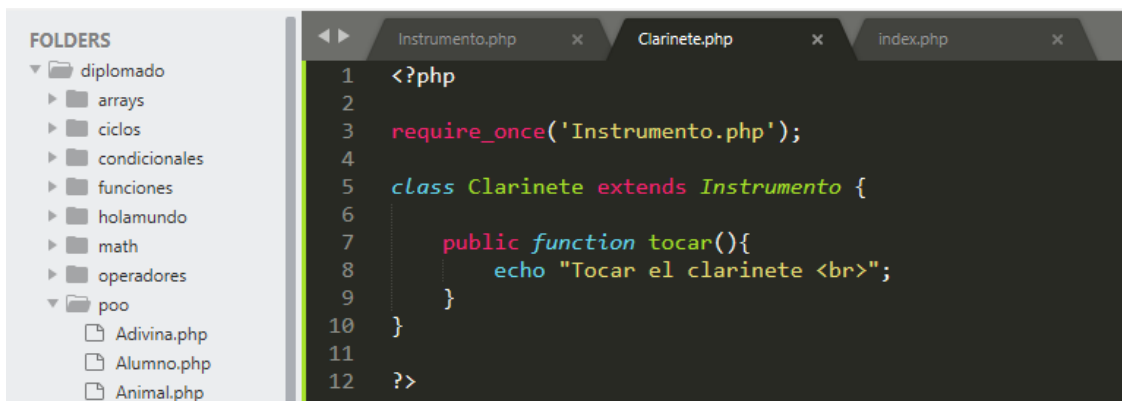
```

1  <?php
2
3  class Instrumento
4  {
5      public function tocar(){
6          echo "Tocar el instrumento <br>";
7      }
8  }
9
10 ?>

```

Ilustración 124.

Clase hija clarinete heredada de la clase padre instrumento:



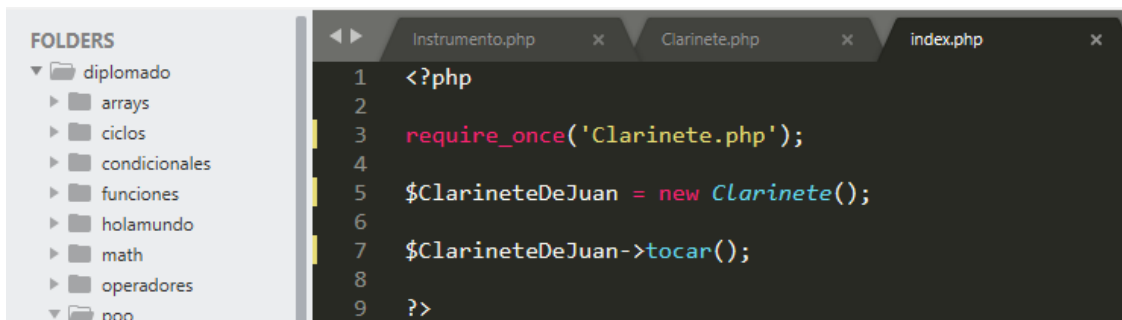
```

1  <?php
2
3  require_once('Instrumento.php');
4
5  class Clarinete extends Instrumento {
6
7      public function tocar(){
8          echo "Tocar el clarinete <br>";
9      }
10 }
11
12 ?>

```

Ilustración 125.

La clase padre implementa el método tocar, la clase hija lo sobrescribe, por lo que ahora son dos métodos diferentes (recordar el uso de la anotación). La estructura de ejecución sería la siguiente:



```

1  <?php
2
3  require_once('Clarinete.php');
4
5  $ClarineteDeJuan = new Clarinete();
6
7  $ClarineteDeJuan->tocar();
8
9  ?>

```

Ilustración 126.

Con base al objeto de tipo clarinete se accede al método tocar de la misma clase (no al padre, dado que ha sido sobrescrita). Obsérvese la salida.

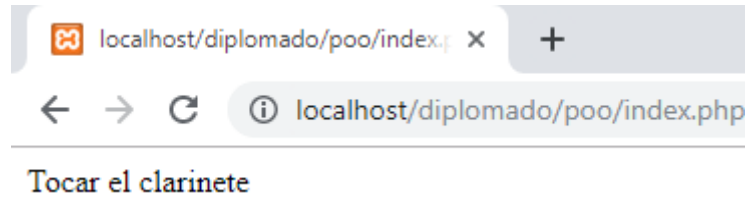
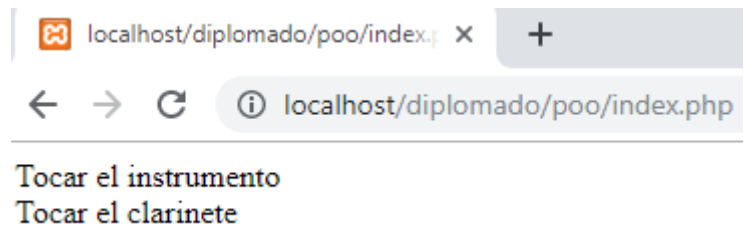
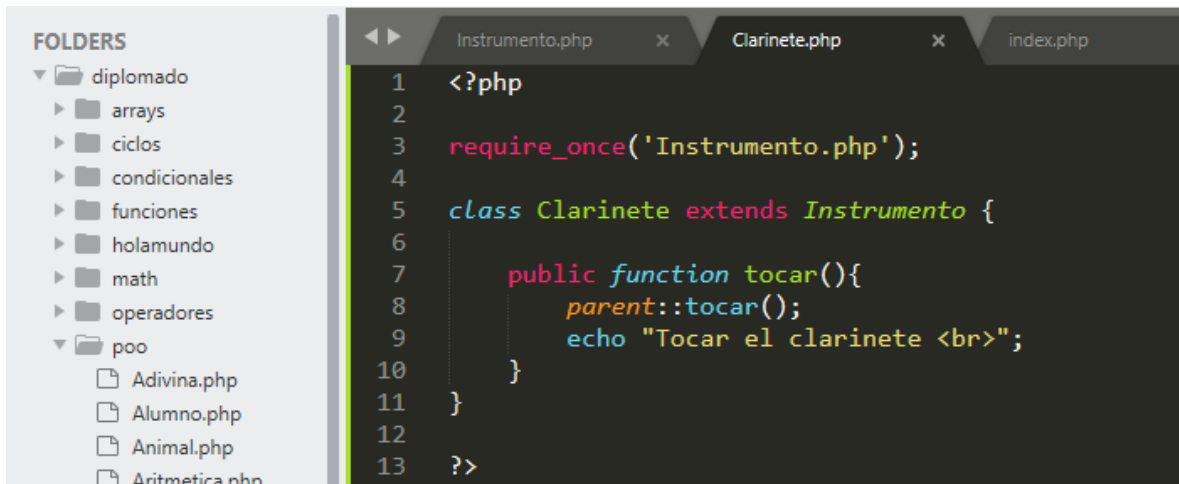


