ÍNDICE

PROGRAMACIÓN ORIENTADA A OBJETOS

PHP sigue un paradigma de programación orientada a objetos (POO) basada en clases.

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

- *Métodos*. Son los miembros de la clase que contienen el código de la misma. *Un método es como una función.* Puede recibir parámetros y devolver valores.

- *Atributos o propiedades*. *Almacenan información acerca del estado del objeto al que pertenecen* (y por tanto, *su valor puede ser distinto para cada uno de los objetos de la misma clase*).

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

Los pilares fundamentales de la POO son:

Herencia. Es el proceso de crear una clase a partir de otra, heredando su comportamiento y características y pudiendo redefinirlos.

Abstracción. Hace referencia a que cada clase oculta en su interior las peculiaridades de su implementación, y presenta al exterior una serie de métodos (interface) cuyo comportamiento está bien definido. Visto desde el exterior, cada objeto es un ente abstracto que realiza un trabajo.

Polimorfismo. Un mismo método puede tener comportamientos distintos en función del objeto con que se utilice.

Encapsulación. En la POO se juntan en un mismo lugar los datos y el código que los manipula.

MOTIVACIÓN

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

Modularidad. La POO permite dividir los programas en partes o módulos más pequeños, que son independientes unos de otros pero pueden comunicarse entre ellos.

Extensibilidad. Si se desean añadir nuevas características a una aplicación, la POO facilita esta tarea de dos formas: añadiendo nuevos métodos al código, o creando nuevos objetos que extienden el comportamiento de los ya existentes.

Mantenimiento. Los programas desarrollados utilizando POO son más sencillos de mantener, debido a la modularidad antes comentada. También ayuda seguir ciertas convenciones al escribirlos, por ejemplo, escribir cada clase en un fichero propio. No debe haber dos clases en un mismo fichero, ni otro código aparte del propio de la clase.

Creación de clases

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

Conviene **hacerlo de forma ordenada;**

1. Primero las **propiedades o atributos**→ Los atributos de una clase son similares a las variables de PHP.

   Es posible indicar un valor en la declaración de la clase. En este caso, *todos los objetos que se instancian a partir de esa clase, partirán con ese valor por defecto en el atributo.*

2. Después los **métodos**, cada uno con su código respectivo.

```php
<?php
class Producto {
private $codigo;
public $nombre;
public $PVP;
public function muestra() {
print "<p>" . $this->codigo . "</p>";
}
}
?>
```

NOTA; *Es preferible que cada clase figure en su propio fichero* (producto.php). Además, muchos programadores prefieren utilizar para las clases nombres que comienzan por letra mayúscula, para de esta forma, *distinguirlos de los objetos y otras variables*.

Cómo instanciar objetos

Una vez definida la clase, podemos usar la palabra **new** para *instanciar objetos*.

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

Y haber declarado la clase en el programa en el que la usemos con include/require (include_once/requiere_once)

```php
require_once('producto.php');
$p = new Producto();
```

Cómo acceder desde un objeto a los atributos o métodos de una clase

Utilizar el **operador flecha** (NOTA, *sólo se pone el símbolo $ delante del nombre del objeto*):

```
require_once('producto.php');
$p = new Producto();
$p->nombre = 'LG';
$p->muestra();
```

Si accedemos dentro de la clase a una propiedad o método de la misma clase, utilizaremos la referencia **$this**

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

Declarar atributos

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

Además, cuando se declara un atributo, se debe indicar su nivel de acceso, siendo los principales niveles:

public. Los atributos declarados como public pueden utilizarse directamente por los objetos de la clase. ej, $nombre

private. Los atributos declarados como private s*ólo pueden ser accedidos y modificados por los métodos definidos en la clase,* no directamente por los objetos de la misma. ej, $codigo.

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

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

```php
<?php
class Persona {
    private string $nombre;
    public function setNombre(string $nom) {
        $this->nombre=$nom;
    }
    public function imprimir(){
        echo $this->nombre;
        echo '<br>';
    }
}
$bruno = new Persona(); // creamos un objeto
$bruno->setNombre("Bruno Díaz");
$bruno->imprimir();
?>
```

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

```php
<?php
class MayorMenor {
    private int $mayor;
    private int $menor;
    public function setMayor(int $may) {
        $this->mayor = $may;
    }
    public function setMenor(int $men) {
        $this->menor = $men;
    }
    public function getMayor() : int {
        return $this->mayor;
    }
    public function getMenor() : int {
        return $this->menor;
    }
}
```

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

Ej. el código siguiente simula que la clase Producto tiene cualquier atributo que queramos usar.

```php
class Producto {
    private $atributos = array();
    public function __get($atributo) {
    return $this->atributos[$atributo];
    }
    public function __set($atributo, $valor) {
    this->atributos[$atributo] = $valor;
    }
    }
```

```php
$p= new Producto();
    print_r($p);
    $p->noexisto = "Ahora si que existo";
    print_r($p);
```

Encapsulación

Las propiedades se definen privadas o protegidas (si queremos que las clases heredadas puedan acceder).
Para cada propiedad, **se añaden métodos públicos.**

Constantes

Además de métodos y propiedades, en una clase también se pueden definir constantes, utilizando la palabra **const.**

No confundas los atributos con las constantes:

Las constantes no pueden cambiar su valor

No usan el carácter $

Su valor va siempre entre comillas y está asociado a la clase, es decir, no existe una copia del mismo en cada objeto. Por tanto, *para acceder a las constantes de una clase*, se debe utilizar el **nombre de la clase** y el **operador ::,** *llamado operador de resolución de ámbito*

```php
class DB {
        const USUARIO = 'dwes';
        …
        }
        print DB::USUARIO;
```

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

Clases estáticas

Se definen utilizando la palabra clave **static** y se referencian con **::**

Los atributos estáticos en PHP **son variables que pertenecen a la clase en sí misma** y **no a una instancia específica de esa clase.** Esto significa que *su valor es compartido por todos los objetos de esa clase,* y *puede ser accedido y modificado sin necesidad de crear una instancia.*

EJ de uso :

- Datos globales a la clase: Cuando necesitas almacenar información que sea común a todos los objetos de una clase, como un contador de instancias o una configuración global.
- Variables de caché: Para almacenar resultados de cálculoso datos que se utilizan con frecuencia.

```
class Producto {
        private static $num_productos = 0;
        public static function nuevoProducto() {
        self::$num_productos++;
        }
       }
```

Los atributos y métodos estáticos **no pueden ser llamados** desde un objeto de la clase **utilizando** el operador **->**. Ni **es posible llamarlos desde un objeto usando $this** dentro de un método estático.

Si el método o atributo es público, **deberá accederse utilizando el nombre de la clase y el operador de resolución de ámbito,** ej. Producto::nuevoProducto()

Si **es privado,** como el atributo $num_productos en el ejemplo anterior, sólo se podrá acceder a él desde los métodos de la propia clase, *utilizando* la palabra **self**. De la misma forma que $this hace referencia al objeto actual, self hace referencia a la clase actual. Es decir, Si desde un método queremos acceder a una propiedad estática de la misma clase, se utiliza la referencia **self**: ej self::$numProductos

```
    Producto::nuevoProducto();
    self::$num_productos ++;
```

Constructores

El constructor de una clase debe llamarse **__construct**. Se pueden utilizar, por ejemplo, para asignar valores a atributos.

Es un método especial dentro de una clase que se ejecuta automáticamente cada vez que se crea un nuevo objeto de esa clase. Su principal función es **inicializar las propiedades (atributos) del objeto con valores iniciales o realizar cualquier otra tarea necesaria para preparar el objeto para su uso.**

```php
class Producto {
        private static $num_productos = 0;
        private $codigo;
        public function __construct() {
        self::$num_productos++;
        }
        }
```

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

```php
class Producto {
        private static $num_productos = 0;
        private $codigo;
        public function __construct($codigo) {
        $this->$codigo = $codigo;
        self::$num_productos++;
        }
        }
```

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

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

```php
class Producto {
        private static $num_productos = 0;
        private $codigo;
        public function __construct($codigo) {
        $this->codigo = $codigo;
        self::$num_productos++;
        }
        public function __destruct() {
        self::$num_productos--;
        }
        }
```

