

CURSO .PHP

PROGRAMACIÓN ORIENTADA A OBJETOS

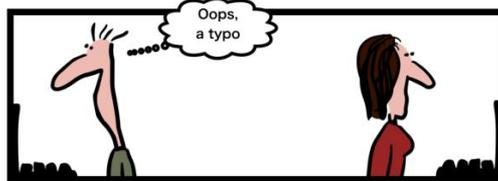
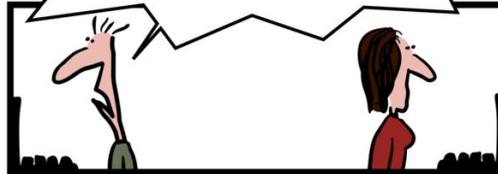


Autor: Jon Vadillo
www.jonvadillo.com

Contenidos

- Fundamentos básicos
- Definir una clase
- Crear una instancia
- Modificadores
- Herencia
- Clases abstractas
- static
- Interfaces
- Excepciones

This is UN-BE-LIEV-A-BLE!!!
This is definitely
a bug in the compiler or in the OS.

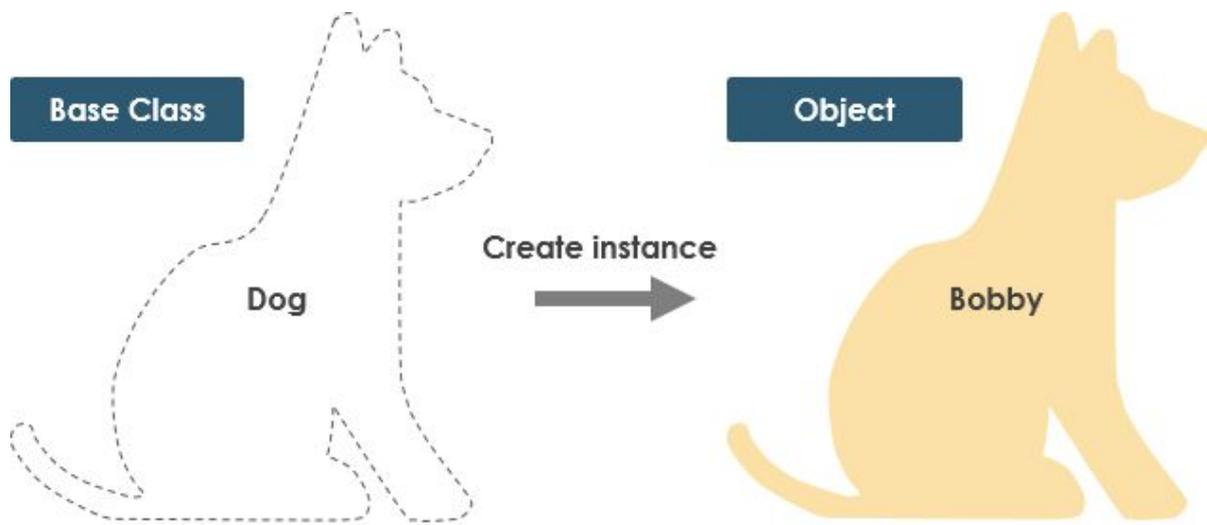


Just happened

Source: <http://geek-and-poke.com/>

Clase

- Concepto utilizado para describir entidades (objetos).
- Están compuestos por:
 - **Propiedades**: definen cómo son los objetos y su estado.
 - **Métodos**: definen el comportamiento, lo que pueden hacer los objetos de esa clase.
- Cada **objeto** creado de una clase se conoce como **instancia**.



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

Source: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/>

Clase

- El constructor se define con el método `__construct()`
 - Siempre será invocado al crear una instancia. Sólo puede existir uno.
- Las propiedades (atributos) se declaran privados (solo accesibles desde dentro) y los métodos públicos.
- Para crear una instancia se utiliza la palabra reservada `new`
- `$this` hace referencia a la propia instancia creada.
- Una buena práctica es **solo crear una clase por archivo** (tendrá el mismo nombre que la clase y extensión .php)

Definir una clase

```
class Persona {  
  
    private $nombre;  
    private string $apellido;  
  
    // constructor  
    public function __construct($nombre, $apellido) {  
        $this->nombre = $nombre;  
        $this->apellido = $apellido;  
    }  
  
    public function saludar() {  
        echo "Hola, me llamo " . $this->nombre . " " . $this->apellido;  
    }  
}  
  
$manu = new Persona("Manu", "García");  
$manu->saludar();
```

Crear una instancia empleando un String

```
$nombreDeClase = 'Persona';  
  
$objeto = new Persona; // new Persona  
$objeto = new $nombreDeClase; // new Persona  
$objeto = new $nombreDeClase(); // new Persona
```

Nota: Si no hay argumentos para pasar al constructor de la clase, se pueden omitir los paréntesis después del nombre de la clase.