Ilustración 127.

Ya el objeto no accede al método de la **superclase** o clase **padre**, sino que directamente accede al declarado en su propia estructura de clase. Si se quisiera obtener la ejecución de ambos métodos, tanto el de la clase **padre** e **hija**, simplemente se tendría que cambiar la siguiente estructura:

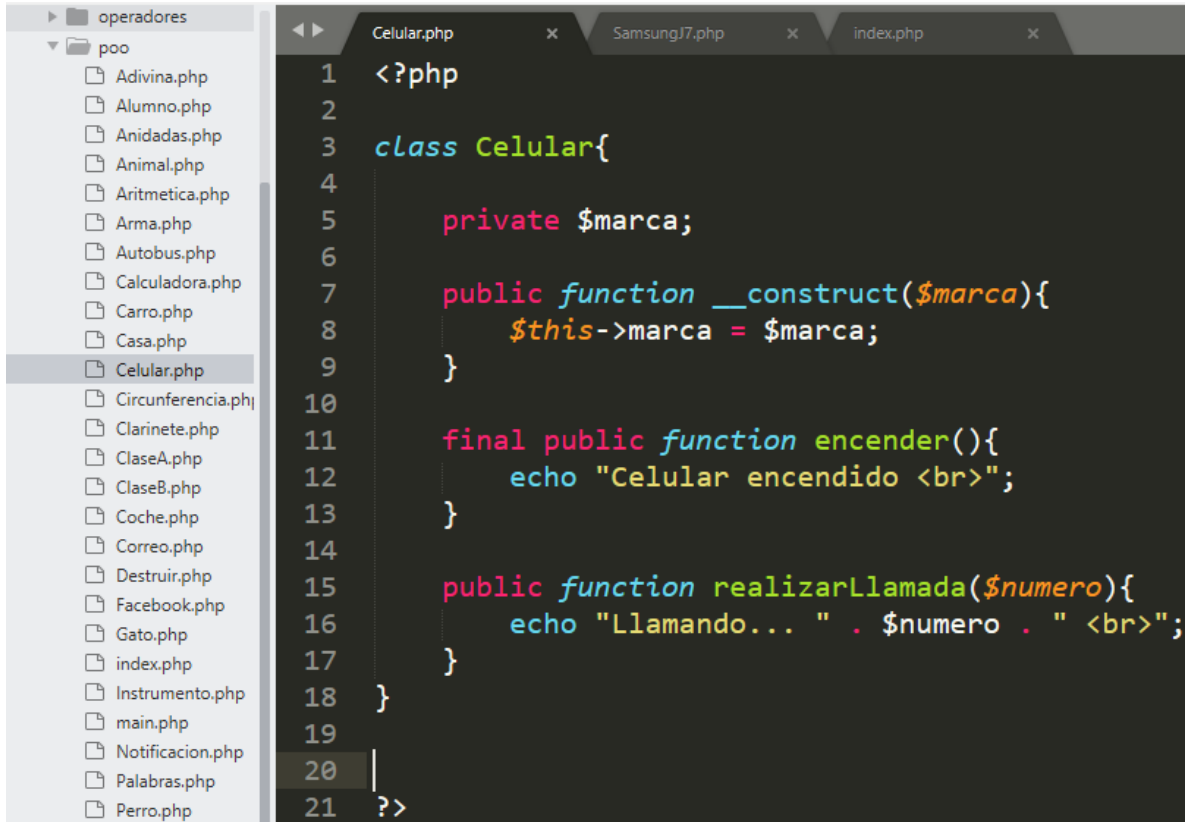


Ilustraciones 128 Y 129.

