unidad didáctica 03

PHP Orientado a Objetos

1. duración y criterios de evaluación

Duración estimada: 18 sesiones

Resultado de aprendizaje y criterios de evaluación:

5. Desarrolla aplicaciones Web identificando y aplicando mecanismos para separar el código de presentación de la lógica de negocio.

   a) Se han identificado las ventajas de separar la lógica de negocio de los aspectos de presentación de la aplicación.

   b) Se han analizado tecnologías y mecanismos que permiten realizar esta separación y sus características principales.

   c) Se han utilizado objetos y controles en el servidor para generar el aspecto visual de la aplicación web en el cliente.

   d) Se han utilizado formularios generados de forma dinámica para responder a los eventos de la aplicación Web.

   e) Se han escrito aplicaciones Web con mantenimiento de estado y separación de la lógica de negocio.

   f) Se han aplicado los principios de la programación orientada a objetos.

   g) Se ha probado y documentado el código.

2. clases y objetos

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

Una **clase** es una **plantilla** que define las atributos y métodos para poder crear objetos.

De este manera, un **objeto** es una **instancia de una clase**.

Tanto los atributos como los métodos se definen con una visibilidad (quién 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.

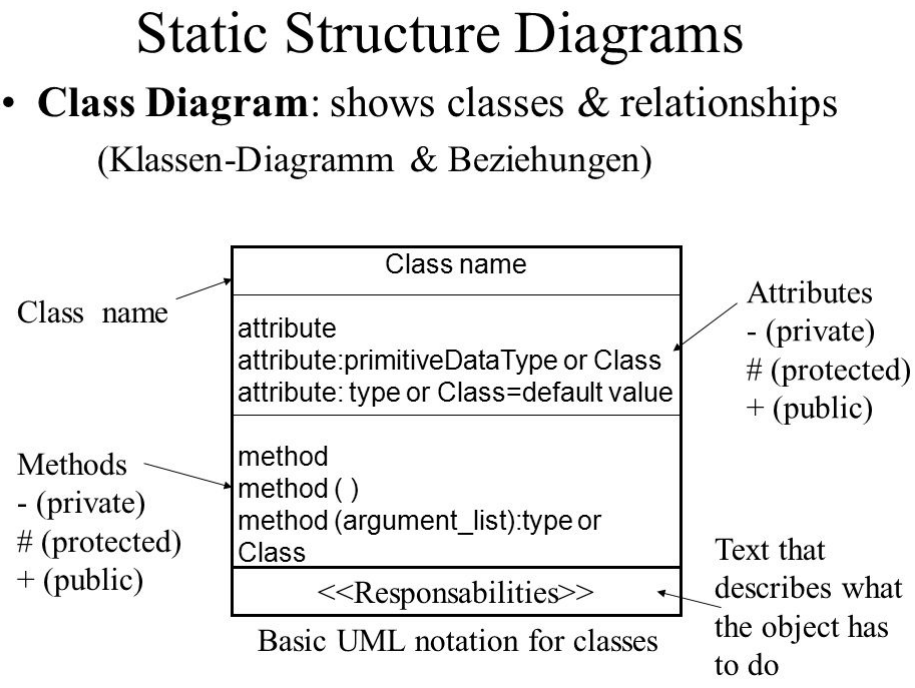
Para declarar una clase, se utiliza la palabra clave `class` seguido del nombre de la clase.

Para instanciar un objeto a partir de la clase, se utiliza `new`:

```php
<?php
    class NombreClase {
    // atributos
    // y métodos
    }

    $ob = new NombreClase();
```

> **clases con mayúscula**:
>
> Todas las clases empiezan por letra mayúscula.

Cuando un proyecto crece, es normal modelar las clases mediante UML (¿recordáis *Entornos de Desarrollo*?). La clases se representan mediante un cuadrado, separando el nombre, de los atributos y los métodos:



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

```
$objeto->atributo;
$objeto->método(parámetros);
```

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

```
$this->atributo;
$this->método(parámetros);
```

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

```
<?php
    class Persona {
        private string $nombre;

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

        public function imprimir(){
            echo $this->nombre;
            echo '<br>';
        }
    }

    $bruno = new Persona(); // creamos un objeto
    $bruno->setNombre("Bruno Díaz");
    $bruno->imprimir();
```

Aunque se pueden declarar varias clases en el mismo archivo, es una mala práctica. Así pues, **cada fichero**:

- **contedrá una sola clase**, y
- **se nombrará con el nombre de la clase**.