Declarar el tipo de las variables (PHP > 7.4)

```
class Persona {  
  
    public string $nombre;  
    public int $edad;  
  
    public function setNombre(string $nombre):void {  
        $this->nombre = $nombre;  
    }  
    public function setEdad(int|float $edad):void {  
        $this->edad = $edad;  
    }  
    public function foo(?int $test):void {  
        var_dump($test);  
    }  
}  
$p = new Persona;  
$p->nombre = "Jon";  
var_dump($p->nombre); // string(3) "Jon"  
var_dump($p->edad); // Fatal error. Es necesario inicializarla antes.
```

Hands on!



Ejercicio 1

- Crea una clase «Publicacion» con las siguientes propiedades: «autores», «año», «editorial», «título» y «texto».
- Añade un constructor y los siguientes métodos:
 - Leer: devuelve el contenido de la propiedad texto.
 - Escribir: recibe un String y lo concatena a la propiedad texto.
- A continuación instancia dos objetos distintos y llama a sus métodos.

Modificadores

- **Public:** cualquiera puede acceder a la variable. Es el valor por defecto.
- **Private:** solo accesible desde la clase que los declara.
- **Protected:** accesible desde la clase que los declara o sus descendientes.
- **Final:** sus descendientes no pueden sobrescribir el valor.
- **Abstract:** solo se puede utilizar una vez se ha definido en la subclase.

Modificadores

```
class Persona
{
    public $nombre = 'Nombre Public';
    protected $apellido = 'Apellido Protected';
    private $edad = 'Edad Private';

    function pruebaModificadores()
    {
        echo $this->nombre;
        echo $this->apellido;
        echo $this->edad;
    }
}
```

```
$persona = new Persona();
echo $persona->nombre; // OK
echo $persona->apellido; // Error
echo $persona->edad; // Error
$persona->pruebaModificadores(); // OK
```

readonly (a partir de PHP 8.1)

```
<?php

class Test {
    public readonly string $prop;

    public function __construct(string $prop) {
        // Legal initialization.
        $this->prop = $prop;
    }
}

$test = new Test("foobar");
// Legal read.
var_dump($test->prop); // string(6) "foobar"

// Illegal reassignment. It does not matter that the assigned value is the same.
$test->prop = "foobar";
// Error: Cannot modify readonly property Test::$prop
?>
```

Getters & Setters

- Se recomienda utilizar **getters** y **setters** explícitos para propiedades en el objeto en lugar de acceder directamente a ellos.
- Ventajas:
 - **Consistencia**: puede cambiar la implementación 'interna' sin tener que modificar ninguna llamada externa.
 - **Control**: especifica qué propiedades pueden ser utilizadas / llamadas desde fuera y aplica una lógica de validación.

Getters & Setters

```
class Persona
{
    private $edad = 9;

    function getEdad()
    {
        return $this->edad;
    }
    function setEdad($edad)
    {
        $this->edad = ($edad > 0) ? $edad : 0;
    }
}
```

```
$persona = new Persona();
$persona->setEdad(8);
echo $persona->getEdad; // edad = 8

$persona->setEdad(-5);
echo $persona->getEdad; // edad = 0
```

Espacios de nombres



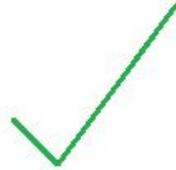
```
Class demoClass
{
}

class demoClass
{
}
```

Can not have same name classes in one project

©www.besthinditutorials.com

All classes at one place



namespace 1

```
class demoClass
{
}
```

namespace 2

```
class demoClass
{
}
```

Classes in different different namespace can not cause error

Espacios de nombres

- En aplicaciones que utilicen varias librerías, pueden coincidir nombres de clases y causar problemas.
- Los espacios de nombres (**Namespaces**) resuelven el problema.
- Un namespace es un contenedor abstracto que agrupa clases, funciones, constantes, etc.

Espacios de nombres

Coche.php

```
namespace MiApp\Modelo;  
  
class Coche {  
    //codigo de la clase  
}
```

index.php

```
include 'MiApp/Modelo/Coche.php';  
  
$coche1 = new MiApp\Modelo\Coche;
```

Nota: es una buena práctica utilizar como namespace el directorio donde está la clase.

use

Coche.php

```
namespace MiApp\Modelo;

class Coche {
    //codigo de la clase
}
```

index.php

```
include 'MiApp/Modelo/Coche.php' ;

use MiApp\Modelo\Coche;

$coche1 = new Coche;
```

Hands on!