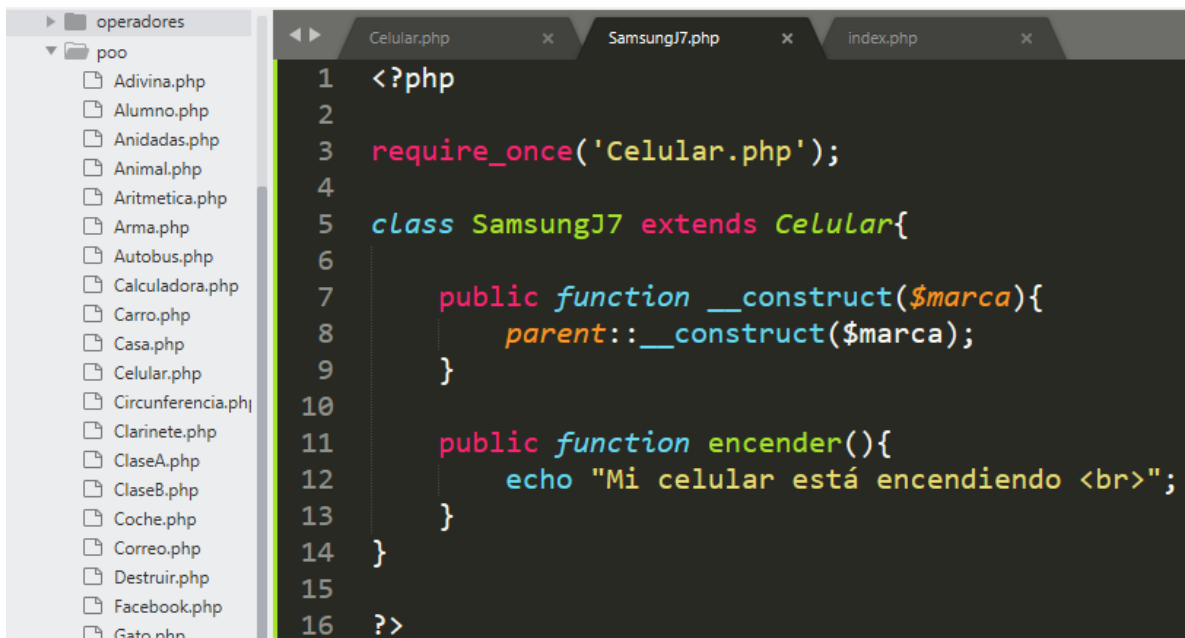
La palabra **parent** hace el llamado al método **tocar** de la superclase, haces las veces de **this**, pero esta ya no hace referencia al objeto o clase actual, sino a su clase padre, en este caso: instrumento.

Final

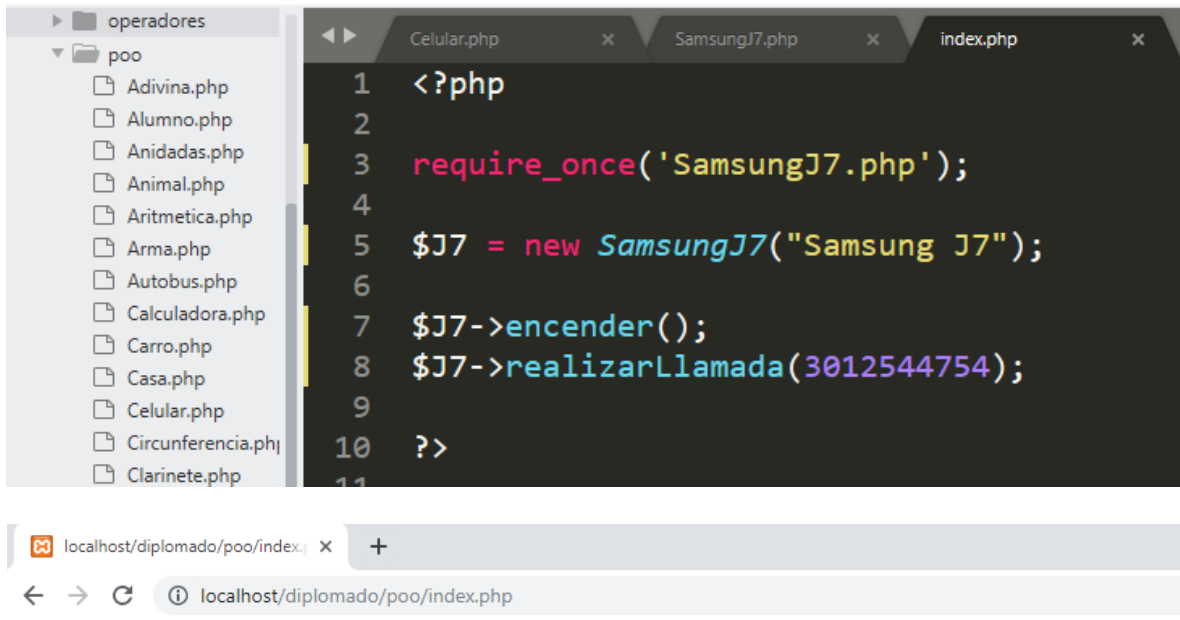
PHP introduce la nueva palabra clave final, que impide que las clases hijas sobrescriban un método, antecediendo su definición con final. Si la propia clase se define como final, entonces no se podrá heredar de ella.



```
1 <?php
2
3 class Celular{
4     private $marca;
5
6     public function __construct($marca){
7         $this->marca = $marca;
8     }
9
10
11     final public function encender(){
12         echo "Celular encendido <br>";
13     }
14
15     public function realizarLlamada($numero){
16         echo "Llamando... " . $numero . " <br>";
17     }
18 }
19
20
21 ?>
```



```
1 <?php
2
3 require_once('Celular.php');
4
5 class SamsungJ7 extends Celular{
6
7     public function __construct($marca){
8         parent::__construct($marca);
9     }
10
11     public function encender(){
12         echo "Mi celular está encendiendo <br>";
13     }
14 }
15
16 ?>
```



