

## Clases y Objetos

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

Una clase es un plantilla que define las propiedades y métodos para poder crear objetos. De este manera, un objeto es una instancia de una clase.

Tanto las propiedades como los métodos se definen con una visibilidad (quien puede acceder)

- Privado - private: Sólo puede acceder la propia clase.
- Protegido - protected: Sólo puede acceder la propia clase o sus descendientes.
- Público - public: Puede acceder cualquier otra clase.

Para declarar una clase, se utiliza la palabra clave `class` seguido del nombre de la clase. Para instanciar un objeto a partir de la clase, se utiliza `new`:

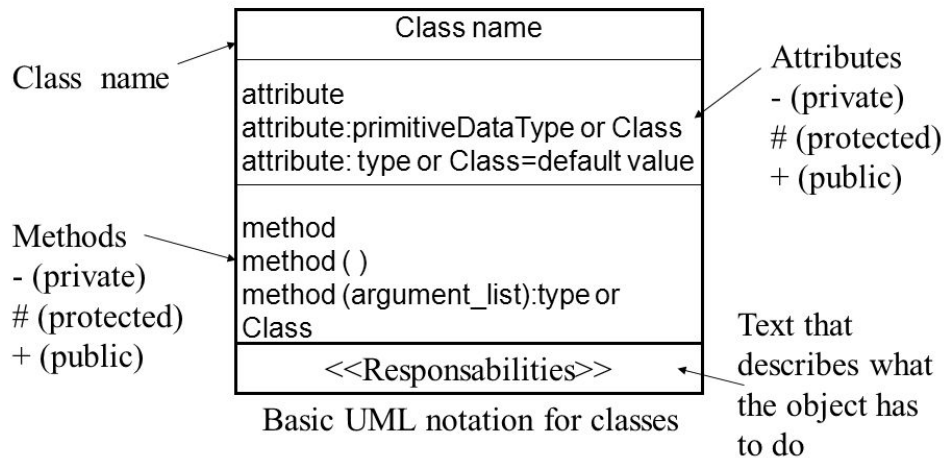
```
1  <?php
2  class NombreClase {
3      // propiedades
4      // y métodos
5  }
6
7  $obj = new NombreClase();
```

Todas las clases empiezan por **letra mayúscula**.

Cuando un proyecto crece, es normal modelar las clases mediante UML. Las clases se representan mediante un cuadrado, separando el nombre, de las propiedades y los métodos:

# Static Structure Diagrams

- **Class Diagram:** shows classes & relationships  
(Klassen-Diagramm & Beziehungen)



Una vez que hemos creado un objeto, se utiliza el operador `->` para acceder a una propiedad o un método:

```
1 | $objeto->propiedad;  
2 | $objeto->método(parámetros);
```

Si desde dentro de la clase, queremos acceder a una propiedad o método de la misma clase, utilizaremos la referencia `$this`;

```
1 | $this->propiedad;  
2 | $this->método(parámetros);
```

Así pues, como ejemplo, codificaríamos una persona en el fichero `Persona.php` como:

```

1  <?php
2  class Persona {
3      private string $nombre;
4
5      public function setNombre(string $nom) {
6          $this->nombre=$nom;
7      }
8
9      public function imprimir(){
10         echo $this->nombre;
11         echo '<br>';
12     }
13 }
14
15 $bruno = new Persona(); // creamos un objeto
16 $bruno->setNombre("Bruno Díaz");
17 $bruno->imprimir();

```

Aunque se pueden declarar varias clases en el mismo archivo, es una mala práctica. Así pues, cada fichero contendrá una sola clase, y se nombrará con el nombre de la clase.

## Encapsulación

Las propiedades se definen privadas o protegidas (si queremos que las clases heredadas puedan acceder).