3. encapsulación

Los atributos se definen *privados* o *protegidos* (si queremos que las clases heredadas puedan acceder).

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

```
public function setAtributo (tipo $param);

public function getAtributo () : tipo;
```

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

```
<?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;
     }
  }
```

3.1. 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.

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

```
<?php
  function maymen(array $numeros) : ?MayorMenor {
    $a = max($numeros);
    $b = min($numeros);

    $result = new MayorMenor();
    $result->setMayor($a);
    $result->setMenor($b);

```

```php
    return $result;
  }

  $resultado =  maymen( [1,76,9,388,41,39,25,97,22] );

  echo "<br>Mayor: ".$resultado->getMayor();
  echo "<br>Menor: ".$resultado->getMenor();
```

4. 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**.

```php
<?php
  class Persona {
     private string $nombre;

     public function __construct(string $nom) {
         $this->nombre = $nom;
     }

     public function imprimir(){
       echo $this->nombre;
       echo '<br>';
     }
  }

  $bruno = new Persona("Bruno Díaz");
  $bruno->imprimir();
```

4.1. constructores en PHP8

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

Para ello, en vez de tener que declarar los atributos como privados o protegidos, y luego dentro del constructor tener que asignar los parámetros a estos atributos, el propio constructor promociona los atributos.

Veámoslo mejor con un ejemplo. Imaginemos una clase `Punto` donde queramos almacenar sus coordenadas. Así quedaría en **PHP7** o anteriores:

```php
<?php
    class Punto {
        protected float $x;
        protected float $y;
        protected float $z;

        public function __construct(
            float $x = 0.0,
            float $y = 0.0,
            float $z = 0.0
        ) {
            $this->x = $x;
            $this->y = $y;
            $this->z = $z;
        }
    }
```

En **PHP8**, quedaría del siguiente modo (mucho más corto, lo que facilita su legibilidad):

```php
<?php
    class Punto {
        public function __construct(
            protected float $x = 0.0,
            protected float $y = 0.0,
            protected float $z = 0.0,
        ) {}
    }
```

el orden importa:

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

```php
<?php
  declare(strict_types=1);

  class NombreClase {
     // atributos

     // constructor

     // getters - setters

     // resto de métodos

  }
?>
```

5. clases estáticas

Son aquellas que tienen atributos (*o propiedades*) y/o métodos estáticos (también se conocen como **de clase**, porque 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 como sigue: `Producto::nuevoProducto()`.
- Si desde un método queremos acceder a un **atributo estático** de la misma clase, se utiliza la referencia `self` como sigue: `self::$numProductos`.

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

     public static function nuevoProducto() {
        self::$numProductos++;
     }
  }

  Producto::nuevoProducto();
  $impuesto = Producto::IVA;
```

También podemos tener clases normales que tengan algun atributo estático:

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

```
El producto Nintendo Switch es el número 3
```

6. 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`: devuelve *true* si la *clase/método/atributo* está definida.
- `get_class_methods` / `get_class_vars` / `get_object_vars`: devuelve un array con los nombres de los *métodos/atributos* de una clase / atributos de un objeto que son accesibles desde dónde se hace la llamada.

Un ejemplo de estas funciones puede ser el siguiente:

```php
<?php
    $p = new Producto("PS5");
    if ($p instanceof Producto) {
        echo "Es un producto.<br/>";
        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"));
        echo "<br/>";
        print_r(get_class_vars("Producto"));
        echo "<br/>";
        print_r(get_object_vars($p));
        echo "<br/>";
        if (method_exists($p, "mostrarResumen")) {
            echo $p->mostrarResumen();
        }
    }
```

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`.

PHP nos permite crear un método que se llamará cuando ejecutemos el operador clone. Este método puede entre otras cosas inicializar algunos atributos.

Si no se define el método `__clone()` se hará una copia idéntica del objeto que le pasamos como parámetro al operador clone. Veamos un ejemplo: Crearemos una clase Persona que tenga como atributos su nombre y edad, definiremos los métodos para cargar y retornar los valores de sus atributos. Haremos que cuando clonemos un objeto de dicha clase la edad de la persona se fije con cero.