The screenshot shows a code editor with a file explorer on the left listing files in a 'poo' directory. The main editor window shows the code in 'index.php':

```

1 <?php
2
3 require_once('SamsungJ7.php');
4
5 $J7 = new SamsungJ7("Samsung J7");
6
7 $J7->encender();
8 $J7->realizarLlamada(3012544754);
9
10 ?>
11

```

Below the code editor, a browser window shows the error message:

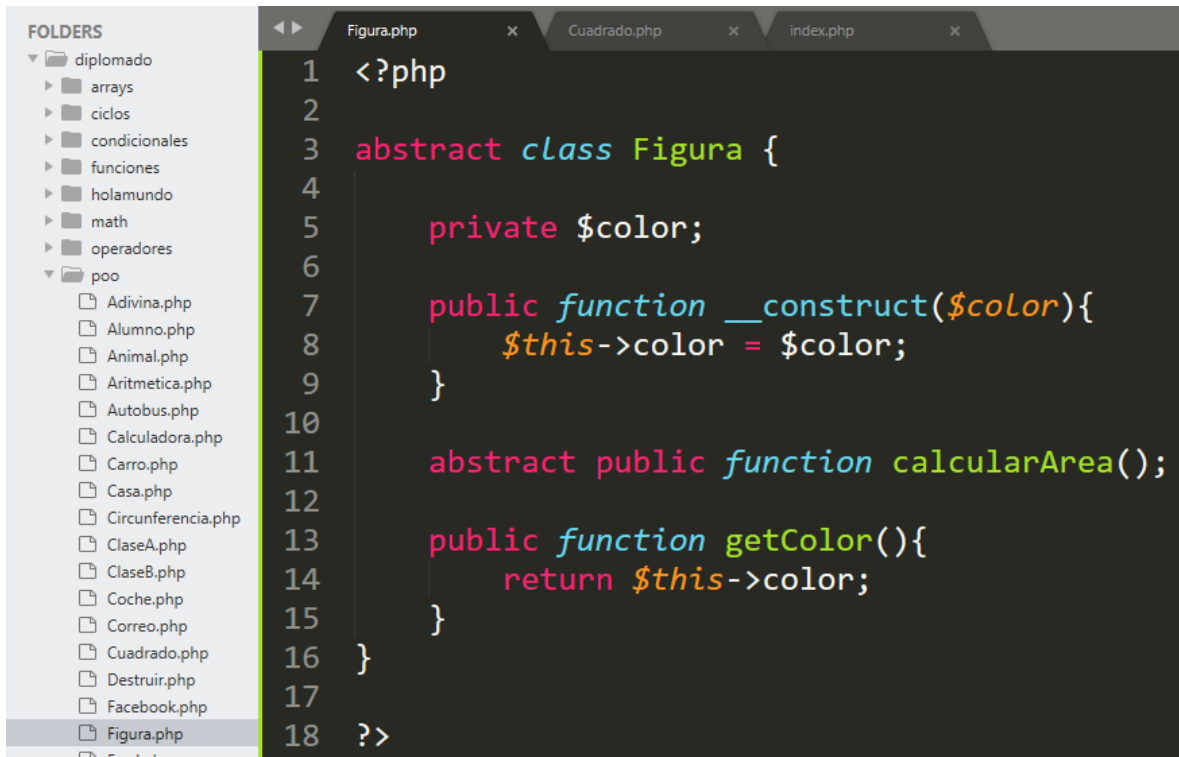
Fatal error: Cannot override final method Celular::encender() in C:\xampp\htdocs\diplomado\poo\SamsungJ7.php on line 5

Ilustraciones 130, 131, 132 y 133.

Clases abstractas

En PHP se dice que son clases abstractas aquellas clases base (superclases) de las que no se permite la creación de objetos. Para ello, se utiliza la palabra clave *abstract*.

En una clase abstracta es posible definir métodos abstractos, los cuales se caracterizan por el hecho de que no pueden ser implementados en la clase base. De ellos, solo se escribe su signatura en la superclase, y su funcionalidad —polimórfica— tiene que indicarse en las clases derivadas (subclases).



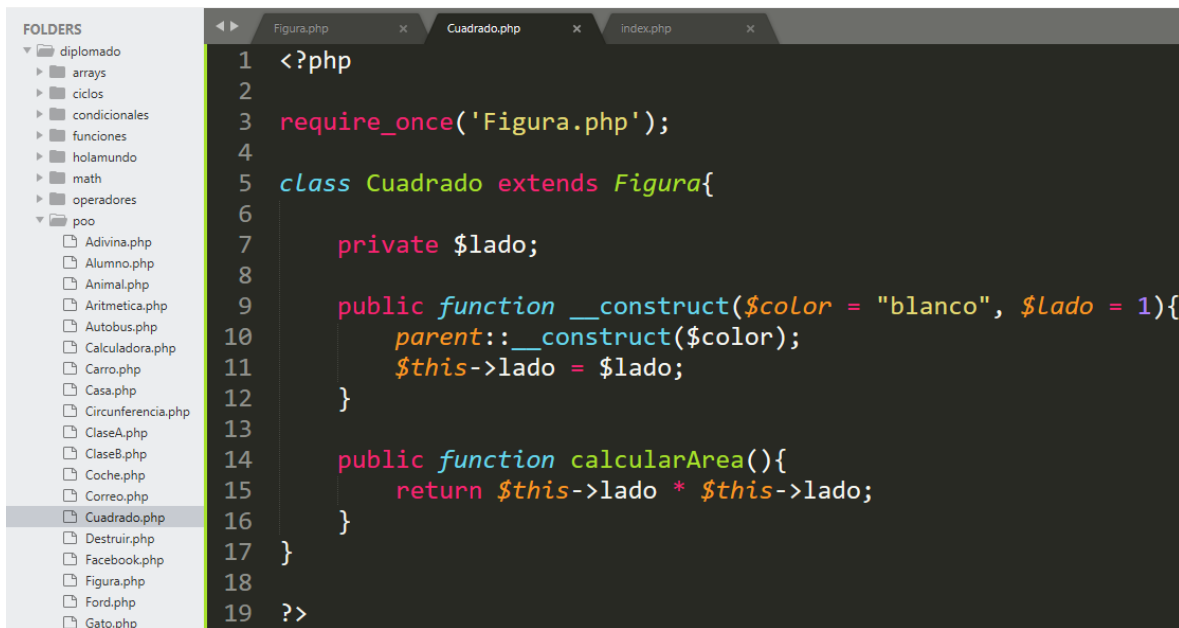
```

1  <?php
2
3  abstract class Figura {
4
5      private $color;
6
7      public function __construct($color){
8          $this->color = $color;
9      }
10
11     abstract public function calcularArea();
12
13     public function getColor(){
14         return $this->color;
15     }
16 }
17
18 ?>