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

        unset($p);
```

Introspección

DEFINICIÓN - Introspección es la capacidad de un programa para examinar su propia estructura y comportamiento en tiempo de ejecución.

En el contexto de las clases, esto significa que podemos obtener información sobre:

Propiedades: Nombre, tipo y valor de las propiedades de una clase.

Métodos: Nombre, argumentos y valor de retorno de los métodos.

Clases: Jerarquía de clases, interfaces implementadas y rasgos utilizados.

Ej de uso, **generar documentación, crear frameworks y bibliotecas flexibles, depurar código, metaprogramación (**escribir código que genera o modifica otro código en tiempo de ejecución)

En PHP existen un conjunto de funciones ya definidas por el lenguaje que permiten obtener información sobre los objetos:

- **instanceof**: permite comprobar si un objeto es de una determinada clase
- **get_class:** devuelve el nombre de la clase
- **get_declared_class:** devuelve un array con los nombres de las clases definidas
- **class_alias**: crea un alias
- **class_exists / method_exists / property_exists**: true si la clase / método / propiedad está definida
- **get_class_methods / get_class_vars / get_object_vars**: Devuelve un array con los nombres de los métodos / propiedades de una clase / propiedades de un objeto que son accesibles desde dónde se hace la llamada (es decir, que sean públicos).

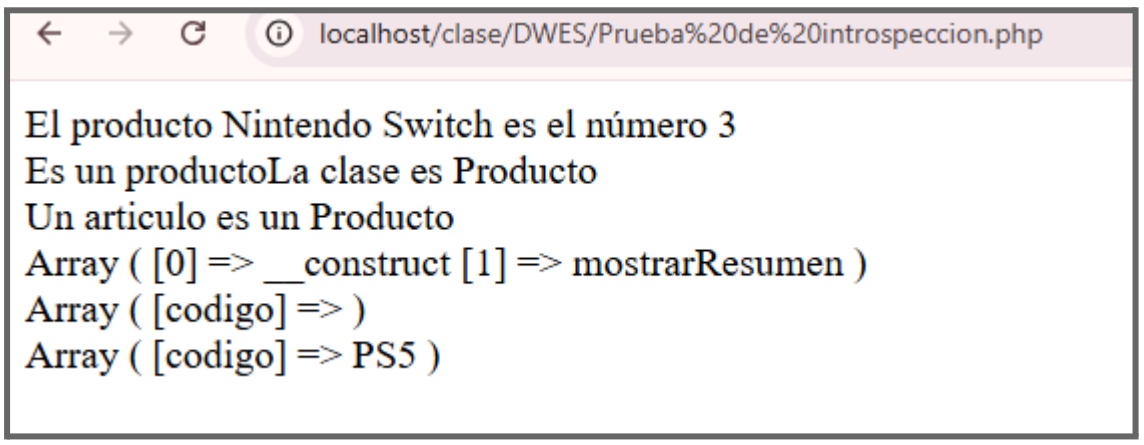
introspección.php

```php
<?php
include_once('claseProducto.php');
$p = new Producto("PS5");
if ($p instanceof Producto) {
    echo " Es un producto";
    echo "La clase es ".get_class($p)."<br>";

    class_alias("Producto", "Articulo");
    $c = new Articulo("Nintendo Switch");
    echo "Un articulo es un ".get_class($c)."<br>";

    print_r (get_class_methods("Producto"));
    print_r(nl2br("\n"));
    print_r(get_class_vars("Producto"));
```

```
    print_r(nl2br("\n"));
    print_r(get_object_vars($p));

    if (method_exists($p, "mostrarResumen")) {
        $p->mostrarResumen();
    }
}
?>
```

claseProducto.php

```
<?php
class Producto {
    const IVA = 0.23;
    private static $numProductos = 0;
    public $codigo;

    public function __construct(string $cod) {
        self::$numProductos++;
        $this->codigo = $cod;
    }
    public function mostrarResumen() : string {
                return "El producto ".$this->codigo." es el número
".self::$numProductos;
    }
}
$prod1 = new Producto("PS5");
$prod2 = new Producto("XBOX Series X");
$prod3 = new Producto("Nintendo Switch");
echo $prod3->mostrarResumen();
print_r(nl2br("\n"));

?>
```

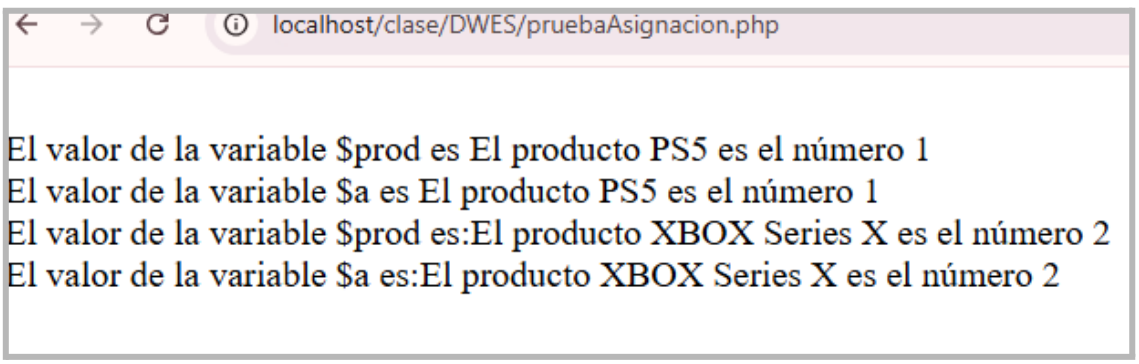


Clonado

¿Qué sucede cuando en PHP se ejecuta un código como el siguiente?

pruebaAsignacion.php

```php
<?php
class Producto {
    const IVA = 0.23;
    private static $numProductos = 0;
    public $codigo;


    public function __construct(string $cod) {
        self::$numProductos++;
        $this->codigo = $cod;
    }
    public function mostrarResumen() : string {
                return "El producto ".$this->codigo." es el número
".self::$numProductos;
    }
}
$prod = new Producto("PS5");
$a=$prod;
print_r(nl2br(" \nEl valor de la variable \$prod es "));
echo $prod->mostrarResumen();
print_r(nl2br(" \nEl valor de la variable \$a es "));
echo $prod->mostrarResumen();
$prod = new Producto("XBOX Series X");
print_r(nl2br(" \nEl valor de la variable \$prod es:"));
echo $prod->mostrarResumen();
print_r(nl2br(" \nEl valor de la variable \$a es:"));
echo $prod->mostrarResumen();
?>
```



En PHP el código anterior simplemente crea un nuevo identificador del mismo objeto. Es decir, cuando se utilice uno cualquiera de los identificadores para cambiar el valor de algún atributo, este cambio se vería también reflejado al acceder utilizando el otro identificador. **Recuerda que en realidad todos se refieren a la única copia que se almacena del mismo**.

Al asignar dos objetos no se copian, se crea una nueva referencia. Si queremos una copia, hay que clonarlo mediante el método **clone(object) : object**

Para el ej

```php
$p = new Producto();
$p->nombre = 'Samsung Galaxy S';
$a = clone($p);
```

Si queremos modificar el clonado por defecto, existe una forma sencilla de personalizar la copia para cada clase particular (por ejemplo, copiar todos los atributos menos alguno), hay que definir el método mágico **__clone()** que se llamará después de copiar todas las propiedades (después de copiar todos los atributos en el nuevo objeto).

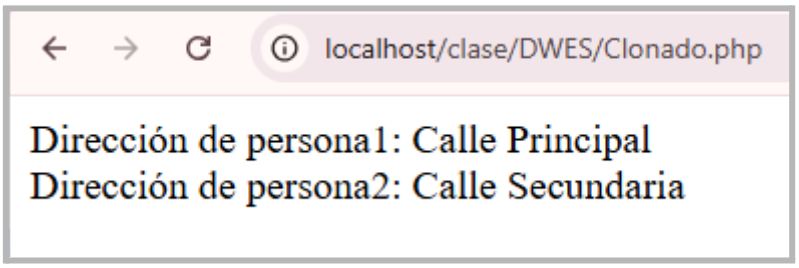
clonado.php

```php
<?php
class Persona {
    public $nombre;
    public $edad;
    public $direccion; // Un objeto de la clase Direccion
    public function __construct($nombre, $edad, Direccion $direccion) {
        $this->nombre = $nombre;
        $this->edad = $edad;
        $this->direccion = $direccion;
    }
    public function __clone() {
        // Clonamos la dirección para evitar referencias compartidas
        $this->direccion = clone $this->direccion;
    }
}
class Direccion {
    public $calle;
    public $numero;
    public $ciudad;
}
// Creamos una dirección
$direccion1 = new Direccion();
$direccion1->calle = "Calle Principal";
$direccion1->numero = 123;
$direccion1->ciudad = "Madrid";
// Creamos una persona
```

```php
$persona1 = new Persona("Juan", 30, $direccion1);
// Clonamos la persona
$persona2 = clone $persona1;
// Modificamos la dirección de la persona clonada
$persona2->direccion->calle = "Calle Secundaria";
// Imprimimos las direcciones
echo "Dirección de persona1: " . $persona1->direccion->calle . "<br>";
echo "Dirección de persona2: " . $persona2->direccion->calle . "<br>";
?>
```

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

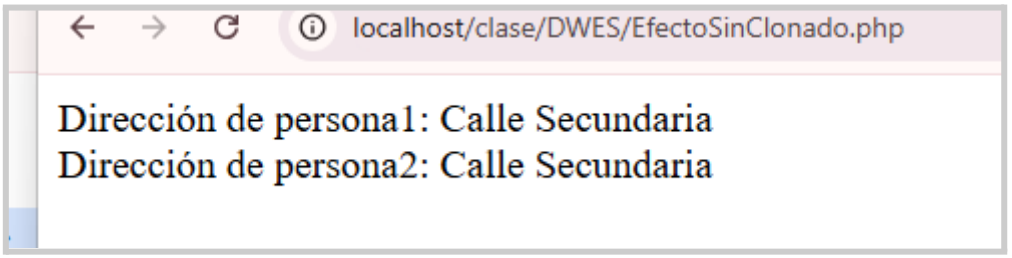
```php
    public function __construct($nombre, $edad, Direccion $direccion)
```



efectoSinClonado.php