Para cada propiedad, se añaden métodos públicos (*getter/setter*):

```

1  public setPropiedad(tipo $param)
2  public getPropiedad() : tipo

```

Las constantes se definen públicas para que sean accesibles por todos los recursos.

```

1  <?php
2  class MayorMenor {
3      private int $mayor;
4      private int $menor;
5
6      public function setMayor(int $may) {
7          $this->mayor = $may;
8      }
9
10     public function setMenor(int $men) {
11         $this->menor = $men;
12     }
13
14     public function getMayor() : int {
15         return $this->mayor;
16     }
17
18     public function getMenor() : int {
19         return $this->menor;
20     }
21 }

```

## Recibiendo y enviando objetos

Es recomendable indicarlo en el tipo de parámetros. Si el objeto puede devolver nulos se pone ? delante del nombre de la clase.

Los objetos que se envían y reciben como parámetros siempre se pasan por referencia.

```

1  <?php
2  function maymen(array $numeros) : ?MayorMenor {
3      $a = max($numeros);
4      $b = min($numeros);
5
6      $result = new MayorMenor();
7      $result->setMayor($a);
8      $result->setMenor($b);
9
10     return $result;
11 }
12
13 $resultado = maymen([1, 76, 9, 388, 41, 39, 25, 97, 22]);
14 echo "<br>Mayor: ".$resultado->getMayor();
15 echo "<br>Menor: ".$resultado->getMenor();

```

## Constructor

El constructor de los objetos se define mediante el método mágico `__construct`. Puede o no tener parámetros, pero sólo puede haber un único constructor.

```

1  <?php
2  class Persona {
3      private string $nombre;
4
5      public function __construct(string $nom) {
6          $this->nombre = $nom;
7      }
8
9      public function imprimir(){
10         echo $this->nombre;
11         echo '<br>';
12     }
13 }
14
15 $bruno = new Persona("Bruno Díaz");
16 $bruno->imprimir();

```

## Constructores en PHP 8

Una de las grandes novedades que ofrece PHP 8 es la simplificación de los constructores con parámetros, lo que se conoce como *promoción de las propiedades del constructor*.

Para ello, en vez de tener que declarar las propiedades como privadas o protegidas, y luego dentro del constructor tener que asignar los parámetros a estas propiedades, el propio constructor promociona las propiedades.

Veámoslo mejor con un ejemplo. Imaginemos una clase `Punto` donde queramos almacenar sus coordenadas:

```

1  <?php
2  class Punto {
3      protected float $x;
4      protected float $y;
5      protected float $z;
6
7      public function __construct(
8          float $x = 0.0,
9          float $y = 0.0,
10         float $z = 0.0
11     ) {
12         $this->x = $x;
13         $this->y = $y;
14         $this->z = $z;
15     }
16 }

```

En PHP 8, quedaría del siguiente modo (mucho más corto, lo que facilita su legibilidad):

```

1  <?php
2  class Punto {
3      public function __construct(
4          protected float $x = 0.0,
5          protected float $y = 0.0,
6          protected float $z = 0.0,
7      ) {}
8  }

```

### El orden importa

A la hora de codificar el orden de los elementos debe ser:

```

1  <?php
2  declare(strict_types=1);
3
4  class NombreClase {
5      // propiedades
6
7      // constructor
8
9      // getters - setters
10
11     // resto de métodos
12 }
13 ?>

```

## Clases estáticas

Son aquellas que tienen propiedades y/o métodos estáticos (también se conocen como *de clase*, por que su valor se comparte entre todas las instancias de la misma clase).

Se declaran con `static` y se referencian con `::`.