```

Ilustración 134.

En la clase **figura** se ha definido un atributo (**color**), un constructor y dos métodos (**calcularArea** y **getColor**), **calcularArea** es abstracto, por lo que no cuenta con implementación.



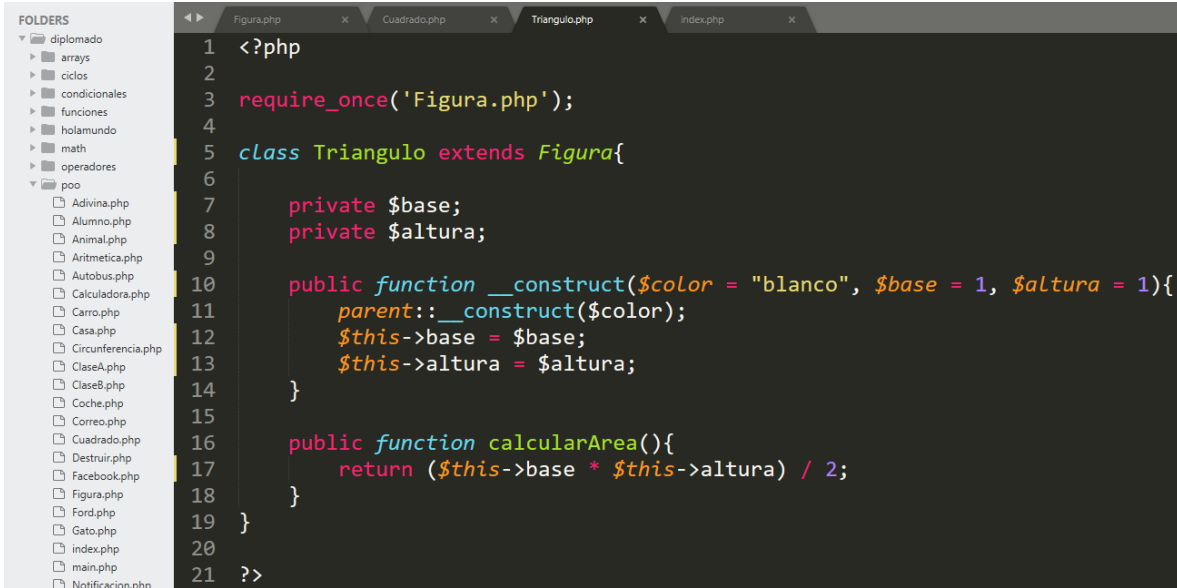
```

1  <?php
2
3  require_once('Figura.php');
4
5  class Cuadrado extends Figura{
6
7      private $lado;
8
9      public function __construct($color = "blanco", $lado = 1){
10         parent::__construct($color);
11         $this->lado = $lado;
12     }
13
14     public function calcularArea(){
15         return $this->lado * $this->lado;
16     }
17 }
18
19 ?>

```

Ilustración 135.

En la clase **cuadrado** se ha definido un atributo (**lado**), un constructor y un método (**calcularArea**).



```

1  <?php
2
3  require_once('Figura.php');
4
5  class Triangulo extends Figura{
6
7      private $base;
8      private $altura;
9
10     public function __construct($color = "blanco", $base = 1, $altura = 1){
11         parent::__construct($color);
12         $this->base = $base;
13         $this->altura = $altura;
14     }
15
16     public function calcularArea(){
17         return ($this->base * $this->altura) / 2;
18     }
19 }
20
21 ?>

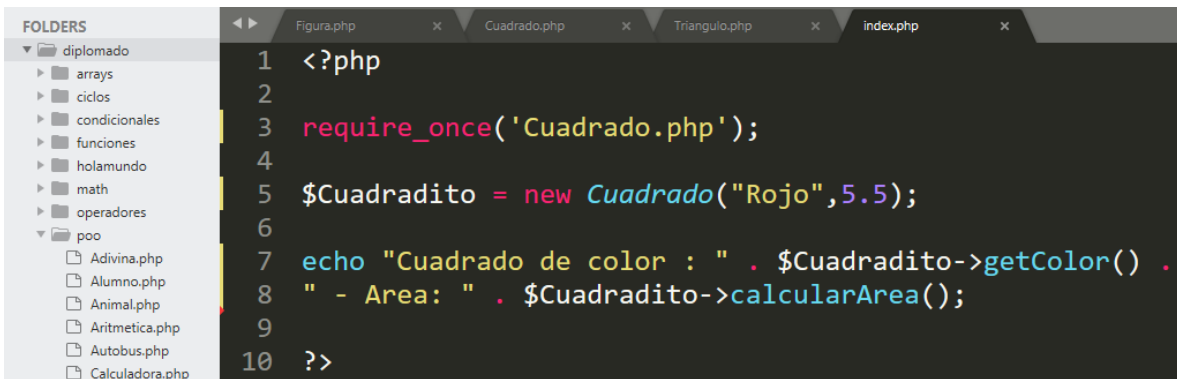
```

Ilustración 136.

