# Object-Oriented Programming with PHP

DWES UD3

# 1.- OOP Introduction

Characteristics of OOP:

- Inheritance: Creating a class from another. Inherits behavior and characteristics.
- Abstraction: Externally, the class only shows the methods (**interface**), not how things are done.
- Polymorphism and Overloading: Methods can have different behaviors depending on how they are used.
- Encapsulation: Data and the code that uses them are together.

# 1.- OOP Introduction

**Classes**

**Properties** (Attributes): Store information about the state of the object they belong to. Their value can differ among objects of the same class.

**Methods**: Contain executable code and define the actions of the object. Similar to a function, they can receive parameters and return values.

**Instance**: Having a defined class and creating an object of that class is called an instance of the class.

# 1.- OOP Introduction

Advantages of OOP:

- Modularity: Allows dividing programs into smaller, independent parts. These parts can be reused in other programs.
- Extensibility: To extend the functionality of classes, only their code needs to be modified.
- Maintenance: Thanks to modularity, maintenance is simpler. Each class should be in a different file.

# 2.- OOP in PHP

PHP was not originally designed for OOP.

OOP features were introduced in PHP 3 and enhanced in PHP 4 and PHP 5

PHP now supports all OOP features except **multiple inheritance** and **method and operator overloading.**

# 2.- OOP in PHP

## Classes

Declaration of a class is done using the **class** keyword followed by the class name and curly braces ( { } ) enclosing property and method definitions.

```
class Producto {
        private $codigo;
        public $nombre;
        public $PVP;

        public function muestra ( ) {
                print "<p>" . $this->codigo . "</p>";
        }
}
```

# 2.- OOP in PHP

## Classes

As a good programming practice, it is recommended that the elements within the class are ordered, first for the properties and then the methods.

Class names should start with a capital letter.

Classes should be in their own file, named ClassName.inc.php.

# 2.- OOP in PHP

**Instantiating objects**

When a variable of a given class is declared, it is said that an **object of the class** is being created, or in other words **an instance of the class**.

To create an instance of an object, the word **new** is used.

```
$miProducto = new Producto();
```

Remember that before you can instantiate an object of a class, the class must be declared:

```
require_once('Producto.inc.php');
```

# 2.- OOP in PHP

**Attributes**

Attributes are like variables. It's possible to set a value to an attribute in the class declaration, so every object has the same value upon instantiation.

To access the attributes and methods of a class, use the **->** operator.

```
$miProducto->nombre = 'Samsung Galaxy Note 7';
$miProducto->muestra();
```

# 2.- OOP in PHP

**Attributes – Access Level**

Depending on the access level with which an attribute is declared, it can be accessed directly or through a method of the class.

```
class Producto {
        private $codigo;
        public $nombre;
        public $PVP;
}
```

- public: can be accessed directly.
- private: can only be accessed within the class or through a class method.
- protected: can be accessed from the class itself and its subclasses.

# 2.- OOP in PHP

## Attributes – Access Level

- Private

  Being part of the internal information of the object, there's more control over the values they store.

  It may also be useful to know the value before storing it.

# 2.- OOP in PHP

**Attributes – Access Level**

```
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;
}
```

# 2.- OOP in PHP

**setters and getters**

The methods that allow us to obtain or change the value of a private attribute are usually named starting with the words **get** and **set**. But you can also use the _ _**get and** _ _**set** magic methods.

```
void _ _set (string name, mixed value)
mixed _ _get (mixed name)
```

**name** is the name of the variable, **mixed** indicates that it can be any type.

If they are declared in a class, PHP will call them when trying to access an attribute that does not exist or is not accessible (private).