- Si queremos acceder a un método estático, se antepone el nombre de la clase: `Producto::nuevoProducto()`.
- Si desde un método queremos acceder a una propiedad estática de la misma clase, se utiliza la referencia `self`: `self::$numProductos`

```

1  <?php
2  class Producto {
3      const IVA = 0.23;
4      private static $numProductos = 0;
5
6      public static function nuevoProducto() {
7          self::$numProductos++;
8      }
9  }
10
11  Producto::nuevoProducto();
12  $impuesto = Producto::IVA;

```

También podemos tener clases normales que tengan alguna propiedad estática:

```

1  <?php
2  class Producto {
3      const IVA = 0.23;
4      private static $numProductos = 0;
5      private $codigo;
6
7      public function __construct(string $cod) {
8          self::$numProductos++;
9          $this->codigo = $cod;
10     }
11
12     public function mostrarResumen() : string {
13         return "El producto ".$this->codigo." es el número ".self::$numProductos;
14     }
15 }
16
17 $prod1 = new Producto("PS5");
18 $prod2 = new Producto("XBOX Series X");
19 $prod3 = new Producto("Nintendo Switch");
20 echo $prod3->mostrarResumen();

```

## Introspección

Al trabajar con clases y objetos, 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.

Un ejemplo de estas funciones puede ser el siguiente:



```

1  <?php
2  $p = new Producto("PS5");
3  if ($p instanceof Producto) {
4      echo "Es un producto";
5      echo "La clase es ".get_class($p);
6
7      class_alias("Producto", "Articulo");
8      $c = new Articulo("Nintendo Switch");
9      echo "Un articulo es un ".get_class($c);
10
11     print_r(get_class_methods("Producto"));
12     print_r(get_class_vars("Producto"));
13     print_r(get_object_vars($p));
14
15     if (method_exists($p, "mostrarResumen")) {
16         $p->mostrarResumen();
17     }
18 }

```

## Clonado

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`

Si queremos modificar el clonado por defecto, hay que definir el método mágico `__clone()` que se llamará después de copiar todas las propiedades.

## Herencia

PHP soporta herencia simple, de manera que una clase solo puede heredar de otra, no de dos clases a la vez. Para ello se utiliza la palabra clave `extends`. Si queremos que la clase A hereda de la clase B haremos:

```

1  class A extends B

```

El hijo hereda los atributos y métodos públicos y protegidos.

### Cada clase en un archivo

Como ya hemos comentado, deberíamos colocar cada clase en un archivo diferente para posteriormente utilizarlo mediante `include`. En los siguiente ejemplo los hemos colocado junto para facilitar su legibilidad.

Por ejemplo, tenemos una clase `Producto` y una `Tv` que hereda de `Producto`:

```
1  <?php
2  class Producto {
3      public $codigo;
4      public $nombre;
5      public $nombreCorto;
6      public $PVP;
7
8      public function mostrarResumen() {
9          echo "<p>Prod:". $this->codigo."</p>";
10     }
11 }
12
13 class Tv extends Producto {
14     public $pulgadas;
15     public $tecnologia;
16 }
```

Podemos utilizar las siguientes funciones para averiguar si hay relación entre dos clases:

- `get_parent_class(object): string`
- `is_subclass_of(object, string): bool`

```
1  <?php
2  $t = new Tv();
3  $t->codigo = 33;
4  if ($t instanceof Producto) {
5      echo $t->mostrarResumen();
6  }
7
8  $padre = get_parent_class($t);
9  echo "<br>La clase padre es: " . $padre;
10 $objetoPadre = new $padre;
11 echo $objetoPadre->mostrarResumen();
12
13 if (is_subclass_of($t, 'Producto')) {
14     echo "<br>Soy un hijo de Producto";
15 }
```

## Sobreescribir métodos

Podemos crear métodos en los hijos con el mismo nombre que el padre, cambiando su comportamiento. Para invocar a los métodos del padre

```
1  <?php
2  class Tv extends Producto {
3      public $pulgadas;
4      public $tecnologia;
5
6      public function mostrarResumen() {
7          parent::mostrarResumen();
8          echo "<p>TV ".$this->tecnologia." de ".$this->pulgadas."</p>";
9      }
10 }
```