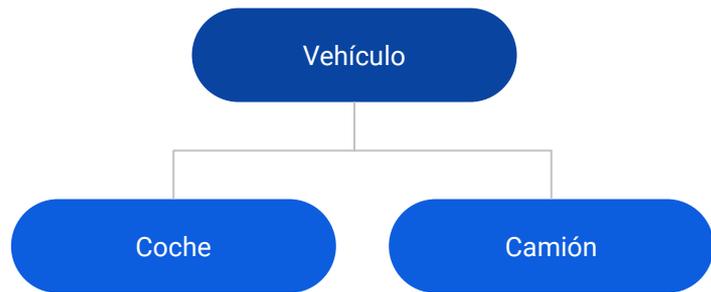


Ejercicio 2

- Partiendo de la clase `Publicación` creada en el ejercicio anterior:
 - Crea una carpeta llamada “models” y mete dentro la clase `Publicación`.
 - Crea un namespace
 - Utiliza la clase desde otro archivo llamado `index.php`

Herencia

- Permite **reutilizar** una clase ya definida y extenderla.
- Una clase sólo puede heredar de una única clase.
- Se utiliza la palabra **extends** seguida de la clase a heredar.



Herencia

```
class Persona {  
  
    private $nombre;  
  
    // constructor  
    public function __construct($nombre) {  
        $this->nombre = $nombre;  
    }  
  
    public function saludar() {  
        echo "Hola, me llamo " . $this->nombre;  
    }  
}
```

```
class Estudiante extends Persona {  
    public function estudiar(){  
        echo "Estoy estudiando";  
    }  
}  
  
// Instancia de la subclase  
$obj = new Estudiante("Mikel");  
$obj->saludar();  
$obj->estudiar();
```

parent::

- Las propiedades y los métodos heredados pueden ser **sobreescritos** (excepto los definidos como final).
 - Para ello basta con declararlos en la subclase con el mismo nombre.
 - Es posible acceder a los métodos sobreescritos utilizando la palabra reservada **parent::**

```
class Estudiante extends Persona
{
    // Sobreescribir el método
    function saludar() {
        echo "Soy un estudiante\n";
        parent::saludar();
    }
}

$estudiante = new Estudiante();
$estudiante->saludar();
```

Hands on!



Ejercicio 3

- Crea una clase llamada `Poligono` con 3 variables (`color`, `altura` y `anchura`), cada una con sus `getters` y `setters`. A continuación crea dos subclases de `Polígono` llamadas `Triángulo` y `Cuadrado`. Ambas tendrán un método llamado `area()` que calculará su área.

Clases abstractas

- **No se pueden instanciar**, se instancian las subclases.
- Toda clase que contenga un método abstracto deberá estar definida como clase abstracta, utilizando la palabra **abstract**.
- Las clases abstractas también pueden contener métodos comunes.
- Los métodos abstractos no pueden estar implementados, solo se declara la firma.
- Las subclases deben implementar **todos** los métodos abstractos.

Hands on!



Ejercicio 4

- Convierte la clase Poligono del ejemplo anterior en clase abstracta y añade el método abstracto `area()`. ¿Qué ventaja obtenemos frente a la implementación anterior?

static

- Un método o propiedad declarado como estático es **accesible aunque no exista ninguna instancia** de la clase.
- **Métodos estáticos:**
 - Se ejecutan sobre la clase, no en instancias creadas.
 - No pueden emplear la pseudovariable **\$this**
- **Propiedades estáticas:**
 - Solo existe una copia para la clase, no pertenece al objeto.
 - No se pueden acceder mediante ->

static

```
class Persona {  
    public static $nombre = "Nombre Estático";  
    public static function mostrarNombreEstatico () {  
        return self::$nombre;  
    }  
}  
  
class Alumno extends Persona {  
    public function mostrarMiNombreEstaticoDeNuevo () {  
        return parent::$nombre;  
    }  
}
```

static

```
//Todas las formas son equivalentes:  
  
echo Persona::mostrarNombreEstatico ();  
  
echo Persona::$nombre;  
  
$clase = "Persona";  
echo $clase::mostrarNombreEstatico ();  
  
$miclase = new Alumno;  
$miclase->mostrarMiNombreEstaticoDeNuevo ();
```

Hands on!