```php
<?php
class Persona {
    public $nombre;
    public $edad;
    public $direccion; // Un objeto de la clase Direccion
    public function __construct($nombre, $edad, Direccion $direccion) {
        $this->nombre = $nombre;
        $this->edad = $edad;
        $this->direccion = $direccion;
    }
    /*public function __clone() {
        // Clonamos la dirección para evitar referencias compartidas
        $this->direccion = clone $this->direccion;
    }*/
}
class Direccion {
    public $calle;
    public $numero;
    public $ciudad;
}
// Creamos una dirección
```

```php
$direccion1 = new Direccion();
$direccion1->calle = "Calle Principal";
$direccion1->numero = 123;
$direccion1->ciudad = "Madrid";
// Creamos una persona
$persona1 = new Persona("Juan", 30, $direccion1);
// Clonamos la persona
$persona2 = clone $persona1;
// Modificamos la dirección de la persona clonada
$persona2->direccion->calle = "Calle Secundaria";
// Imprimimos las direcciones
echo "Dirección de persona1: " . $persona1->direccion->calle . "<br>";
echo "Dirección de persona2: " . $persona2->direccion->calle . "<br>";
?>
```



IMPORTANTE, En PHP puedes crear referencias a variables (como números enteros o cadenas de texto), utilizando el operador &:

```php
$p = new Producto();
$p->nombre = 'Samsung Galaxy S';
$a = clone($p);
// El resultado de comparar $a === $p da falso
// pues $a y $p no hacen referencia al mismo objeto
$a = & $p;
// Ahora el resultado de comparar $a === $p da verdadero
// pues $a y $p son referencias al mismo objeto.
```

Herencia

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

ej. de herencia → Una tienda web que vende electrónica de distintos tipos.

claseProducto.php ->Para manejar la entrada de productos genéricos (entrada en el almacen) creamos una clase llamada Producto, con algunos atributos y un método que genera una salida personalizada en formato HTML del código.

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

claseExtProducto.php→ Para almacenar características de los productos (ej, pulgadas de monitores, capacidad y procesado de ordenadores, tecnología …). Esta clase será una **clase heredada de** la clase **Producto.**

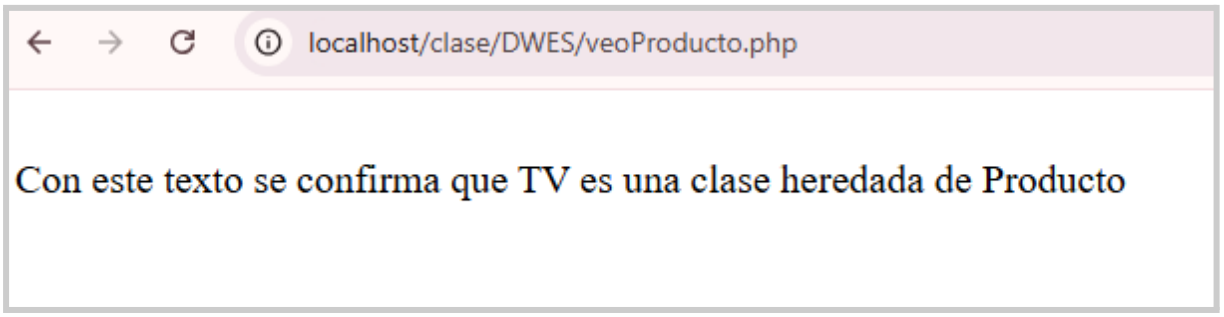
```php
<?php
class TV extends Producto
{
public $pulgadas;
public $tecnologia;
}
?>
```

Para definir una clase que herede de otra, utilizar la palabra **extends *indicando la superclase (*** `class TV extends Producto)`. *Los nuevos objetos que se instancian a partir de la subclase* **son también objetos de la clase base**; se puede *comprobar utilizando el operador instanceof (ver ejemplo a continuación)*.

veoProducto.php

```php
<?php
include_once ('claseProductoElectronica.php');
include_once ('claseExtProducto.php');
$t = new TV();
if ($t instanceof Producto)
{
// Este código se ejecuta pues la condición es cierta
echo "<br>Con este texto se confirma que TV es una clase heredada de
Producto";
}
?>
```

RESULTADO

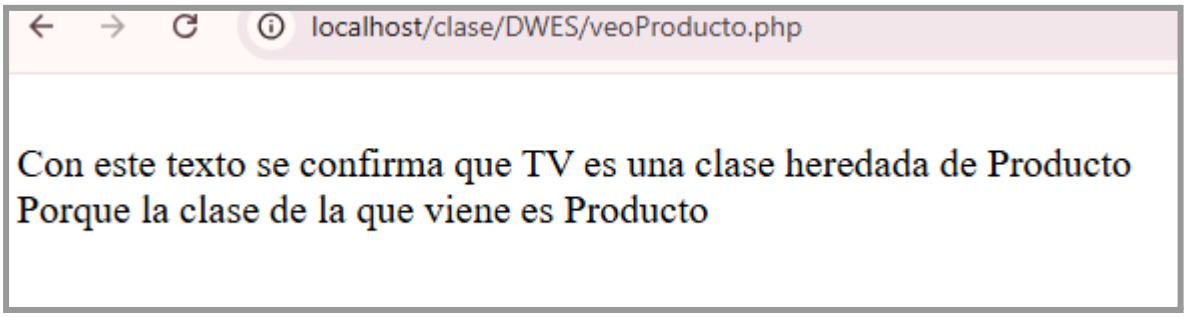


De las funciones definidas en PHP que permiten obtener información sobre los objetos y que están relacionadas con la herencia de clases destacamos:

get_parent_class ;Devuelve el nombre de la clase padre del objeto o la clase que se indica.

is_subclass_of ; Comprueba si el objeto tiene esta clase como uno de sus padres.

```php
<?php
include_once ('claseProductoElectronica.php');
include_once ('claseExtProducto.php');
$t = new TV();
if ($t instanceof Producto)
{
// Este código se ejecuta pues la condición es cierta
echo "<br>Con este texto se confirma que TV es una clase heredada de Producto";
$claseOriginal=get_parent_class($t);
if (is_subclass_of($t, 'Producto')) {
echo "<br>Porque efectivamente la clase de la que viene es ". $claseOriginal."<br>";
}
}
?>
```



IMPORTANTE, **La nueva clase hereda todos los atributos y métodos públicos** de la clase base, **pero no los privados**.

> *Para crear en la clase base un método no visible al exterior* (como los privados) *que se herede a las subclases,* debes *utilizar la palabra* **protected** en lugar de private.

Constructor en hijos

En los hijos no se crea ningún constructor de manera automática. Por lo que si no lo hay, *se invoca automáticamente al del padre.* En cambio, **si lo definimos en el hijo**, hemos de **invocar al del padre de manera explícita**.

EJ,claseProductoElectronica.php

```php
<?php
class Producto
{
public function __construct(public int $codigo, public $nombre=null,public $nombre_corto=null, public $PVP=null) {
    $this->codigo = $codigo;
}
final public function muestra()
{
echo "<br> El código del producto desde la superclase es ". $this->codigo . "<br>";
}
}
?>
```

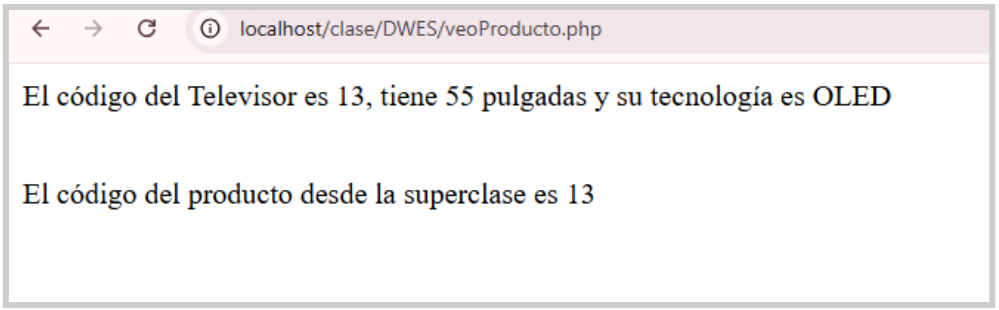
claseExtProducto.php

```php
<?php
include_once ('claseProductoElectronica.php');
class TV extends Producto
{
public function __construct(int $codigo, private int $pulgadas, private string $tecnologia){
    parent::__construct($codigo);
}
public function muestraHerencia()
{
echo "<p>El código del Televisor es {$this->codigo}, tiene {$this->pulgadas} pulgadas y su tecnología es {$this->tecnologia} </p>";
}
}
```

```php
?>
```

veoProducto.php

```php
<?php
include_once ('claseProductoElectronica.php');
include_once ('claseExtProducto.php');

//constructor en la clase heredada
$t = new TV(13,55,"OLED");
$p = new Producto(13);
$t->muestraHerencia();
$p->muestra();

?>
```



Sobrescribir métodos

Para **redefinir el comportamiento de los métodos existentes en la clase base,** crear **en la subclase** un n**uevo método con el mismo nombre**.

claseProductoElectronica.php