```php
<?php
 class Persona {
   private $nombre;
   private $edad;
   public function fijarNombreEdad($nom,$ed) {
     $this->nombre=$nom;
     $this->edad=$ed;
   }
   public function getNombre() {
     return $this->nombre;
   }
   public function getEdad() {
     return $this->edad;
   }
   public function __clone() {
     $this->edad=0;
   }
 }
 $persona1=new Persona();
 $persona1->fijarNombreEdad('Juan',20);
 echo 'Datos de $persona1:';
 echo $persona1->retornarNombre().' - '.$persona1->retornarEdad().'<br>';
 $persona2=clone($persona1);
 echo 'Datos de $persona2:';
 echo $persona2->retornarNombre().' - '.$persona2->retornarEdad().'<br>';
?>
```

El método `__clone` se ejecutará cuando llamemos al operador `clone` para esta clase:

```php
public function __clone() {
 $this->edad=0;
}
```

Es decir cuando realicemos la asignación:

```php
$persona2 = clone($persona1);
```

inicialmente se hace una copia idéntica de $persona1 pero luego se ejecuta el método __clone con lo que el atributo $edad se modifica.

Si queremos que una clase no pueda clonarse simplemente podemos implementar el siguiente código en el método __clone():

```php
public function __clone() {
 die('No esta permitido clonar objetos de esta clase');
}
```

Más información en https://www.php.net/manual/es/language.oop5.cloning.php

7. herencia

PHP soporta herencia simple, de manera que una clase **solo puede heredar de otra clase**, no de dos o más 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
```

La clase hija hereda los atributos y métodos públicos y protegidos de la clase madre.

> **cada clase en un archivo**:
>
> Como ya hemos comentado, deberíamos colocar cada clase en un archivo diferente para posteriormente utilizarlo mediante `include`. En los siguientes ejemplos los hemos colocado juntos para facilitar su legibilidad.

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

```
1  <?php
2      class Producto {
3          protected $codigo;
4          protected $nombre;
5          protected $nombreCorto;
6          protected $PVP;
7
8          public function mostrarResumen() {
9              echo "<p>Prod:".$this->codigo."</p>";
10         }
11     }
12
13     class Tv extends Producto {
14         protected $pulgadas;
15         protected $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      $madre = get_parent_class($t);
9      echo "<br>La clase madre es: " . $madre;
10     $objetoMadre = new $madre;
11     echo $objetoMadre->mostrarResumen();
12
```

```
    if (is_subclass_of($t, 'Producto')) {
        echo "<br>Soy una hija de Producto";
}
```

7.1. sobreescribir métodos

Podemos crear métodos en las clase hijas con el mismo nombre que la clase madre, cambiando su comportamiento. Para invocar a los métodos de la clase madre: `parent::nombreMetodo()`.

```
<?php
class Tv extends Producto {
   public $pulgadas;
   public $tecnologia;

   public function mostrarResumen() {
      parent::mostrarResumen();
      echo "<p>TV ".$this->tecnologia." de ".$this->pulgadas."</p>";
   }
}
```

7.2. constructor en clases hijas

En cambio, si lo definimos en la clase hija, hemos de invocar al de la clase madre de manera explícita.

PHP 7:

```
<?php
  class Producto {
    public string $codigo;

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

    public function mostrarResumen() {
      echo "<p>Prod:".$this->codigo."</p>";
    }
  }

  class Tv extends Producto {
    public $pulgadas;
    public $tecnologia;

    public function __construct(string $codigo, int $pulgadas, string
$tecnologia) {
      parent::__construct($codigo);
      $this->pulgadas = $pulgadas;
      $this->tecnologia = $tecnologia;
    }

    public function mostrarResumen() {
      parent::mostrarResumen();
      echo "<p>TV ".$this->tecnologia." de ".$this->pulgadas."</p>";
```

```
    }
}
```

PHP 8:

```php
<?php
  class Producto {
    public function __construct(private string $codigo) { }

    public function mostrarResumen() {
      echo "<p>Prod:".$this->codigo."</p>";
    }
  }

  class Tv extends Producto {

    public function __construct(
      string $codigo,
      private int $pulgadas,
      private string $tecnologia)
    {
      parent::__construct($codigo);
    }

    public function mostrarResumen() {
      parent::mostrarResumen();
      echo "<p>TV ".$this->tecnologia." de ".$this->pulgadas."</p>";
    }
  }
```

8. clases abstractas

Las clases abstractas obligan a heredar de estas clases, ya que no se permite su instanciación; sí se permite a las clases heredadas (siempre que no estén también definidas como abstractas). Se define mediante `abstract class NombreClase {`.