# 2.- OOP in PHP

```php
class Producto {
    private $nombre;
    private $precio;

    public function __set ($propiedad, $valor) {
        $this->$propiedad = $valor;
    }
    public function __get ($propiedad) {
        return $this->$propiedad;
    }
}
```

```php
$consola = new Producto();
$consola->nombre = "PS5";     // acceso al método mágico __set(nombre, "PS5")
$consola->precio = 499.99;    // acceso al método mágico __set(precio, 499.99)
echo $consola->nombre;        // acceso al método mágico __get(nombre)
```

# 2.- OOP in PHP

**Attributes - Access Level**

```
private $codigo;
public function _ _set ($atributo, $valor) {
        switch($atributo) {
                case ‘codigo': if (noExisteCodigo($ valor)) {
                        $this->codigo = $ valor;
                        return true;
                }
                return false;
        }
}
public function _ _get ($atributo) {
        switch($atributo) {
                case ‘codigo': return $this->codigo;
        }
}
```

# 2.- OOP in PHP

**$this object**

Every instance of an object has a reference to itself that is used when a method of that object is invoked. This reference to itself is stored in the variable **$this** which is only accessible from the methods of the object itself.

```
class Producto {
        private $codigo;

        public function cambiarCodigo ($cod) {
                $this->codigo = $cod;
        }
}
```

# 2.- OOP in PHP

**class constants**

Class constants are common to all instances of the class.
They are defined with **const**, their name is usually written in **capital letters**, it does not have the **$** symbol, its value is always enclosed in quotes and it is public.

To access these constants we will use:

- Outside the class: the name of the class and the scope resolution operator:: **(Class::CONSTANT)** or the instance of the class and the scope resolution operator **($class::CONSTANT)**

- Inside the class: the reserved word self, followed by the scope resolution operator:: **(self::CONSTANT)**

# 2.- OOP in PHP

**class constants**

```
class Coche {
        const RUEDAS = '4';
        private $modelo;
        ...
        self::RUEDAS;
        ...
}

echo Coche::RUEDAS;

miCoche = new Coche();
echo $miCoche::RUEDAS;
```

# 2.- OOP in PHP

**Static attributes and methods**

Static attributes and methods, also called class attributes and methods, as their name indicates, do not depend on an instance of the object.
They depend on the class itself, so to access them the name of the class and the operator seen previously will be used **::**

To define them, the word **static** is used.

If they are **public**, they can be accessed using the class name.

If they are **private**, they can be accessed from within the class (declaration) using the word **self**.

# 2.- OOP in PHP

**Static attributes and methods**

```
class Producto {
        private static $cantidadProductos = 0;

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

Producto::nuevoProducto();
```

# 2.- OOP in PHP

**Static attributes and methods**

Static attributes are used to store general information about the class, such as the number of instantiated objects. There is only one value for the attribute and it is stored at the class level, not the instance level.

Static methods usually perform a specific task. For example, a math class has static methods for calculating logarithms or square roots. There is no point in creating an object if all you want is to perform a calculation.

Since they are called from the class, **$this** cannot be used within these methods.

# 2.- OOP in PHP

**Constructor**

Constructors are executed when the object is created, constructor methods must be called_ _**construct**

You can see the _ _ symbols that have already been seen before, these indicate that it is a magical PHP method.

There can only be one constructor method in each class since PHP does not support method overloading.

# 2.- OOP in PHP

**Constructor**

```
class Producto {
        private static $num_productos = 0;
        private $codigo;

        public function _ _construct ( ) {
                self::$num_productos++;
        }
}

$miProducto = new Producto();
```

# 2.- OOP in PHP

**Constructor**

```
class Producto {
        private static $cantidadProductos = 0;
        private $nombre;

        public function _ _construct ($codigo) {
                    $this->$nombre = $codigo;
                self::$cantidadProductos++;
        }
}

$miProducto = new Producto('GALAXYS');
```

# 2.- OOP in PHP

**Destructor**

You can also define a destructor method_ _**destruct** included since PHP5

A destructor allows you to define the actions that will be executed when the object instance is deleted.

# 2.- OOP in PHP

**Destructor**

```
class Producto {
        private static $cantidadProductos = 0;
        private $codigo;

        public function _ _construct ($codigo) {
                $this->$codigo = $codigo;
                self::$cantidadProductos++;
        }
        public function _ _destruct ( ) {
                self::$cantidadProductos--;
        }
}
```

# 2.- OOP in PHP

## Using objects

We have already seen how to instantiate an object and how to access its public properties and methods:

```
$miProducto = new Producto();
$miProducto->nombre = 'Samsung Galaxy S';
$miProducto->muestra();
```

You can check what class an object is with the **instanceof** operator:

```
if ($miProducto instanceof Producto) {
        ...
}
```

# 2.- OOP in PHP

**Using objects**

There are other very useful functions in developing applications with OOP

get_class
class_exists
get_declared_classes
class_alias
get_class_methods
method_exists
get_class_vars
get_object_vars
property_exists

# 2.- OOP in PHP

**Copy objects**

```
$miProducto = new Producto();
$nuevoProducto = $miProducto;
```

You might assume that the above code creates a copy of the object as it does with variables, but it doesn't work the same when it comes to objects.