```php
<?php
class Producto
{
public $codigo;
public $nombre;
public $nombre_corto;
public $PVP;
public function __construct($codigo=null) {
    $this->codigo = $codigo;
}
public function muestra()
{
echo "<br> El código del producto es ". $this->codigo . "<br>";
}
}
```

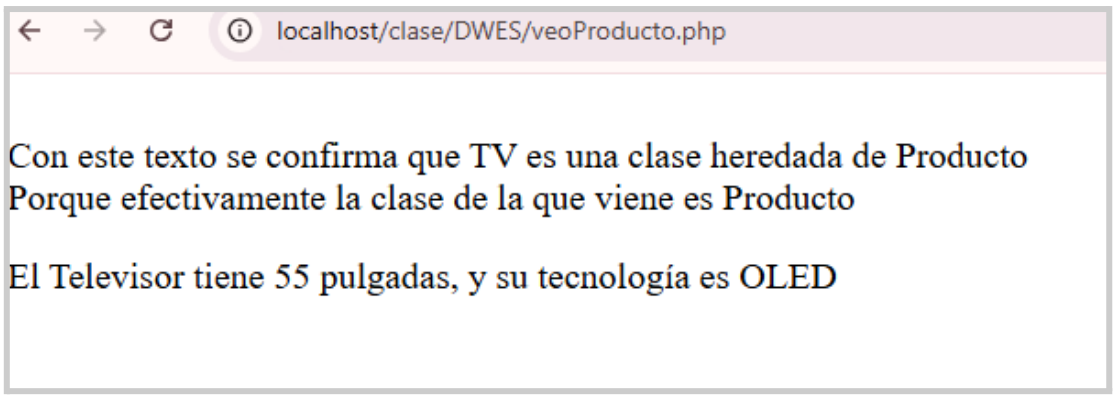
```
?>
```

claseExtProducto.php

```
<?php
include_once ('claseProductoElectronica.php');
class TV extends Producto
{
public $pulgadas;
public $tecnologia;
public function muestra()
{
echo "<p>El Televisor tiene {$this->pulgadas} pulgadas, y su tecnología
es {$this->tecnologia} </p>";
}
}
?>
```

veoProducto.php

```
<?php
include_once ('claseProductoElectronica.php');
include_once ('claseExtProducto.php');
$t = new TV();
$p = new Producto(13);

$t->pulgadas = 55;
$t->tecnologia = "OLED";
if ($t instanceof Producto)
{
// Este código se ejecuta pues la condición es cierta
echo "<br>Con este texto se confirma que TV es una clase heredada de
Producto";
$claseOriginal=get_parent_class($t);
if (is_subclass_of($t, 'Producto')) {
echo "<br>Porque efectivamente la clase de la que viene es ".
$claseOriginal."<br>";
$t->muestra();
}
}
?>
```



Clase final

Utilizando la palabra **final** se evita que las clases heredadas puedan redefinir el comportamiento de los métodos existentes en la superclase. La palabra **final** puede usarse en la definición del método e incluso en la definición de la clase **final class Producto { ... }**

EJ, redefiniendo el método muestra() de claseProductoElectronica.php con

```
final public function muestra()
```

```php
<?php
class Producto
{
public $codigo;
public $nombre;
public $nombre_corto;
public $PVP;
public function __construct($codigo=null) {
    $this->codigo = $codigo;
}
final public function muestra()
{
echo "<br> El código del producto desde la superclase es ".
$this->codigo . "<br>";
}
}
?>
```

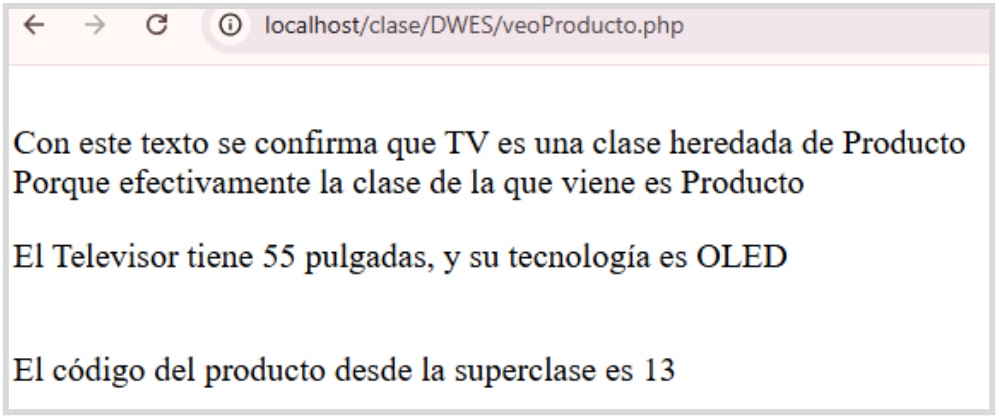
El resultado al ejecutar el código es un Fatal error por redefinición del método muestra en la clase heredada

Fatal error: Cannot override final method Producto::muestra() in **C:\Users\PC Maria\Documents\apache_1\DWES_M\DWES\claseExtProducto.php** on line 7

Tenemos que corregir el código en claseExtProducto.php y veoProducto.php como sigue

```php
<?php
include_once ('claseProductoElectronica.php');
class TV extends Producto
{
public $pulgadas;
public $tecnologia;
public function muestraHerencia()
{
echo "<p>El Televisor tiene {$this->pulgadas} pulgadas, y su tecnología
es {$this->tecnologia} </p>";
}
}
?>
```

```php
<?php
include_once ('claseProductoElectronica.php');
include_once ('claseExtProducto.php');
$t = new TV();
$p = new Producto(13);

$t->pulgadas = 55;
$t->tecnologia = "OLED";
if ($t instanceof Producto)
{
// Este código se ejecuta pues la condición es cierta
echo "<br>Con este texto se confirma que TV es una clase heredada de
Producto";
$claseOriginal=get_parent_class($t);
if (is_subclass_of($t, 'Producto')) {
echo "<br>Porque efectivamente la clase de la que viene es ".
$claseOriginal."<br>";
$t->muestraHerencia();
$p->muestra();
}
}
?>
```



Clases abstractas¶

Son clases opuestas a **final.**

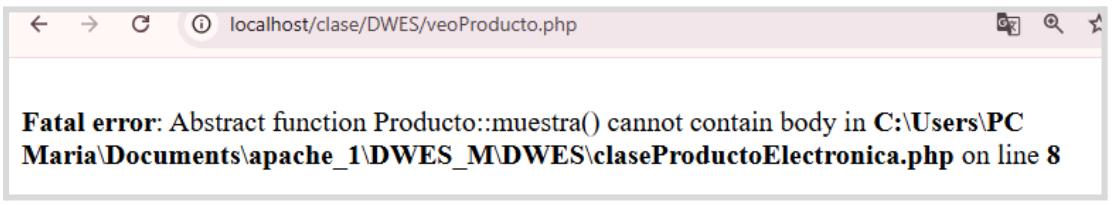
Las clases abstractas obligan a heredar de una clase, ya que no se permite su instanciación. Se define mediante **abstract class NombreClase {.**

Una clase abstracta **puede contener** *propiedades y métodos no-abstractos. y/o métodos abstractos.*

Ej. si modificamos nuestro código de claseProductoElectronica.php

```php
<?php
abstract class Producto
{
public function __construct(public int $codigo, public $nombre=null,public $nombre_corto=null, public $PVP=null) {
    $this->codigo = $codigo;
}
// Método abstracto
abstract public function muestra();
}
?>
```

El resultado en el cliente será



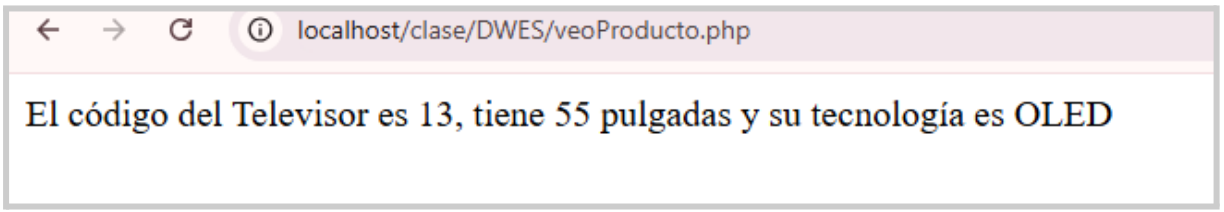
Para corregir este error debemos modificar el código de claseExtProducto.php como sigue

```php
<?php
include_once ('claseProductoElectronica.php');
class TV extends Producto
{
public function __construct(int $codigo, private int $pulgadas, private
string $tecnologia){
    parent::__construct($codigo);
}
public function muestra()//obligado a implementarlo al ser abstracta la
clase Producto
{
echo "<p>El código del Televisor es {$this->codigo}, tiene
{$this->pulgadas} pulgadas y su tecnología es {$this->tecnologia}
</p>";
}
}
?>
```

veoProducto.php

```php
<?php
include_once ('claseProductoElectronica.php');
include_once ('claseExtProducto.php');
//constructor en la clase heredada
$t = new TV(13,55,"OLED");
$t->muestra();
?>
```

Siendo el resultado:



Introducimos arrays en el constructor

Para facilitar la creación de nuevos objetos, crearemos un constructor al que se le pasará un array con los valores de los atributos del nuevo producto.

ej.claseProductoElectronica.php

```php
<?php
class Producto
{
    public $codigo;
    public $nombre;
    public $nombre_corto;
    public $PVP;