## 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.

```

1  <?php
2  class Producto {
3      public function __construct(private string $codigo) { }
4
5      public function mostrarResumen() {
6          echo "<p>Prod:". $this->codigo. "</p>";
7      }
8  }
9
10 class Tv extends Producto {
11
12     public function __construct(
13         string $codigo,
14         private int $pulgadas,
15         private string $tecnologia)
16     {
17         parent::__construct($codigo);
18     }
19
20     public function mostrarResumen() {
21         parent::mostrarResumen();
22         echo "<p>TV ". $this->tecnologia. " de ". $this->pulgadas. "</p>";
23     }
24 }

```

## Clases abstractas

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.

```

1  <?php
2  // Clase abstracta
3  abstract class Producto {
4      private $codigo;
5      public function getCodigo() : string {
6          return $this->codigo;
7      }
8      // Método abstracto
9      abstract public function mostrarResumen();
10 }

```

Cuando una clase hereda de una clase abstracta, obligatoriamente debe implementar los métodos que tiene el padre marcados como abstractos.

```

1  <?php
2  class Tv extends Producto {
3      public $pulgadas;
4      public $tecnologia;
5
6      public function mostrarResumen() { //obligado a implementarlo
7          echo "<p>Código " . $this->getCodigo() . "</p>";
8          echo "<p>TV " . $this->tecnologia . " de " . $this->pulgadas . "</p>";
9      }
10 }
11
12 $t = new Tv();
13 echo $t->getCodigo();

```

## Clases finales

Son clases opuestas a abstractas, ya que evitan que se pueda heredar una clase o método para sobreescribirlo.

```

1  <?php
2  class Producto {
3      private $codigo;
4
5      public function getCodigo() : string {
6          return $this->codigo;
7      }
8
9      final public function mostrarResumen() : string {
10         return "Producto " . $this->codigo;
11     }
12 }
13
14 // No podremos heredar de Microondas
15 final class Microondas extends Producto {
16     private $potencia;
17
18     public function getPotencia() : int {
19         return $this->potencia;
20     }
21
22     // No podemos implementar mostrarResumen()
23 }

```

## 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`.

```

1  <?php
2  interface Nombreable {
3      // declaración de funciones
4  }
5  class NombreClase implements NombreInterfaz {
6      // código de la clase

```

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

```

1  <?php
2  interface Mostrable {
3      public function mostrarResumen() : string;
4  }
5
6  interface MostrableTodo extends Mostrable {
7      public function mostrarTodo() : string;
8  }
9
10 interface Facturable {
11     public function generarFactura() : string;
12 }
13
14 class Producto implements MostrableTodo, Facturable {
15     // Implementaciones de los métodos
16     // Obligatoriamente deberá implementar public function mostrarResumen, mostrarTodo y generarFactura
17 }

```

## 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.

```

1  <?php
2  $p1 = new Libro();
3  $p1->setNombre("Harry Potter");
4  $p1->setAutor("JK Rowling");
5  echo $p1;
6
7  // Method chaining
8  $p2 = new Libro();
9  $p2->setNombre("Patria")->setAutor("Aramburu");
10 echo $p2;

```

Para facilitarlo, vamos a modificar todos sus métodos mutadores (que modifican datos, *setters*, ...) para que devuelvan una referencia a *\$this*:

```

1  <?php
2  class Libro {
3      private string $nombre;
4      private string $autor;
5
6      public function getNombre() : string {
7          return $this->nombre;
8      }
9      public function setNombre(string $nombre) : Libro {
10         $this->nombre = $nombre;
11         return $this;
12     }
13
14     public function getAutor() : string {
15         return $this->autor;
16     }
17     public function setAutor(string $autor) : Libro {
18         $this->autor = $autor;
19         return $this;
20     }
21
22     public function __toString() : string {
23         return $this->nombre." de ".$this->autor;
24     }
25 }

```

## Métodos mágicos

Todas las clases PHP ofrecen un conjunto de métodos, también conocidos como *magic methods* que se pueden sobrescribir para sustituir su comportamiento. Algunos de ellos ya los hemos utilizado.

Ante cualquier duda, es conveniente consultar la [documentación oficial](#).