With the previous code you can have a new reference to the same object, it would be like creating an **alias** of the object. If any attribute of $newProduct is changed “also” it is changed in $myProduct and vice versa (it is really the same object).

With objects, the **clone** function is used if you want to create a copy of the object.

```
$nuevoProducto = clone($miProducto);
```

# 2.- OOP in PHP

**Compare objects**

Using the == and === operators you can check if two objects contain the same values or even if they are the same object.

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

// The result of $a == $p is true because are two identical copies
// The result of $a === $p is false since although they have their attributes, they have the same values, they are different objects
// What would $pCopia === $ p give?
```

# 2.- OOP in PHP

**Convert an object to string**

As in other programming languages, PHP also has a method to indicate how the object will behave when it is treated as a string of characters.

The method must be defined_ _**toString**

As it is a magical method, it can be used by invoking it directly or it will act automatically when you want to display the object as a string.

# 2.- OOP in PHP

**Convert an object to string**

```
class Producto {
        private $codigo;
        private $nombre;

        public function _ _construct ($codigo) {
                $this->codigo = $codigo;
        }

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

        public function _ _toString ( ) {
                return 'Codigo: '. $this->codigo .'<br>Nombre:'. $this->nombre;
        }
}
```

# 2.- OOP in PHP

**Convert an object to string**

```
$miProducto = new Producto('Nintendo');
$miProducto->ponerNombre('Wii');


// The following instructions will produce the same result
echo $miProducto->_ _toString();
echo $miProducto;
```

# 2.- OOP in PHP

**Inheritance**

Inheritance allows defining classes based on other existing ones. New classes are also called **subclasses**. The classes that serve as the base are called **superclasses**.

For example, imagine a vehicle buying and selling application, you could have a vehicle superclass with properties common to all vehicles and then different subclasses with specific properties that could be car, motorcycle...

# 2.- OOP in PHP

**Inheritance**

```
class Producto {
        public $codigo;
        public $nombre;
        public $nombre_corto;
        public $PVP;

        public function muestra ( ) {
                echo $this->codigo;
        }
}
```

If all the products have only those properties with the previous class it would work, but if for example the products are televisions, speakers, laptops... there are properties that are interesting to have of one type but not another.

# 2.- OOP in PHP

**Inheritance**

```
class Television extends Producto {
        public $pulgadas;
        public $tecnologia;
}

class Altavoz extends Producto {
        public $potencia;
        public $canales;
}
```

The word **extends** indicates that the new class is based on the indicated class but will also have the attributes indicated in the **subclass**.

# 2.- OOP in PHP

**Inheritance**

On the same manual page seen previously you can consult very useful functions for working with inheritance: functions for classes and objects

get_parent_class

is_subclass_of

# 2.- OOP in PHP

**Inheritance**

Keep in mind that from a subclass you cannot access a property or method that is **private** in the superclass, to do so you must define it as **protected** in the superclass.

Although there is no method overloading, overriding a method in the subclass is allowed.

```
class Television extends Producto {
        public $pulgadas;
        public $tecnologia;
        public function muestra ( ) {
                print $this->pulgadas . ' pulgadas';
        }
}
```

# 2.- OOP in PHP

**Clases y métodos finales**

Sometimes it is not interesting that subclasses can redefine the behavior of methods. It may also be interesting not to create subclasses.

For this the word **final** is used.

```
final class Producto ( ) {
        ...
}

public final function ejemplo ( ) {
        ...
}
```

# 2.- OOP in PHP

**Abstract classes and methods**

In contrast to the final statement, **abstract** can be used, which indicates that this class cannot have instantiated objects, but can be used as the basis for a subclass.