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

public function muestra()
{
echo "<br> El código del producto desde la superclase es ".
$this->codigo . "<br>";
echo "<br> El nombre del producto desde la superclase es ".
$this->nombre . "<br>";
echo "<br> El nombre corto del producto desde la superclase es ".
$this->nombre_corto . "<br>";
echo "<br> El PVP del producto desde la superclase es ". $this->PVP.
"<br>";
}
}
?>
```

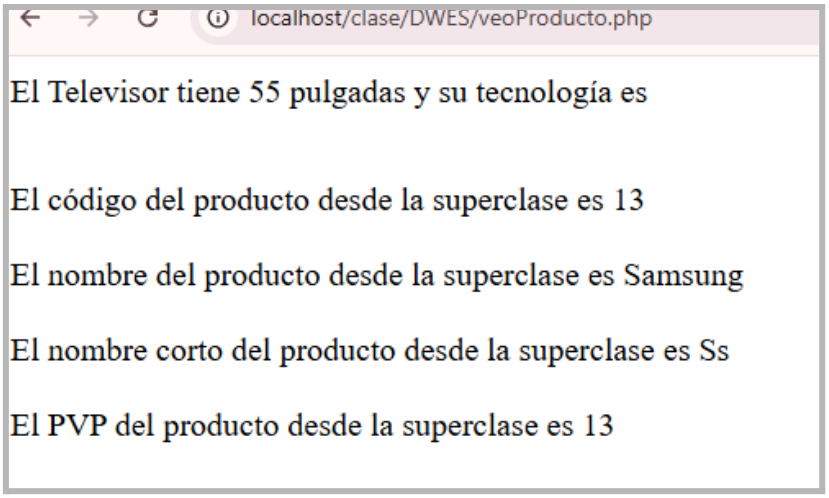
claseExtProducto.php

```php
<?php
include_once ('claseProductoElectronica.php');
class TV extends Producto
{
public $pulgadas;
public $tecnologia;
public function __construct($row){
    parent::__construct($row);
    $this->pulgadas = $row['pulgadas'];
}
public function muestra()
{
```

```
echo "<p>El Televisor tiene {$this->pulgadas} pulgadas y su tecnología
es {$this->tecnologia} </p>";
}
}
?>
```

veoProducto.php

```
<?php
include_once ('claseProductoElectronica.php');
include_once ('claseExtProducto.php');
$datosProducto = [
    'codg' => 13,
    'nomb' => 'Samsung',
    'nom_cort' => 'Ss',
    'PVP' => 13,
    'pulgadas'=>55
];
$t = new TV($datosProducto);
$p = new Producto($datosProducto);
$t->muestra();
$p->muestra();
?>
```



Interfaces

Permite definir un contrato con las firmas de los métodos a cumplir. Así pues, **sólo** *contiene* ***declaraciones de funciones*** *y* ***todas deben ser públicas.***

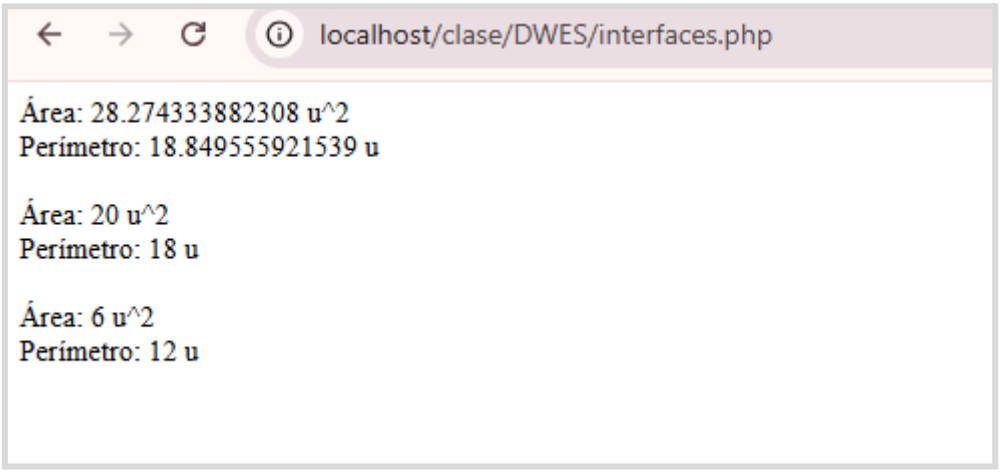
Se declaran con la palabra clave **interface** y luego *las clases que cumplan el contrato* lo realizan mediante *la palabra clave* **implements**.

ej interface.php

```php
<?php

interface FormaGeometrica {
    public function calcularArea();
    public function calcularPerimetro();
}

class Circulo implements FormaGeometrica {
    private $radio;

    public function __construct($radio) {
        $this->radio = $radio;
    }

    public function calcularArea() {
        return pi() * pow($this->radio, 2);
    }

    public function calcularPerimetro() {
        return 2 * pi() * $this->radio;
    }
}

class Rectangulo implements FormaGeometrica {
    private $base;
    private $altura;

    public function __construct($base, $altura) {
        $this->base = $base;
        $this->altura = $altura;
    }

    public function calcularArea() {
        return $this->base * $this->altura;
    }

    public function calcularPerimetro() {
        return 2 * ($this->base + $this->altura);
    }
}
```

```php
// Nueva clase que implementa la interfaz
class Triangulo implements FormaGeometrica {
    private $base;
    private $altura;
    private $lado1;
    private $lado2;
    private $lado3;

    public function __construct($base, $altura, $lado1, $lado2, $lado3)
{
        $this->base = $base;
        $this->altura = $altura;
        $this->lado1 = $lado1;
        $this->lado2 = $lado2;
        $this->lado3 = $lado3;
    }

    public function calcularArea() {
        return ($this->base * $this->altura) / 2;
    }

    public function calcularPerimetro() {
        return $this->lado1 + $this->lado2 + $this->lado3;
    }
}

// Función que acepta cualquier forma que implemente FormaGeometrica
function mostrarInformacionForma(FormaGeometrica $forma) {
    echo "Área: " . $forma->calcularArea() . " u^2 <br>";
    echo "Perímetro: " . $forma->calcularPerimetro() . " u <br><br>";
}

// Uso genérico con distintas formas
$formas = [
    new Circulo(3),
    new Rectangulo(4, 5),
    new Triangulo(3, 4, 3, 4, 5)
];

foreach ($formas as $forma) {
    mostrarInformacionForma($forma);
}
```

```
?>
```



Se permite la herencia de interfaces. Además, una clase puede implementar varios interfaces (en este caso, sí soporta la herecia múltiple, pero sólo de interfaces).

EJ.

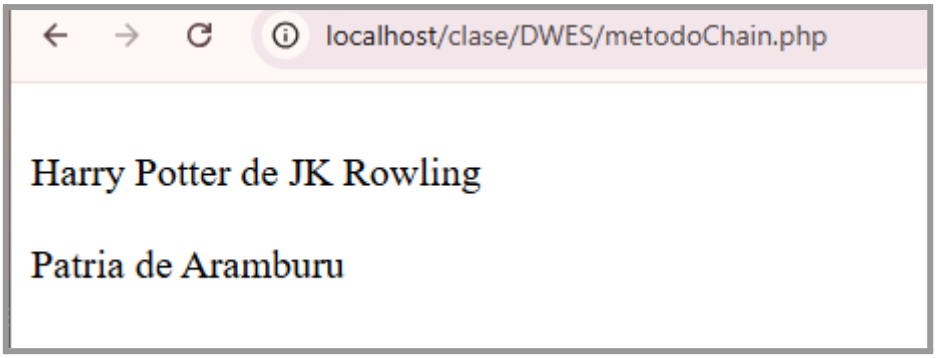
```
<?php
interface Mostrable {
    public function mostrarResumen() : string;
}

interface MostrableTodo extends Mostrable {
    public function mostrarTodo() : string;
}

interface Facturable {
    public function generarFactura() : string;
}

class Producto implements MostrableTodo, Facturable {
    // Implementaciones de los métodos
    // Obligatoriamente deberá implementar public function mostrarResumen, mostrarTodo y generarFactura
}
```

Métodos encadenados

Sigue el planteamiento de la programación funcional, y también se conoce como method chaining. Plantea que sobre un objeto se realizan varias llamadas.

Los métodos encadenados son una técnica en la programación orientada a objetos que permite llamar a múltiples métodos de un objeto en una sola línea de código. Esto se logra retornando `$this` al final de cada método.