En la clase **triángulo** se han definido dos atributos (**base** y **altura**), un constructor y un método (**calcularArea**).

Como se puede observar, el método **calcularArea** ha sido definido abstracto (**abstract**) en la superclase abstracta figura, indicándose solamente su signatura.

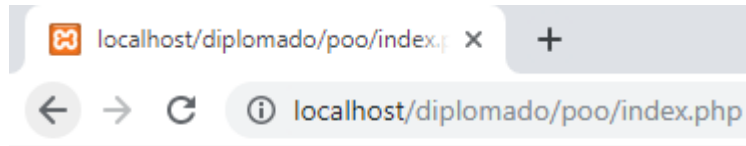
Por otro lado, véase que en cada una de las subclases (cuadrado y triángulo) se ha implementado dicho método.



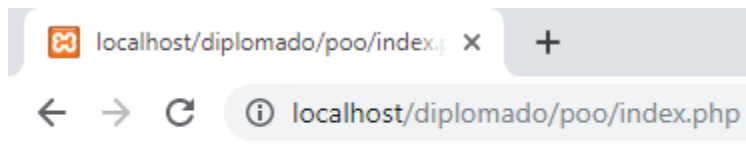
```

1  <?php
2
3  require_once('Cuadrado.php');
4
5  $Cuadradito = new Cuadrado("Rojo",5.5);
6
7  echo "Cuadrado de color : " . $Cuadradito->getColor() .
8  " - Area: " . $Cuadradito->calcularArea();
9
10 ?>

```



Cuadrado de color : Rojo - Area: 30.25



Triangulo de color : Verde - Area: 22.5

Ilustraciones 137, 138, 139 y 140.

Fíjese que en la ejecución de ambas clases se ha invocado al método `getColor`, definido e implementado en la superclase **figura**. Sin embargo, el método abstracto **calcularArea** únicamente se implementa en las subclases (**cuadrado** y **triángulo**).

Interfaces

Las interfaces de objetos permiten crear código con el cual especificar qué métodos deben ser implementados por una clase, sin tener que definir cómo estos métodos son manipulados.

Las interfaces se definen de la misma manera que una clase, aunque reemplazando la palabra reservada **class** por la palabra reservada **interface**, y sin que ninguno de sus métodos tenga su contenido definido.

Todos los métodos declarados en una interfaz deben ser públicos, ya que esta es la naturaleza de una interfaz.



```

1  <?php
2
3  interface Arma{
4
5      public function recargar();
6
7      public function disparar($cantidad);
8
9      public function cambiar($arma);
10 }
11
12 ?>

```

Ilustración 141.

Para implementar una interfaz se utiliza el operador *implements*. Todos los métodos en una interfaz deben ser implementados dentro de la clase.



```

1  <?php
2
3  require_once('Arma.php');
4
5  class Pistola implements Arma{
6
7      private $cantidad;
8
9      public function recargar(){
10         echo "El arma está cargada <br>";
11     }
12
13     public function disparar($cantidad){
14         echo "Disparaste " . $cantidad . " tiros <br>";
15     }
16
17     public function cambiar($arma){
18         echo "Cambiaste tu arma por: " . $arma . "<br>";
19     }
20 }
21
22 ?>

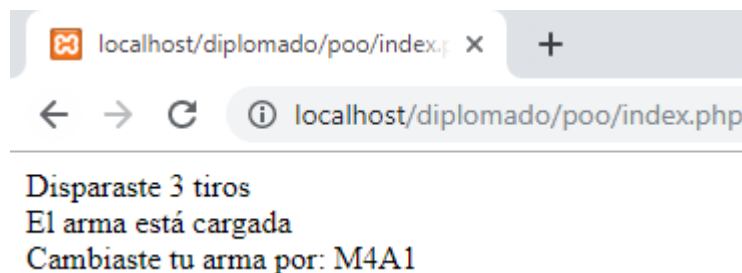
```

FOLDERS

- diplomado
 - arrays
 - ciclos
 - condicionales
 - funciones
 - holamundo
 - math
 - operadores
 - poo
 - Adivina.php
 - Alumno.php
 - Anidadas.php
 - Animal.php

```

1  <?php
2
3  require_once('Pistola.php');
4
5  $Pistolita = new Pistola();
6
7  $Pistolita->disparar(3);
8  $Pistolita->recargar();
9  $Pistolita->cambiar("M4A1");
10
11  ?>
            
```



Ilustraciones 142, 143 y 144.

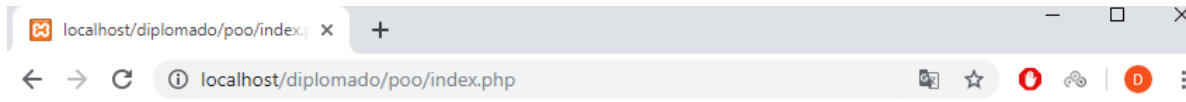
El no cumplir con esta regla resultará en un error fatal.