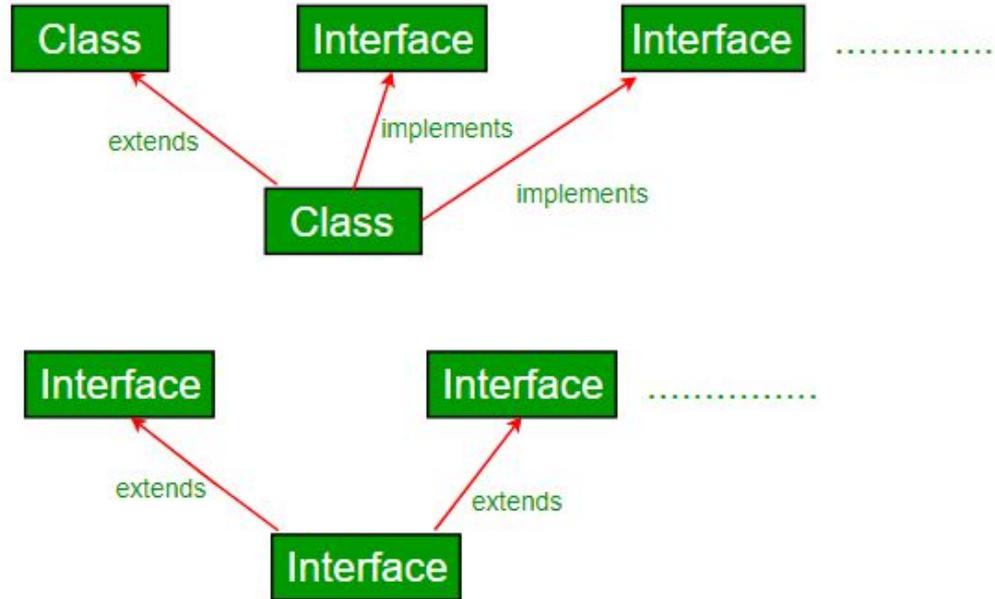
Ejercicio 5

- Crea una clase llamada Persona con una variable estática llamada “personasEnElMundo” e inicializada a cero. El constructor deberá incrementar la variable cada vez que se crea un objeto. Crea 5 objetos y a continuación muestra la variable estática por pantalla.

Interfaces

- Una interfaz (**interface**) es un “acuerdo” que debe cumplir la clase que lo implemente. Contiene la **declaración de métodos** que las clases tendrán que implementar.
- También **puede contener constantes** (no podrán ser sobrescritas).
- La clase que implemente una interface tendrá que contener todos sus métodos. Los métodos definidos serán todos **públicos**.
- Una clase puede implementar **múltiples interfaces**.