Los más destacables son:

- `__construct()`
- `__destruct()` → se invoca al perder la referencia. Se utiliza para cerrar una conexión a la BD, cerrar un fichero, ...
- `__toString()` → representación del objeto como cadena. Es decir, cuando hacemos `echo $objeto` se ejecuta automáticamente este método.

- `__get(propiedad)`, `__set(propiedad, valor)` → Permitiría acceder a las propiedad privadas, aunque siempre es más legible/mantenible codificar los *getter/setter*.
- `__isset(propiedad)`, `__unset(propiedad)` → Permite averiguar o quitar el valor a una propiedad.
- `__sleep()`, `__wakeup()` → Se ejecutan al recuperar (*unserialize*<sup>^</sup>) o almacenar un objeto que se serializa (*serialize*), y se utilizan para permite definir qué propiedades se serializan.
- `__call()`, `__callStatic()` → Se ejecutan al llamar a un método que no es público. Permiten sobrecargar métodos.

## 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 respetando 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:



```

1  <?php
2  namespace Dwes\Ejemplos;
3
4  const IVA = 0.21;
5
6  class Producto {
7      public $nombre;
8
9      public function muestra() : void {
10         echo "<p>Prod:" . $this->nombre . "</p>";
11     }
12 }

```

## 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`

```

1  <?php
2  namespace Dwes\Ejemplos;
3
4  include_once("Producto.php");
5
6  echo IVA; // sin cualificar
7  echo Utilidades\IVA; // acceso cualificado. Daría error, no existe \Dwes\Ejemplos\Utilidades\IVA
8  echo \Dwes\Ejemplos\IVA; // totalmente cualificado
9
10 $p1 = new Producto(); // lo busca en el mismo namespace y encuentra \Dwes\Ejemplos\Producto
11 $p2 = new Model\Producto(); // daría error, no existe el namespace Model. Está buscando \Dwes\Ejemplos\Model\
12 $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:

```
1  <?php
2  include_once("Dwes\Ejemplo\Producto.php");
3
4  use const Dwes\Ejemplos\IVA;
5  use \Dwes\Ejemplos\Producto;
6
7  echo IVA;
8  $p1 = new Producto();
```

En resumen, `use` permite acceder sin cualificar a recursos que están en otro *namespace*. Si estamos en el mismo espacio de nombre, no necesitamos `use`.

## Organización

Todo proyecto, conforme crece, necesita organizar su código fuente. Se plantea una organización en la que los archivos que interactúan con el navegador se colocan en la 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 referencia: 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

¿No es tedioso tener que hacer el `include` de las clases? El *autoload* viene al rescate.

Así pues, 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`