FOLDERS

- diplomado
 - arrays
 - ciclos
 - condicionales
 - funciones
 - holamundo
 - math
 - operadores
 - poo
 - Adivina.php
 - Alumno.php
 - Anidadas.php
 - Animal.php
 - Aritmetica.php
 - Arma.php
 - Autobus.php
 - Calculadora.php
 - Carro.php
 - Casa.php
 - Circunferencia.php
 - Clarinete.php

```

1  <?php
2
3  require_once('Arma.php');
4
5  class Pistola implements Arma{
6
7      private $cantidad;
8
9      public function disparar($cantidad){
10         echo "Disparaste " . $cantidad . " tiros <br>";
11     }
12
13     public function cambiar($arma){
14         echo "Cambiaste tu arma por: " . $arma . "<br>";
15     }
16 }
17
18 ?>
            
```



Ilustraciones 145 y 146.

El error especifica la ausencia de un método y además resalta cuál es el método que no se encuentra.

Las clases pueden implementar más de una interfaz si se deseara, separándolas cada una por una coma.



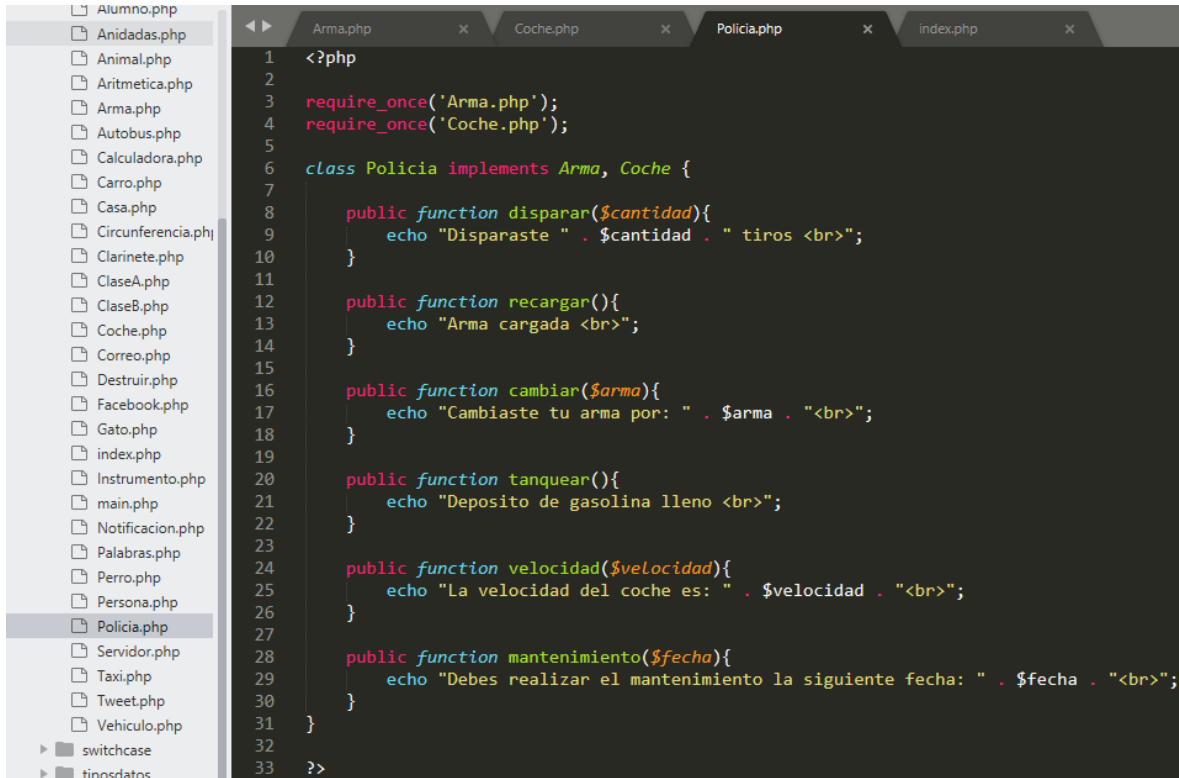
The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer lists various PHP files, including 'Arma.php'. The code editor shows the content of 'Arma.php', which defines an interface named 'Arma' with three methods: 'recargar()', 'disparar(\$cantidad)', and 'cambiar(\$arma)'.

```
1 <?php
2
3 interface Arma{
4
5     public function recargar();
6
7     public function disparar($cantidad);
8
9     public function cambiar($arma);
10 }
11
12 ?>
```



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer lists various PHP files, including 'Coche.php'. The code editor shows the content of 'Coche.php', which defines an interface named 'Coche' with three methods: 'tanquear()', 'velocidad(\$velocidad)', and 'mantenimiento(\$fecha)'.