```
abstract class Producto { ... }
```

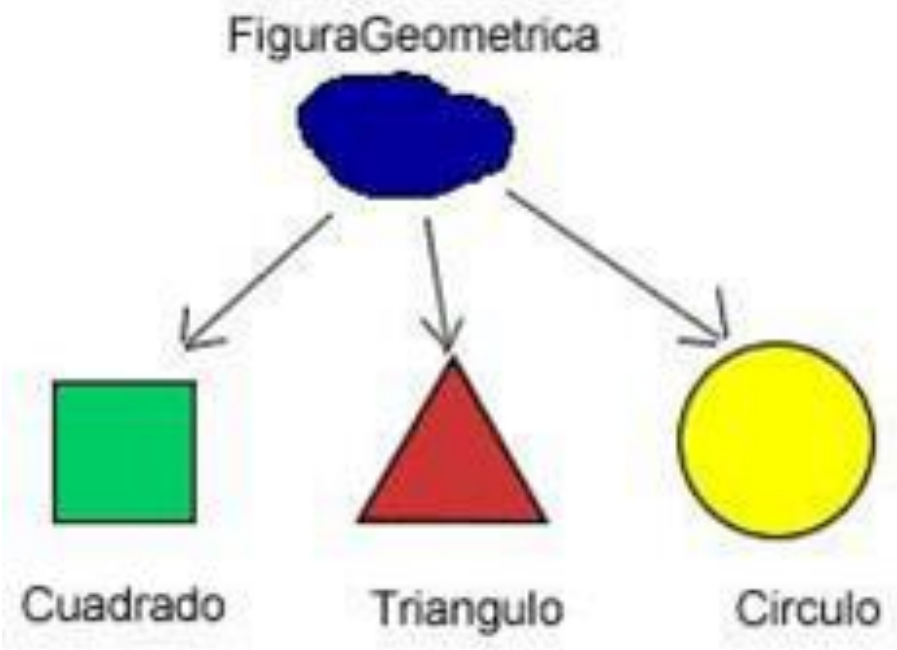
If a method is defined **abstract** in a superclass, that module cannot have code in the superclass and subclasses are required to define said method

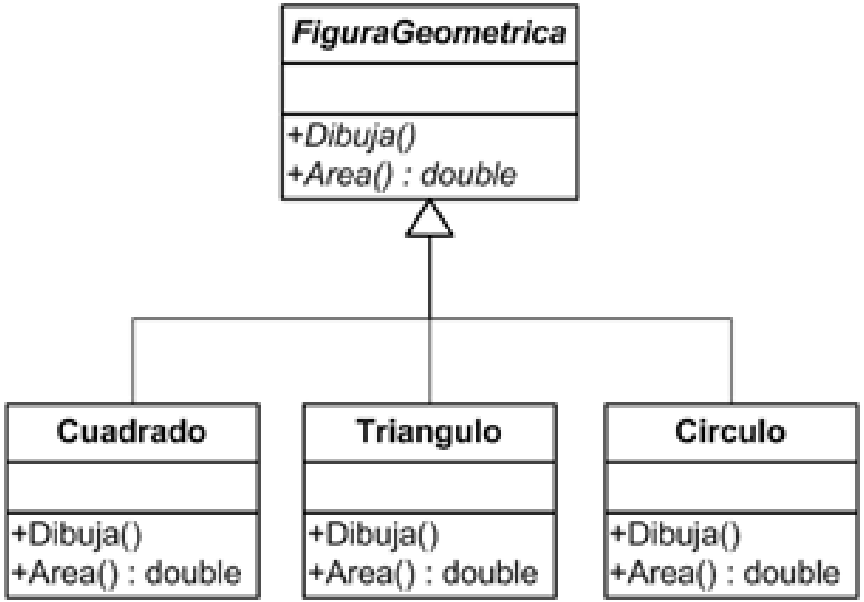
```
abstract public function prueba ( );
```

You cannot declare a class as **abstract** and **final** at the same time..

# 2.- OOP in PHP

**Example of abstract classes**



# 2.- OOP in PHP

```
abstract class Figura {
        protected $color;

        public function _ _set($name, $value) {
                if ($name == 'Color' && is_string($value) === true)
                        $this->color = $value;
                }

        abstract public function Dibuja ( );
        abstract public function Area ( );
}

class Cuadrado extends Figura {
        public function Dibuja ( ) {
                echo 'Dib Cuadrado '. $this->color;
        }

        public function Area ( ) {
                        return 0;
        }
}
```

```
class Circulo extends Figura {
        public function Dibuja ( ) {
                echo 'Dib Circulo '.$this->color;
        }
        public function Area ( ) {
                return 0;
        }
}

class Triangulo extends Figura {
         public function Dibuja ( ) {
                echo 'Dib Triang '.$this->color;
         }
         public function Area() {
                return 0;
         }
}