Interfaces

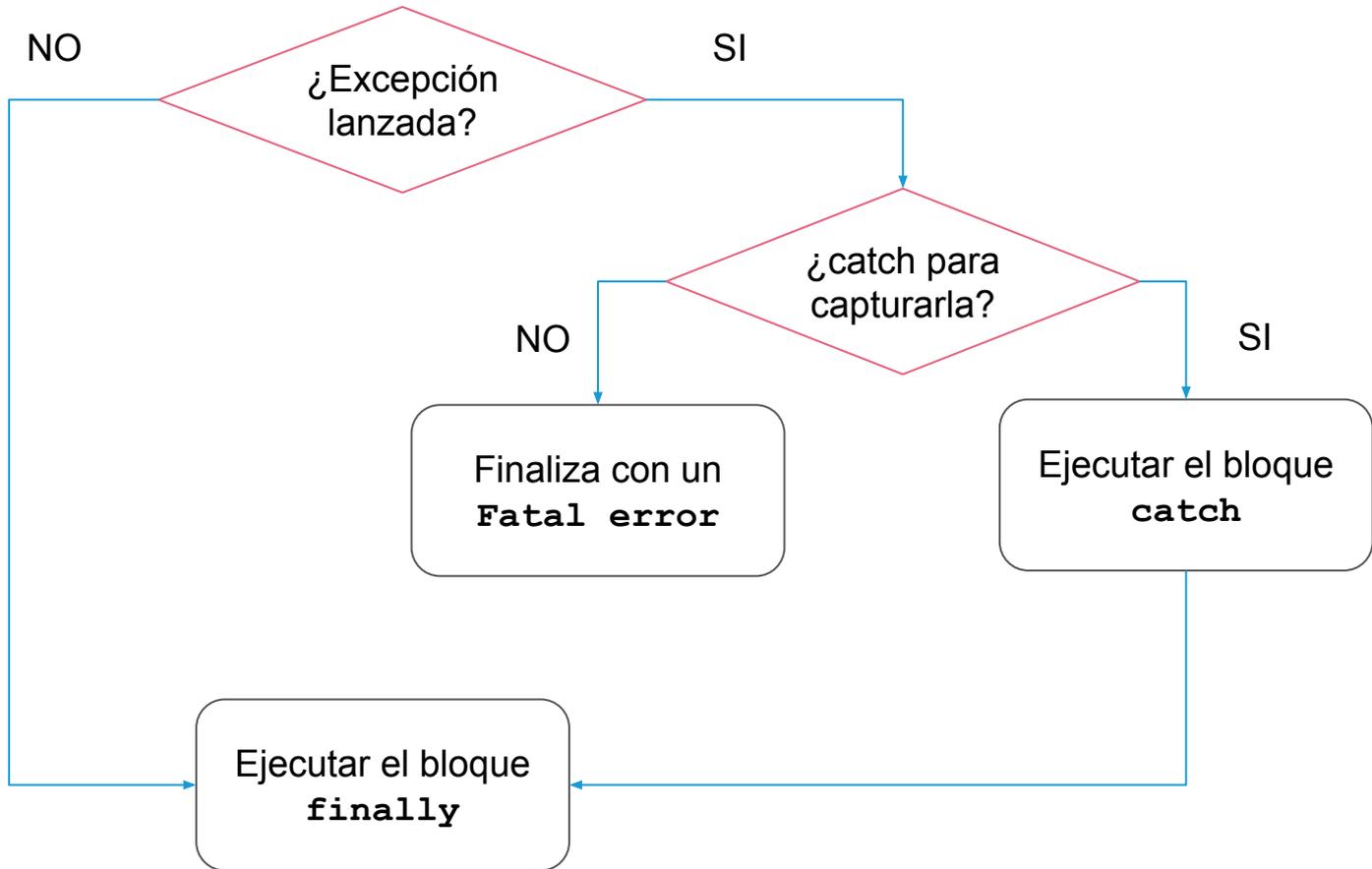


Interfaces

```
interface Vehiculo {  
    public function acelerar();  
    public function frenar();  
}  
  
class Coche implements Vehiculo {  
    public function acelerar(){  
        echo "He acelerado a 100 km/h";  
    }  
}
```

Excepciones

- Facilitan el **manejo de errores**. Las excepciones pueden ser lanzadas y capturadas:
 - **Lanzar una excepción**: es una sencilla forma de **informar** de un error de forma controlada.
 - **Capturar la excepción**: se define un bloque de código (**try**) a ejecutar, el cual es capaz de **capturar** las excepciones que ocurran en su interior y así **reaccionar** de forma controlada ante los errores.



Lanzar una excepción

```
public function miFuncion($parametro){  
  
    if (!$parametro) {  
        throw new Exception("El parámetro no puede ser  
false");  
    } else {  
        return 1;  
    }  
  
}
```

Capturar una excepción

```
try {  
  
    // codigo  
    // Si algo va mal, pueden saltar excepciones  
    // codigo: no se ejecuta si ha ocurrido una excepción  
  
} catch (Exception $e) {  
    // la excepción es capturada y se ejecuta el bloque  
    // $e->getMessage() contiene el mensaje de error.  
} finally {  
    // codigo: siempre se ejecuta  
}
```

Hands on!



Ejercicio 6

- Crea una función que reciba dos números como parámetros y realice la división del primero entre el segundo. La función debe lanzar una excepción en caso de que el valor del segundo parámetro sea igual a cero. A continuación haz un programa que utilice la función creada con distintos números. En caso de que uno de los divisores sea cero, tendrá que capturar la excepción y mostrar un mensaje indicando que no ha podido realizarse la operación.

Hands on!



Ejercicio 7

- Crea una función que calcule el área de un cuadrado a partir de los dos lados. La función debe lanzar una excepción en caso de que el valor de uno de los lados sea negativo. A continuación haz un programa que utilice la función creada con distintos casos. En caso de que uno de los lados sea negativo, tendrá que capturar la excepción y mostrar un mensaje indicando que no ha podido realizarse la operación.

Sources

- [PHP Group](https://www.php.net/): <https://www.php.net/>
- [PHP The Right Way](https://phptherightway.com): <https://phptherightway.com>
- [WikiBooks PHP](https://en.wikibooks.org/wiki/PHP_Programming): https://en.wikibooks.org/wiki/PHP_Programming