```php
<?php
class Libro {
    private string $nombre;
    private string $autor;

    public function getNombre() : string {
        return $this->nombre;
    }
    public function setNombre(string $nombre) : Libro {
        $this->nombre = $nombre;
        return $this;
    }

    public function getAutor() : string {
        return $this->autor;
    }
    public function setAutor(string $autor) : Libro {
        $this->autor = $autor;
        return $this;
    }

    public function __toString() : string {
        return $this->nombre." de ".$this->autor;
    }
}

$p1 = new Libro();
$p1->setNombre("Harry Potter");
$p1->setAutor("JK Rowling");
echo "<br>".$p1."<br>";
// Method chaining
$p2 = new Libro();
$p2->setNombre("Patria")->setAutor("Aramburu");
echo "<br>".$p2."<br>";
?>
```



Espacio de nombres

Desde PHP 5.3 y también conocidos como Namespaces, permiten organizar las clases/interfaces, funciones y/o constantes de forma similar a los paquetes en Java.

RECOMENDACIÓN→ Un sólo namespace por archivo y crear una estructura de carpetas respectando los niveles/subniveles (igual que se hace en Java)

Se declaran en la primera línea mediante la palabra clave namespace seguida del nombre del espacio de nombres asignado (cada subnivel se separa con la barra invertida \):

Por ejemplo, para colocar la clase Producto dentro del namespace Dwes\Ejemplos lo haríamos así:

```php
<?php
namespace Dwes\Ejemplos;

const IVA = 0.21;

class Producto {
    public $nombre;

    public function muestra() : void {
        echo"<p>Prod:" . $this->nombre . "</p>";
    }
}
?>
```

Acceso

Para referenciar a un recurso que contiene un namespace, primero hemos de tenerlo disponible haciendo uso de include o require. Si el recurso está en el mismo namespace, se realiza un acceso directo (se conoce como acceso sin cualificar).

Realmente hay tres tipos de acceso:

sin cualificar: recurso

cualificado: rutaRelativa\recurso → no hace falta poner el namespace completo

totalmente cualificado: \rutaAbsoluta\recurso

```php
<?php
namespace Dwes\Ejemplos;
include_once("Producto.php");
echo IVA; // sin cualificar
echo Utilidades\IVA; // acceso cualificado. Daría error, no existe \Dwes\Ejemplos\Utilidades\IVA
echo \Dwes\Ejemplos\IVA; // totalmente cualificado
$p1 = new Producto(); // lo busca en el mismo namespace y encuentra \Dwes\Ejemplos\Producto
$p2 = new Model\Producto(); // daría error, no existe el namespace Model. Está buscando \Dwes\Ejemplos\Model\Producto
$p3 = new \Dwes\Ejemplos\Producto(); // \Dwes\Ejemplos\Producto
```

Para evitar la referencia cualificada podemos declarar el uso mediante **use** (similar a hacer import en Java). Se hace en la cabecera, tras el namespace:

Los tipos posibles son:

use const nombreCualificadoConstante

use function nombreCualificadoFuncion

use nombreCualificadoClase

use nombreCualificadoClase as NuevoNombre // para renombrar elementos

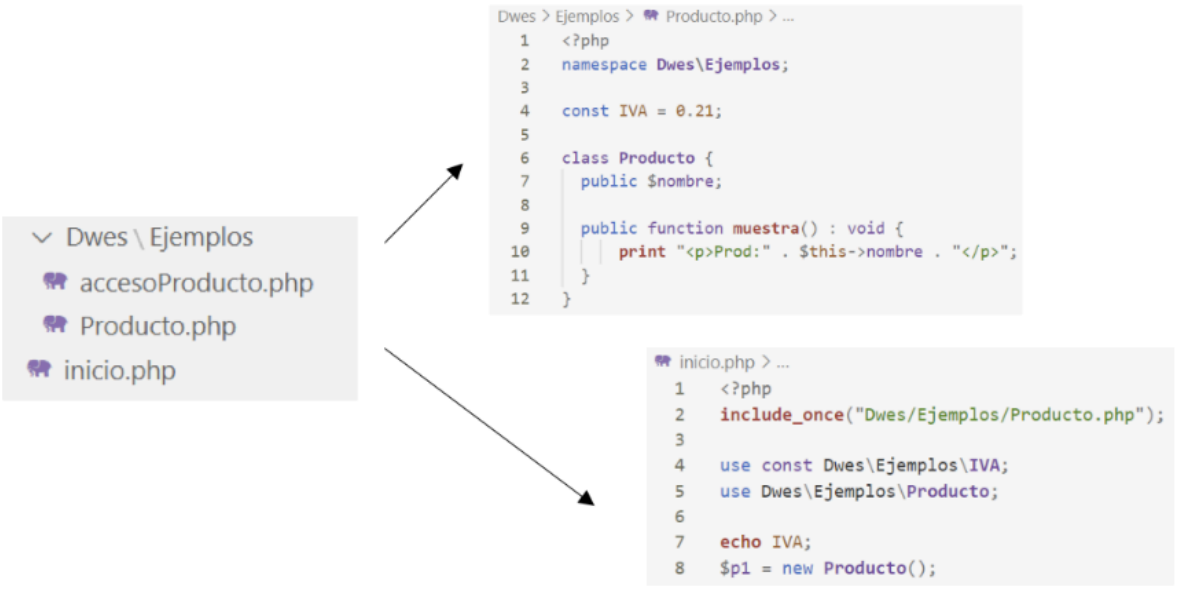
Por ejemplo, si queremos utilizar la clase \Dwes\Ejemplos\Producto desde un recurso que se encuentra en la raíz, por ejemplo en inicio.php, haríamos:

```php
<?php
include_once("Dwes\Ejemplo\Producto.php");
use const Dwes\Ejemplos\IVA;
use \Dwes\Ejemplos\Producto;
echo IVA;
$p1 = new Producto();
```

Organización

Todo proyecto, conforme crece, necesita organizar su código fuente.

Se plantea una organización en la que los **archivos que interactuan con el navegador** se colocan **en el raíz**, y las **clases que definamos** van dentro de un **namespace** *(y dentro de su propia carpeta src o app).*



Organización del código fuente

Organización, includes y usos

- Colocaremos **cada recurso en un fichero aparte.**
- En la *primera línea* indicaremos su **namespace** (si no está en el raíz).
- *Si utilizamos otros recursos*, haremos un **include_once** de esos recursos (clases, interfaces, etc...).
  - Cada recurso debe incluir todos los otros recursos que referencie: la clase de la que hereda, interfaces que implementa, clases utilizadas/recibidas como parámetros, etc...
- *Si los recursos están en un espacio de nombres diferente al que estamos*, emplearemos **use** con la ruta completa *para luego utilizar referencias sin cualificar.*

Autoload

Permite cargar las clases **(no las constantes ni las funciones**) que se van a utilizar y ***evitar*** *tener que hacer el* ***include_once*** *de cada una de ellas*. Para ello, se utiliza la función **spl_autoload_register ()**

> Cuando se llama a **spl_autoload_register(),** *se registra una función que será llamada cuando PHP no encuentre una clase definida.*
>
> *La función registrada intenta localizar el archivo que contiene la definición de la clase no encontrada.*

```
spl_autoload_register(function($clase) {
    include 'clases/' . $clase . '.php';
});
```

¿Cómo organizamos ahora nuestro código aprovechando el autoload?

Para facilitar la búsqueda de los recursos a incluir, es recomendable colocar todas las clases dentro de una misma carpeta.

Nosotros **la vamos a colocar** dentro de **app** (más adelante, *cuando estudiemos Laravel veremos el motivo de esta decisión*).

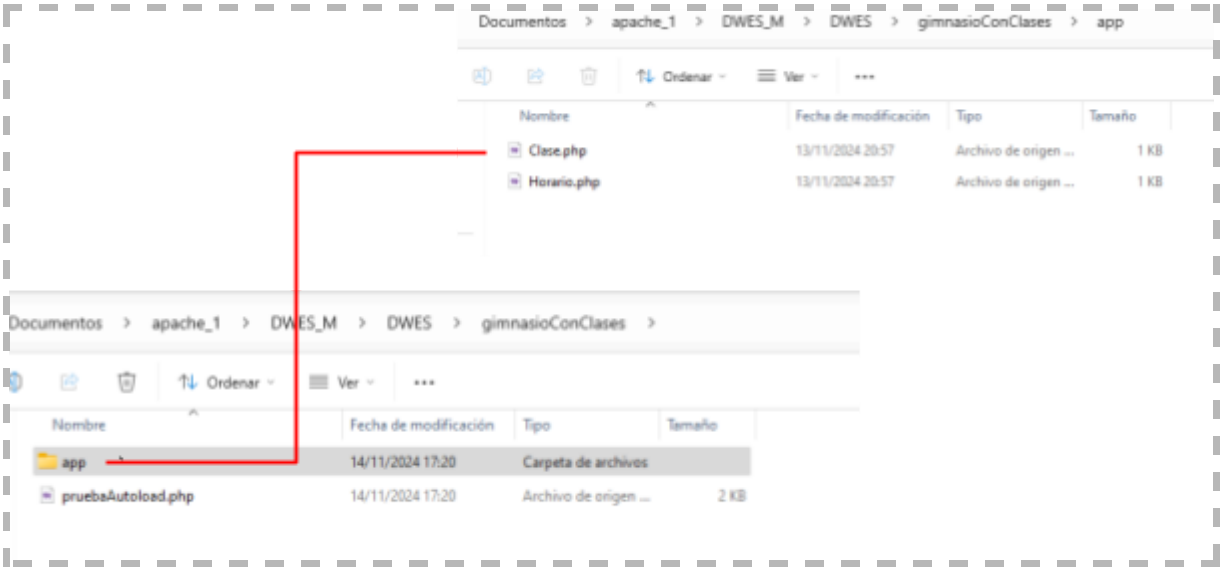
Otras carpetas que podemos crear son **test** para colocar las pruebas PhpUnit que luego realizaremos, **o** la carpeta **vendor** *donde se almacenarán las librerías del proyecto* (esta *carpeta es un estándard dentro de PHP,* ya que *Composer* la crea automáticamente).

Como hemos colocado todos nuestros recursos dentro de app, ahora nuestro autoload.php (el cual colocamos en la carpeta raíz) sólo va a buscar dentro de esa carpeta:

CAMBIAMOS