// Uso
$cuadrado = new Cuadrado();
$cuadrado->Color = 'Negro'; // Se usa el _ _ set de Figura
$cuadrado->Dibuja();
```

# 2.- OOP in PHP

**Calling superclass methods**

Sometimes, even if a method is overwritten in a subclass, the method of its superclass can also be executed from it, for this the scope resolution operator is used ::
An example of a call would be the following:

```
parent::metodoElegido();
```

# 2.- OOP in PHP

```
class Producto {

        public $codigo;
        public $nombre;
        public $modelo;
        public $PVP;

        public function muestra ( ) {
                print $this->codigo;
        }

        public function _ _construct ($row) {
                $this->codigo = $row['cod'];
                $this->nombre = $row['nombre'];
                $this->modelo = $row[modelo];
                $this->PVP = $row['PVP'];
        }
}
```

```
class TV extends Producto {
public $pulgadas;
        public $tecnologia;

        public function muestra ( ) {
                print $this->pulgadas . ' pulgadas ';
        }

        public function _ _construct($row) {
                parent::__construct($row);
                $this->pulgadas = $row['pulgadas'];
                $this->tecnologia = $row['tecnologia'];
                }
        }
```

# 2.- OOP in PHP

**Interfaces**

An **interface** can be understood as an empty class that only contains declarations to methods that will be empty without including the implemented source code. They are defined with the word interface. It is used as a template to create other classes, so that these classes must have defined all the code for the methods indicated in the interface. For a class to follow that template, the word **implements** is used.

```
interface mostrarDatos {
        public function mostrar();
}

class Television extends Producto implements mostrarDatos {
        public function mostrar ( ) {
        ...
        }
}
```

# 2.- OOP in PHP

**Interfaces VS Abstract Classes**

When using OOP one of the most common doubts is which solution to choose, or abstract classes or interfaces. Both solutions allow defining rules for classes to inherit or implement, but neither allows instantiating objects.

- Abstract classes: their methods can contain code, if there is going to be common code in several subclasses, the code is implemented in the abstract class. Interfaces: the methods provided are empty, if there is common code it will have to be implemented in all classes that implement the interface.

- Attributes: abstract classes can have them, interfaces cannot.

- Inheritance: you cannot have multiple inheritance but you can create a class that implements several interfaces.

# 2.- OOP in PHP

**Traits**

One of the typical characteristics of inheritance is being able to reuse code already coded in the superclass.

Multiple inheritance is not allowed in PHP, but traits are available, which allow code to be reused, reducing the limitations of simple inheritance.

Traits are similar to classes, but they are not allowed to extend classes, instantiate them, or implement interfaces and only allow functionality to be grouped.

# 2.- OOP in PHP

```
trait Hello {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait World {
    public function sayWorld() {
        echo 'World';
    }
}
```

```
class MyHelloWorld {
    use Hello, World;
    public function sayExclamationMark() {
        echo '!';
    }
}

$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();
$o->sayExclamationMark();
```

The result of executing the above would be: Hello World!

# 2.- OOP in PHP