```

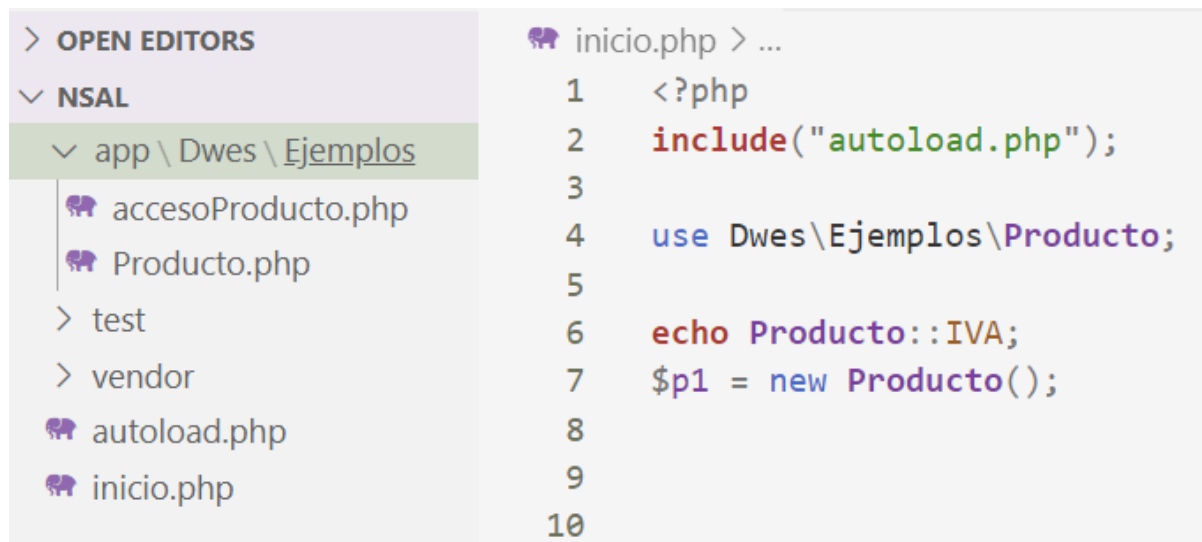
1  <?php
2  spl_autoload_register( function( $nombreClase ) {
3      include_once $nombreClase.'.php';
4  } );
5  ?>

```

### ¿Por qué se llaman *autoload*?

Porque antes se realizaba mediante el método mágico `__autoload()`, el cual está *deprecated* desde PHP 7.2

Y ¿cómo organizamos ahora nuestro código aprovechando el *autoload*?



```

> OPEN EDITORS
  NSAL
    app \ Dwes \ Ejemplos
      accesoProducto.php
      Producto.php
    > test
    > vendor
    autoload.php
    inicio.php

inicio.php > ...
1  <?php
2  include("autoload.php");
3
4  use Dwes\Ejemplos\Producto;
5
6  echo Producto::IVA;
7  $p1 = new Producto();
8
9
10

```

*Organización con 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 estándar 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:

```
1  <?php
2  spl_autoload_register( function( $nombreClase ) {
3      include_once "app/".$nombreClase.'.php';
4  } );
5  ?>
```

## 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.

Podéis comprobar el listado completo de constantes de <https://www.php.net/manual/es/errorfunc.constants.php>

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

- A nivel de php.ini:
  - error\_reporting: indica los niveles de errores a notificar
    - error\_reporting = E\_ALL & ~E\_NOTICE -> Todos los errores menos los avisos en tiempo de ejecución.
  - 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 que función se invocará cada vez que se encuentre un error. El manejador recibe como parámetros el nivel del error y el mensaje

A continuación tenemos un ejemplo mediante código:

```

1  <?php
2  error_reporting(E_ALL & ~E_NOTICE & ~E_WARNING);
3  $resultado = $dividendo / $divisor;
4
5  error_reporting(E_ALL & ~E_NOTICE);
6  set_error_handler("miManejadorErrores");
7  $resultado = $dividendo / $divisor;
8  restore_error_handler(); // vuelve al anterior
9
10 function miManejadorErrores($nivel, $mensaje) {
11     switch($nivel) {
12         case E_WARNING:
13             echo "<strong>Warning</strong>: $mensaje.<br/>";
14             break;
15         default:
16             echo "Error de tipo no especificado: $mensaje.<br/>";
17     }
18 }

```

```

1  Error de tipo no especificado: Undefined variable: dividendo.
2  Error de tipo no especificado: Undefined variable: divisor.
3  Error de tipo Warning: Division by zero.

```

## Excepciones

La gestión de excepciones forma parte desde PHP 5. 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).