```php
<?php
session_start();
require_once('Horario.php');
require_once('Clases.php');
```

POR:




```php
<?php
session_start();
// Función de autocarga para buscar clases en la carpeta 'app'
spl_autoload_register(function ($clase) {
    require_once 'app/' . $clase . '.php';
});
```

Mecanismos de mantenimiento del estado

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

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

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

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

Esta cadena se puede almacenar en cualquier parte, como puede ser **la sesión del usuario, o una base de datos.**

A partir de ella, es posible **reconstruir el objeto** original utilizando la función **unserialize.**

ej. claseClases.php

```php
<?php
class Clase {
    public $nombre;
    public $horarios = []; // Array asociativo de objetos Horario
    public function __construct($nombre) {
        $this->nombre = $nombre;
    }
}
?>
```

claseHorario.php

```php
<?php
class Horario {
    public $dia;
    public $hora;
    public $plazasTotales;
    public $plazasDisponibles;
    public $plazasReservadas;
    public $reservado;
    public function __construct($dia, $hora, $plazasTotales) {
        $this->dia = $dia;
        $this->hora = $hora;
        $this->plazasTotales = $plazasTotales;
        $this->plazasDisponibles = $plazasTotales;
```

```
        $this->plazasReservadas = 0;
        $this->reservado = 0;
    }
}
?>
```

creaccionObjetosAlmacenamientoSesion.php

```
<?php
session_start();
require_once('claseHorario.php');
require_once('claseClases.php');
// Crear objetos de las clases
$yoga = new Clase('yoga');
$yoga->horarios['lunes'] = new Horario('lunes', '19:00', 20);
$yoga->horarios['miércoles'] = new Horario('miércoles', '08:00', 20);
$yoga->horarios['viernes'] = new Horario('viernes', '10:00', 20);

$zumba = new Clase('zumba');
$zumba->horarios['martes'] = new Horario('martes', '19:00', 20);
$zumba->horarios['jueves'] = new Horario('jueves', '08:00', 20);

$crossfit = new Clase('crossfit');
$crossfit->horarios['martes'] = new Horario('martes', '10:00', 20);
$crossfit->horarios['jueves'] = new Horario('jueves', '18:00', 20);
$crossfit->horarios['viernes'] = new Horario('viernes', '08:00', 20);
// Almacenar en sesión (serializar)
$_SESSION['horarios'] = serialize([$yoga, $zumba, $crossfit]);
echo "<br>";
print_r($_SESSION['horarios']);
    echo "<br>";
    //imprimo por pantalla los datos de $_SESSION['horarios'] de forma
legible
    $horarios = unserialize($_SESSION['horarios']);
// Creamos una tabla HTML para una mejor visualización
echo "<table>";
echo"<tr><th>Clase</th><th>Día</th><th>Hora</th><th>Plazas
Disponibles</th></tr>";
// Iteramos sobre las clases y sus horarios
foreach ($horarios as $clase) {
    echo "<tr>";
    echo "<td>" . $clase->nombre . "</td>";
    foreach ($clase->horarios as $horario) {
        echo "<tr>";
```

```
        echo "<td></td>"; // Celda vacía para alinear los horarios
        echo "<td>" . $horario->dia . "</td>";
        echo "<td>" . $horario->hora . "</td>";
        echo "<td>" . $horario->plazasDisponibles . "</td>";
        echo "</tr>";
    }
    echo "</tr>";
}
echo "</table>";
?>
```

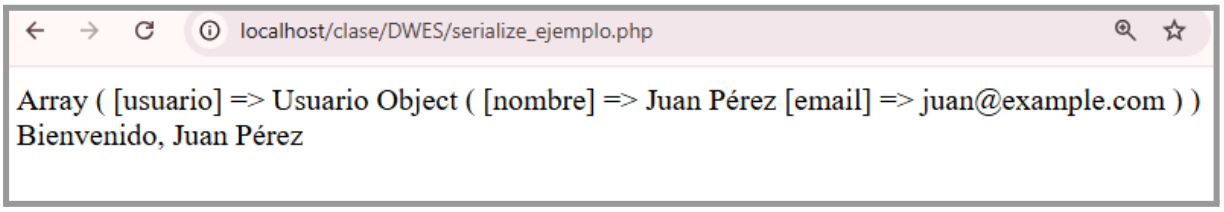
IMPORTANTE

- Las funciones serialize y unserialize se utilizan mucho con objetos, pero sirven para convertir en una cadena cualquier tipo de dato, excepto el tipo resource
- Cuando se aplican a un objeto, convierten y recuperan toda la información del mismo, incluyendo sus atributos privados. La única información que no se puede mantener utilizando estas funciones es la que contienen los atributos estáticos de las clases.
- Pero en PHP esto aún es más fácil. Los objetos que se añadan a la sesión del usuario son serializados automáticamente. Por tanto, no es necesario usar serialize ni unserialize. (VER EL SIGUIENTE EJEMPLO)
- Para poder de-serializar un objeto, debe estar definida su clase. Al igual que antes, si recuperamos la información almacenada en la sesión del usuario, no será necesario utilizar la función unserialize.

ej. serialize_ejemplo.php

```
<?php
class Usuario {
    public $nombre;
    public $email;
    public function __construct($nombre, $email) {
        $this->nombre = $nombre;
        $this->email = $email;
    }
}
// Crear un usuario
$usuario = new Usuario("Juan Pérez", "juan@example.com");
// Almacenar el usuario en la sesión
$_SESSION['usuario'] = $usuario;
print_r ($_SESSION);
// En otra página, recuperar el usuario de la sesión
if (isset($_SESSION['usuario'])) {
    $usuario_recuperado = $_SESSION['usuario'];
    echo "<br>Bienvenido, " . $usuario_recuperado->nombre;
}
```

```
?>
```



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

Gestión de Errores

PHP clasifica los errores que ocurren en diferentes niveles. Cada nivel se identifica con una constante. Por ejemplo:

E_ERROR: errores fatales, no recuperables. Se interrumpe el script.

E_WARNING: advertencias en tiempo de ejecución. El script no se interrumpe.

E_NOTICE: avisos en tiempo de ejecución.

Para la configuración de los errores podemos hacerlo de dos formas:

- A nivel de **php.ini:**

Mediante el parámetro *error_reporting (*que indica los niveles de errores a notificar)

display_errors: indica si mostrar o no los errores por pantalla. En entornos de producción es común ponerlo a off

-Mediante código con las siguientes funciones:

error_reporting(codigo) -> Controla qué errores notificar

set_error_handler(nombreManejador) -> Indica qué función se invocará cada vez que se encuentre un error. El manejador recibe como parámetros el nivel del error y el mensaje

```php
<?php
$divisor=0;
error_reporting(E_ALL & ~E_NOTICE & ~E_WARNING);
$resultado = $dividendo / $divisor;

error_reporting(E_ALL & ~E_NOTICE);
set_error_handler("miManejadorErrores");
$resultado = $dividendo / $divisor;
restore_error_handler(); // vuelve al anterior
```

```php
function miManejadorErrores($nivel, $mensaje) {
    switch($nivel) {
        case E_WARNING:
            echo "<strong>Warning</strong>: $mensaje.<br/>";
            break;
        default:
            echo "Error de tipo no especificado: $mensaje.<br/>";
    }
}
?>
```

```
Error de tipo no especificado: Undefined variable: dividendo.
Error de tipo no especificado: Undefined variable: divisor.
Error de tipo Warning: Division by zero.
```

Excepciones

Su funcionamiento es similar a Java, haciendo uso de un bloque ***try*** / ***catch*** / ***finally***. Si detectamos una situación anómala y queremos lanzar una excepción, deberemos realizar **throw *new* Exception** (adjuntando el mensaje que lo ha provocado).

La clase ***Exception*** es la clase padre de todas las excepciones. Su constructor recibe mensaje[,codigoError][,excepcionPrevia].

A partir de un objeto *Exception*, podemos acceder a los métodos **getMessage()**y **getCode()** para obtener el mensaje y el código de error de la excepción capturada.