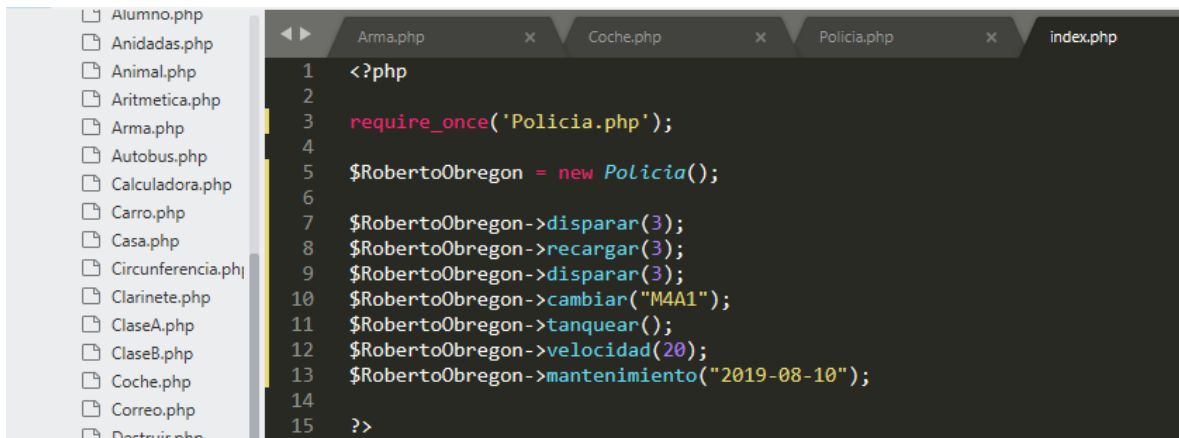
```
1 <?php
2
3 interface Coche{
4
5     public function tanquear();
6
7     public function velocidad($velocidad);
8
9     public function mantenimiento($fecha);
10 }
11
12 ?>
```



```

1  <?php
2
3  require_once('Arma.php');
4  require_once('Coche.php');
5
6  class Policia implements Arma, Coche {
7
8      public function disparar($cantidad){
9          echo "Disparaste " . $cantidad . " tiros <br>";
10     }
11
12     public function recargar(){
13         echo "Arma cargada <br>";
14     }
15
16     public function cambiar($arma){
17         echo "Cambiaste tu arma por: " . $arma . "<br>";
18     }
19
20     public function tanquear(){
21         echo "Deposito de gasolina lleno <br>";
22     }
23
24     public function velocidad($velocidad){
25         echo "La velocidad del coche es: " . $velocidad . "<br>";
26     }
27
28     public function mantenimiento($fecha){
29         echo "Debes realizar el mantenimiento la siguiente fecha: " . $fecha . "<br>";
30     }
31 }
32
33 ?>

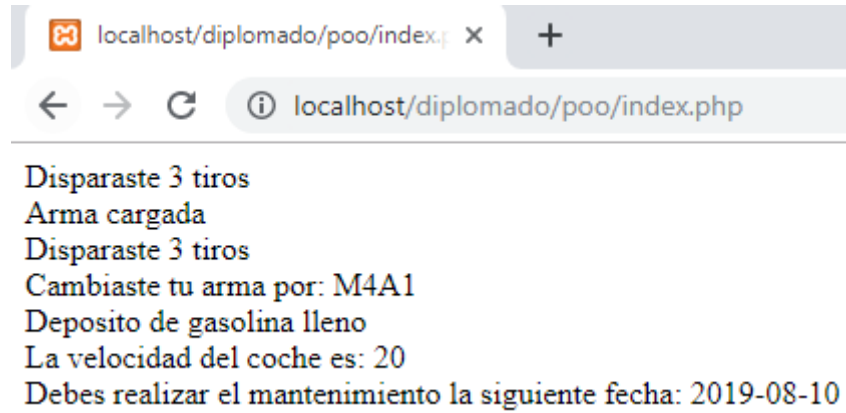
```



```

1  <?php
2
3  require_once('Policia.php');
4
5  $RobertoObregon = new Policia();
6
7  $RobertoObregon->disparar(3);
8  $RobertoObregon->recargar(3);
9  $RobertoObregon->disparar(3);
10 $RobertoObregon->cambiar("M4A1");
11 $RobertoObregon->tanquear();
12 $RobertoObregon->velocidad(20);
13 $RobertoObregon->mantenimiento("2019-08-10");
14
15 ?>

```



Ilustraciones 147, 148, 149, 150 y 151.

En este ejemplo se realiza la implementación de dos interfaces en una sola clase con los respectivos métodos que se adhieren en la implementación.


Es posible tener constantes dentro de las interfaces. Las constantes de interfaces funcionan como las constantes de clases, excepto porque no pueden ser sobrescritas por una clase/interfaz que las herede.

Las interfaces se suelen utilizar cuando se tienen muchas clases que tienen en común un comportamiento, pudiendo asegurar así que ciertos métodos estén disponibles en cualquiera de los objetos que queramos crear. Son especialmente importantes para la arquitectura de aplicaciones complejas.



Referencias bibliográficas

Lázaro, D. (2018). *Métodos mágicos en PHP*. <https://diego.com.es/metodos-magicos-en-php>



Esta guía fue elaborada para ser utilizada con fines didácticos como material de consulta de los participantes en el diplomado virtual en PROGRAMACIÓN EN PHP del Politécnico de Colombia, y solo podrá ser reproducida con esos fines. Por lo tanto, se agradece a los usuarios referirla en los escritos donde se utilice la información que aquí se presenta.

GUÍA DIDÁCTICA 3

M2-DV59-GU03

MÓDULO 3: PROGRAMACIÓN ORIENTADA A OBJETOS [POO]

© DERECHOS RESERVADOS - POLITÉCNICO DE COLOMBIA, 2023
Medellín, Colombia

Proceso: Gestión Académica Virtual
Realización del texto: Diego Palacio, docente
Revisión del texto: Comité de Revisión
Diseño: Comunicaciones

Editado por el Politécnico de Colombia.