```

1  <?php
2  try {
3      if ($divisor == 0) {
4          throw new Exception("División por cero.");
5      }
6      $resultado = $dividendo / $divisor;
7  } catch (Exception $e) {
8      echo "Se ha producido el siguiente error: ".$e->getMessage();
9  }

```

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.

El propio lenguaje ofrece un conjunto de excepciones ya definidas, las cuales podemos capturar (y lanzar desde PHP 7). Se recomienda su consulta en la [documentación oficial](#).

## Creando excepciones

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

```
1  <?php
2  class HolaExcepcion extends Exception {}
```

Si

queremos, y es recomendable dependiendo de los requisitos, podemos sobrecargar los métodos mágicos, por ejemplo, sobrecargando el constructor y llamando al constructor del padre, o rescribir el método `__toString` para cambiar su mensaje:

```
1  <?php
2  class MiExcepcion extends Exception {
3      public function __construct($msj, $codigo = 0, Exception $previa = null) {
4          // código propio
5          parent::__construct($msj, $codigo, $previa);
6      }
7      public function __toString() {
8          return __CLASS__ . ": [{ $this->code }]: { $this->message }\n";
9      }
10     public function miFuncion() {
11         echo "Una función personalizada para este tipo de excepción\n";
12     }
13 }
```

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

### [Mediante nombre totalmente cualificado](#)

```
1  <?php
2  namespace \Dwes\Ejemplos;
3
4  class AppExcepcion extends \Exception {}
```

### [Mediante use](#)



```

1  <?php
2  namespace \Dwes\Ejemplos;
3
4  use Exception;
5
6  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.

```

1  <?php
2  $email = "ejemplo@ejemplo.com";
3  try {
4      // Comprueba si el email es válido
5      if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE) {
6          throw new MiExcepcion($email);
7      }
8      // Comprueba la palabra ejemplo en la dirección email
9      if(strpos($email, "ejemplo") !== FALSE) {
10         throw new Exception("$email es un email de ejemplo no válido");
11     }
12 } catch (MiExcepcion $e) {
13     echo $e->miFuncion();
14 } catch (Exception $e) {
15     echo $e->getMessage();
16 }

```

## Autoevaluación

¿Qué pasaría al ejecutar el siguiente código?

```

1  <?php
2  class MainException extends Exception {}
3  class SubException extends MainException {}
4
5  try {
6      throw new SubException("Lanzada SubException");
7  } catch (MainException $e) {
8      echo "Capturada MainException " . $e->getMessage();
9  } catch (SubException $e) {
10     echo "Capturada SubException " . $e->getMessage();
11 } catch (Exception $e) {
12     echo "Capturada Exception " . $e->getMessage();
13 }

```

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



```

1  <?php
2  class MainException extends Exception {}
3  class SubException extends MainException {}
4
5  try {
6      throw new SubException("Lanzada SubException");
7  } catch (MainException | SubException $e) {
8      echo "Capturada Exception " . $e->getMessage();
9  }

```

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:

```

1  <?php
2  try {
3      // tu codigo
4  } catch (Throwable $e) {
5      echo 'Forma de capturar errores y excepciones a la vez';
6  }

```

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

```

1  <?php
2  try {
3      // Genera una notificación que no se captura
4      echo $variableNoAsignada;
5      // Error fatal que se captura
6      funcionQueNoExiste();
7  } catch (Error $e) {
8      echo "Error capturado: " . $e->getMessage();
9  }

```

## Relanzar excepciones

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

```

1  <?php
2  class AppException extends Exception {}
3
4  try {
5      // Código de negocio que falla
6  } catch (Exception $e) {
7      throw new AppException("AppException: ".$e->getMessage(), $e->getCode(), $e);
8  }

```

## 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.

Podéis consultar la documentación

en <https://www.php.net/manual/es/book.spl.php> o ver algunos ejemplos

en <https://diego.com.es/tutorial-de-la-libreria-spl-de-php>

También define un conjunto de excepciones que podemos utilizar para que las lancen nuestras aplicaciones:

- LogicException (extends Exception)
  - BadFunctionCallException
  - BadMethodCallException
  - DomainException
  - InvalidArgumentException
  - LengthException
  - OutOfRangeException
- RuntimeException (extends Exception)
  - OutOfBoundsException
  - OverflowException
  - RangeException
  - UnderflowException
  - UnexpectedValueException

También podéis consultar la documentación de estas excepciones en <https://www.php.net/manual/es/spl.exceptions.php>.