```php
<?php

// Configurar el nivel de reporte de errores

error_reporting(E_ALL);

ini_set('display_errors', 1);

ini_set('log_errors', 1);

ini_set('error_log', 'errores.log');



// Función para manejar errores personalizados

function miManejadorErrores($nivel, $mensaje, $archivo, $linea)

{
```

```php
    $error = "[Nivel: $nivel] Error en $archivo en la línea $linea: $mensaje";
    error_log($error); // Registrar el error en el archivo de log
    echo "<strong>Error capturado:</strong> $mensaje<br/>";
}

// Registrar el manejador de errores
set_error_handler("miManejadorErrores");

try {
    // Bloque de código que podría generar errores o excepciones
    echo "Inicio del script.<br/>";

    // Generar un aviso (notice): variable indefinida
    echo $variableNoDefinida;

    // Generar una advertencia (warning): división por cero
    $resultado = 10 / 0;

    // Generar una excepción
    throw new Exception("Esto es una excepción.");

    echo "Este mensaje no se ejecutará debido a la excepción.<br/>";
} catch (DivisionByZeroError $e) {
    // Manejar el error de división por cero
        echo "<strong>Error:</strong> División por cero: " . $e->getMessage() . "<br/>";
    error_log("División por cero: " . $e->getMessage());
```

```php
} catch (Throwable $e) {

    // Manejar cualquier otra excepción

     echo "<strong>Excepción capturada:</strong> " . $e->getMessage() .
"<br/>";

    error_log("Excepción: " . $e->getMessage() . " en " . $e->getFile()
. ":" . $e->getLine());

} finally {

      // Código que se ejecutará siempre, independientemente de si
ocurrió un error o no

    echo "Bloque finally: limpieza de recursos.<br/>";

}



// Restaurar el manejador de errores predeterminado

restore_error_handler();

?>
```

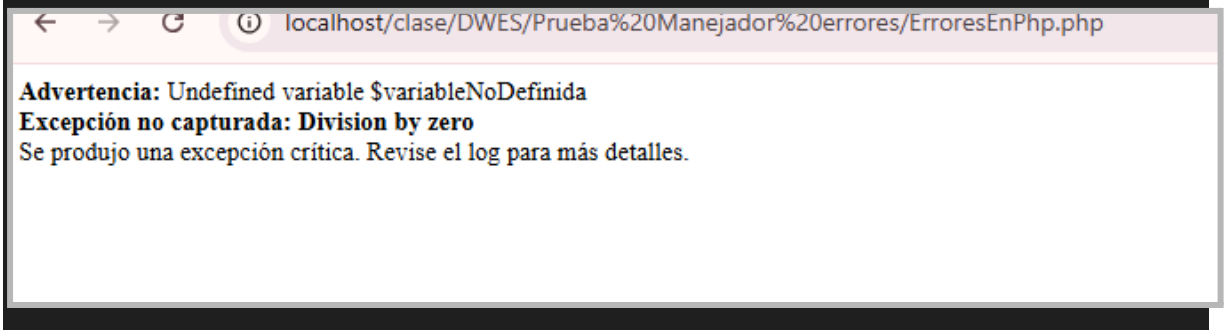
Inicio del script.
Error capturado: Undefined variable $variableNoDefinida
Error: División por cero: Division by zero
Bloque finally: limpieza de recursos.

Otro código posible sería

```php
<?php
// Configurar el nivel de reporte de errores
error_reporting(E_ALL); // Mostrar todos los errores, advertencias y
avisos
ini_set('display_errors', 1); // Mostrar errores en pantalla (útil para
desarrollo)
ini_set('log_errors', 1); // Activar registro de errores
ini_set('error_log', 'errores.log'); // Archivo donde se registrarán
los errores

// Manejador personalizado de errores
function miManejadorErrores($nivel, $mensaje, $archivo, $linea)
```

```php
{
    // Formato del mensaje de error
     $error = "[Nivel: $nivel] Error en $archivo en la línea $linea: $mensaje";

    // Decidir cómo manejar los errores según su nivel
    switch ($nivel) {
        case E_WARNING:
            echo "<strong>Advertencia:</strong> $mensaje<br/>";
            break;
        case E_NOTICE:
        case E_USER_NOTICE:
            echo "<strong>Aviso:</strong> $mensaje<br/>";
            break;
        default:
            echo "<strong>Error:</strong> $mensaje<br/>";
    }

    // Registrar el error en el archivo de log
    error_log($error);

    // Si el error es crítico, detener la ejecución
    if ($nivel === E_ERROR) {
            die("Se produjo un error crítico. Revise el log para más detalles.");
    }
}

// Registrar el manejador de errores
set_error_handler("miManejadorErrores");

// Registrar un manejador global para excepciones
function miManejadorExcepciones(Throwable $excepcion)
{
    $mensaje = "Excepción no capturada: " . $excepcion->getMessage();
    echo "<strong>".$mensaje." </strong><br/>";
        error_log($mensaje . " " . $excepcion->getFile() . ":" . $excepcion->getLine());
      die("Se produjo una excepción crítica. Revise el log para más detalles.");
}
set_exception_handler("miManejadorExcepciones");
```

```php
// Ejemplo de error: aviso (notice)
echo $variableNoDefinida;

// Ejemplo de error: advertencia (warning)
echo 10 / 0;

// Ejemplo de excepción
throw new Exception("Esto es una excepción no capturada");

// Restaurar manejadores predeterminados ( para revertir el uso del
//set_error_handler y set_exception_handler
restore_error_handler();
restore_exception_handler();
?>
```



El propio lenguaje ofrece un conjunto de excepciones ya definidas, las cuales podemos capturar. Se recomienda revisar la documentación oficial:

https://www.php.net/manual/es/class.exception.php

Creando excepciones

Para crear una excepción, la forma más corta es crear una clase que únicamente herede de *Exception.*

Si queremos, y es recomendable, p*odemos sobrecargar los métodos mágicos*, por ejemplo, sobrecargando el constructor y *llamando al constructor del padre, o reescribir el método* ***__toString*** *para cambiar su mensaje*:

En las aplicaciones reales, es muy común capturar una excepción de sistema y lanzar una de aplicación que hemos definido nostros. También podemos lanzar las excepciones sin necesidad de estar dentro de un try/catch.

```php
<?php
class MiExcepcion extends Exception {
    public function __construct($msj, $codigo = 0, Exception $previa =
null) {
        // código propio
        parent::__construct($msj, $codigo, $previa);
    }
```

```
    public function __toString() {
        return __CLASS__ . ": [{$this->code}]: {$this->message}\n";
    }
    public function miFuncion() {
        echo "Una función personalizada para este tipo de excepción\n";
    }
}
try {
    // Código de negocio que falla
} catch (Exception $e) {
    throw new AppException("AppException: ".$e->getMessage(),
$e->getCode(), $e);
}
```

IMPORTANTE Si definimos una excepción de aplicación dentro de un namespace, cuando referenciemos a ***Exception***, *deberemos referenciarla mediante su nombre totalmente cualificado (\Exception), o utilizando use*:

```
<?php
namespace \Dwes\Ejemplos;
class AppExcepcion extends \Exception {}
```

```
<?php
namespace \Dwes\Ejemplos;
use Exception;
class AppExcepcion extends Exception {}
```

Excepciones múltiples

Se pueden usar excepciones múltiples para comprobar diferentes condiciones. A la hora de capturarlas, se hace de más específica a más general.

```
<?php
$email = "ejemplo@ejemplo.com";
try {
    // Comprueba si el email es válido
    if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE) {
        throw new MiExcepcion($email);
    }
    // Comprueba la palabra ejemplo en la dirección email
    if(strpos($email, "ejemplo") !== FALSE) {
        throw new Exception("$email es un email de ejemplo no válido");
    }
} catch (MiExcepcion $e) {
```

```
    echo $e->miFuncion();
} catch(Exception $e) {
    echo $e->getMessage();
}
```

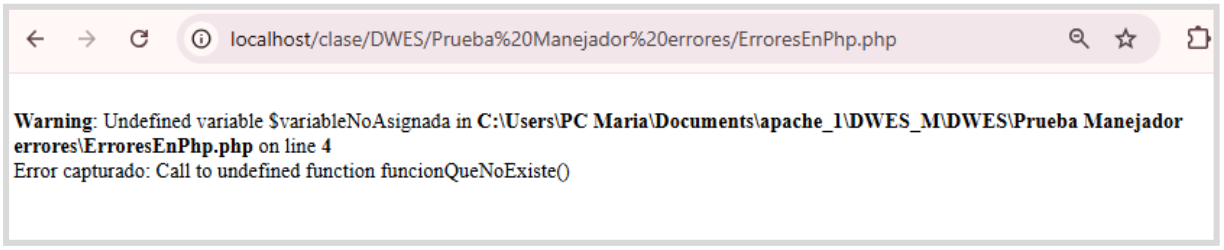
Si en el mismo catch queremos capturar varias excepciones, hemos de utilizar el operador |:

```
<?php
class MainException extends Exception {}
class SubException extends MainException {}

try {
    throw new SubException("Lanzada SubException");
} catch (MainException | SubException $e ) {
    echo "Capturada Exception " . $e->getMessage();
}
```

Desde PHP 7, existe el tipo ***Throwable***, el cual es un interfaz que implementan tanto los errores como las excepciones, y nos permite capturar los dos tipos a la vez:

Si sólo queremos capturar los errores fatales, podemos hacer uso de la clase ***Error:***

```
<?php
try {
    // Genera una notificación que no se captura
    echo $variableNoAsignada;
    // Error fatal que se captura
    funcionQueNoExiste();
} catch (Error $e) {
    echo "Error capturado: " . $e->getMessage();
}
```



SPL

Standard PHP Library es el conjunto de funciones y utilidades que ofrece PHP, como:

- Estructuras de datos

Pila, cola, cola de prioridad, lista doblemente enlazada, etc...

- Conjunto de iteradores diseñados para recorrer estructuras agregadas

  arrays, resultados de bases de datos, árboles XML, listados de directorios, etc.

documentación en https://www.php.net/manual/es/book.spl.php

También define un conjunto de excepciones que podemos utilizar para que las lancen nuestras aplicaciones (https://www.php.net/manual/es/spl.exceptions.php):

LogicException (extends Exception)

BadFunctionCallException

BadMethodCallException

DomainException

InvalidArgumentException

LengthException

OutOfRangeException

RuntimeException (extends Exception)

OutOfBoundsException

OverflowException

RangeException

UnderflowException

UnexpectedValueException