Una clase abstracta puede contener atributos y métodos no-abstractos, y/o métodos abstractos.

```php
<?php
  // Clase abstracta
  abstract class Producto {
     private $codigo;

     public function getCodigo() : string {
        return $this->codigo;
     }

     // Método abstracto
     abstract public function mostrarResumen();
  }
```

Cuando una clase hereda de una clase abstracta, **obligatóriamente debe implementar los métodos** que tiene la clase madre marcados como abstractos.

```php
<?php
  class Tv extends Producto {
    public $pulgadas;
    public $tecnologia;

    public function mostrarResumen() { //obligado a implementarlo
      echo "<p>Código ".$this->getCodigo()."</p>";
      echo "<p>TV ".$this->tecnologia." de ".$this->pulgadas."</p>";
    }
  }

  $t = new Tv();
  echo $t->getCodigo();
```

9. clases finales

Son clases opuestas a abstractas, ya que evitan que se pueda heredar una clase o método para sobreescribirlo. Se define mediante `final class NombreClase {`.

```php
<?php
  class Producto {
    private $codigo;

    public function getCodigo() : string {
        return $this->codigo;
    }

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

  // No podremos heredar de Microondas
  final class Microondas extends Producto {
    private $potencia;

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

    // No podemos implementar mostrarResumen()
  }
```

10. 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`.

```php
<?php
    interface Nombreable {
        // declaración de funciones
    }
    class NombreClase implements NombreInterfaz {
        // código de la clase
```

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).

```php
<?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,
     //    - public function mostrarTodo y
     //    - public function generarFactura
  }
```

11. 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; o sea, consiste en llamar varios métodos de un objeto en una misma línea, basta con añadir al final de cada método: `return $this;` Con el siguiente ejemplo se entiende fácilmente.

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

```php
<?php
    class Coche {
        public function lavar(){
            echo "Coche lavado.\n";
            return $this;
        }

        public function encerar(){
            echo "Coche encerado.\n";
            return $this;
        }
    }

    // ejemplo llamada a los métodos encadenados
    $coche = new Coche;
    $coche->lavar()->encerar();
```

```
// mostrará
Coche lavado.
Coche encerado.
```

Otro ejemplo:

```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;
```

```
20      }
21
22      public function __toString() : string {
23          return $this->nombre." de ".$this->autor;
24      }
25  }
```

12. métodos mágicos

Todas las clases PHP ofrecen un conjunto de métodos, también conocidos como *magic methods* que se pueden sobreescribir 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(atributo)`, `set(atributo, valor)` → Permitiría acceder a los atributos privados, aunque siempre es más legible/mantenible codificar los getter/setter.
- `__isset(atributo)`, `__unset(atributo)` → Permite averiguar o quitar el valor a un atributo.
- `sleep()`, `wakeup()` → Se ejecutan al recuperar (*unserialize^*) o almacenar un objeto que se serializa (*serialize*), y se utilizan para permite definir qué atributos se serializan.
- `call()`, `callStatic()` → Se ejecutan al llamar a un método que no es público. Permiten sobrecargar métodos.

13. 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) en un ámbito restringido; evitando colisiones o conflictos de nombre con otros elementos que se pueden crear más adelante (o con algunas librerías que utilices).

> **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>";
        }
    }
```

13.1. 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");

  // sin cualificar
  echo IVA;

  // acceso cualificado.
  // Daría error, no existe \Dwes\Ejemplos\Utilidades\IVA
```

```php
echo Utilidades\IVA;

// totalmente cualificado
echo \Dwes\Ejemplos\IVA;

$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
```

> video

13.1.1. use

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:

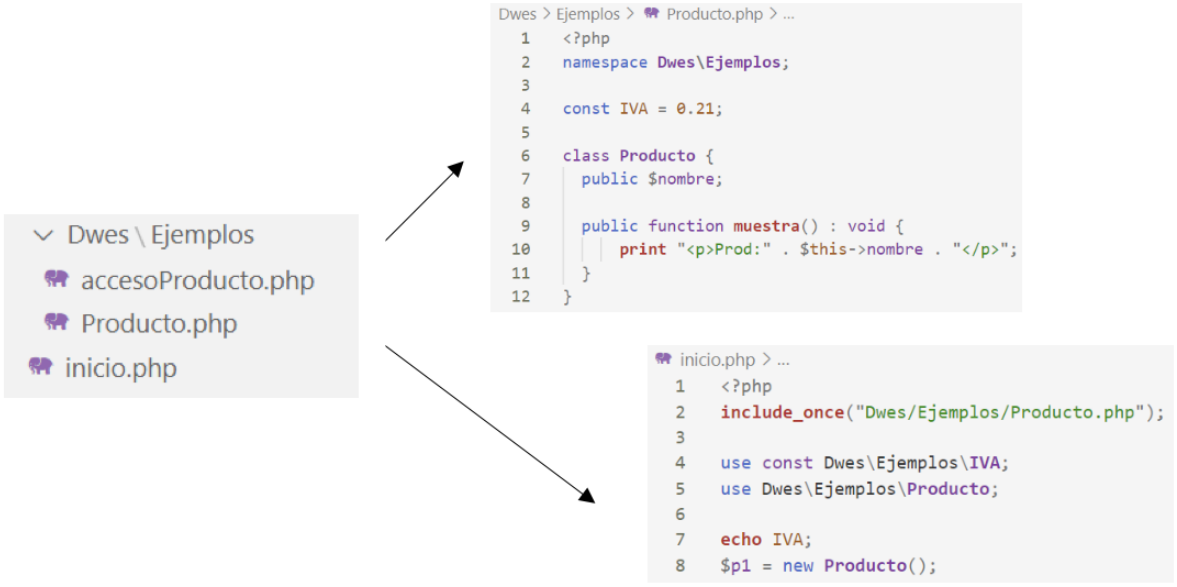
```php
<?php
    include_once("Dwes\Ejemplo\Producto.php");

    use const Dwes\Ejemplos\IVA;
    use \Dwes\Ejemplos\Producto;

    echo IVA;
    $p1 = new Producto();
```

> **to use or not to use**
>
> 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`.

> video

13.2. 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, 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.

13.3. 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`.

```php
<?php
    spl_autoload_register( function( $nombreClase ) {
        include_once $nombreClase.'.php';
    } );
?>
```

por qué se llama 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?

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:

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

autoload y rutas erróneas:

En *Ubuntu* al hacer el include de la clase que recibe como parámetro, las barras de los namespace (\ ) son diferentes a las de las rutas (/). Por ello, es mejor que utilicemos el fichero autoload:

```php
<?php
    spl_autoload_register( function( $nombreClase ) {
      $ruta = "app\\".$nombreClase.'.php';
      $ruta = str_replace("\\", "/", $ruta); // Sustituimos las barras
      include_once $ruta';
      } );
?>
```

video

Aquí encontramos un pequeño video en el que https://www.youtube.com/watch?v=lP1hbkNA2uc

14. 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.
        - `rerror_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 qué 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:

Funciones para la gestión de errores:

```php
<?php
    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

    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/>";
      }
    }
```

Consola:

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

15. 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).

```php
<?php
    try {
        if ($divisor == 0) {
            throw new Exception("División por cero.");
        }
        $resultado = $dividendo / $divisor;
    } catch (Exception $e) {
        echo "Se ha producido el siguiente error: ".$e->getMessage();
    }
```

La clase `Exception` es la clase madre 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.

15.1. creando excepciones

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

```php
<?php
  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 de la clase madre, o rescribir el método `__toString` para cambiar su mensaje:

```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";
    }
  }
```

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`:

Mediante nombre totalmente cualificado

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

    class AppExcepcion extends \Exception {}
```

Mediante use

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

    use Exception;

    class AppExcepcion extends Exception {}
```

15.2. 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
<?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();
    }
```

autoevaluación:

¿Qué pasaría al ejecutar el siguiente código?

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

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

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

```php
<?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:

```php
<?php
    try {
        // tu codigo
    } catch (Throwable $e) {
        echo 'Forma de capturar errores y excepciones a la vez';
    }
```

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

```php
<?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();
    }
```

15.3. 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`.

```php
<?php
    class AppException extends Exception {}

    try {
        // Código de negocio que falla
    } catch (Exception $e) {
        throw new AppException("AppException: ".$e->getMessage(), $e->getCode(), $e);
    }
```

16. 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.

17. referencias

- Manual de PHP
- Manual de OO en PHP - www.desarrolloweb.com
- videos Curso PHP 8 desde cero (Actulizado):