```
class Base {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait SayWorld {
    public function sayHello() {
        parent::sayHello();
        echo 'World!';
    }
}
```

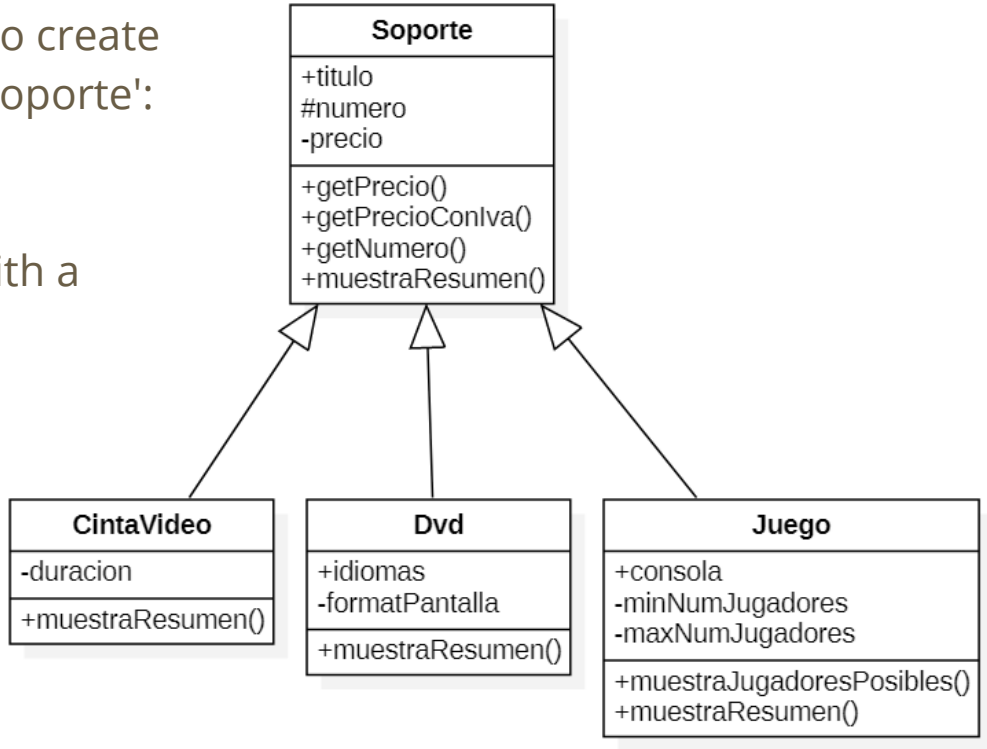
```
class MyHelloWorld extends Base {
    use SayWorld;
}

$o = new MyHelloWorld();
$o->sayHello();
```

The result of executing the above would be: Hello World!

# Class inheritance exercise

From the diagram in the image, you have to create the necessary classes. You will start with ‘Soporte':

- Create the constructor
- Create setters and getters
- Define a private constant called 'VAT' with a value of 21%



# Class inheritance exercise

Test the class with the following code:

```
include "Soporte.php";

$soporte1 = new Soporte("Tenet", 22, 3);
echo "<strong>" . $soporte1->titulo . "</strong>";
echo "<br>Precio: " . $soporte1->getPrecio() . " euros";
echo "<br>Precio IVA incluido: " . $soporte1->getPrecioConIVA() . " euros";
$soporte1->muestraResumen();
```

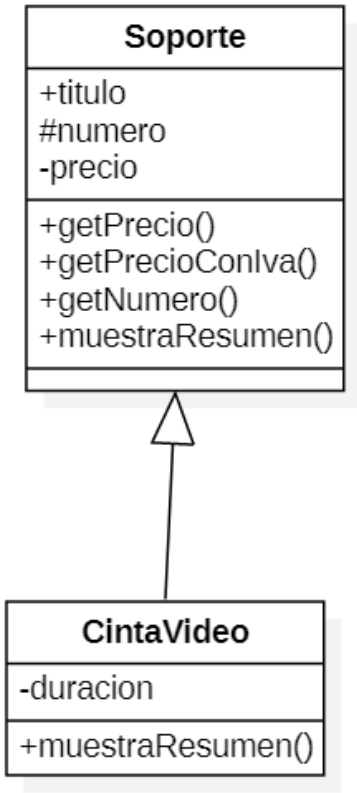
# Class inheritance exercise

The result should be:

```
Tenet
Precio: 3 euros
Precio IVA incluido: 3.63 euros
Tenet
3 € (IVA no incluido)
```

# Class inheritance exercise

Create the CintaVideo class which inherits from Soporte. Add the duration attribute and override both the constructor and the showSummary method. Try it with the following code:




```
include "CintaVideo.php";

$miCinta = new CintaVideo("Los cazafantasmas", 23, 3.5, 107);
echo "<strong>" . $miCinta->titulo . "</strong>";
echo "<br>Precio: " . $miCinta->getPrecio() . " euros";
echo "<br>Precio IVA incluido: " . $miCinta->getPrecioConIva() . " euros";
$miCinta->muestraResumen();
```

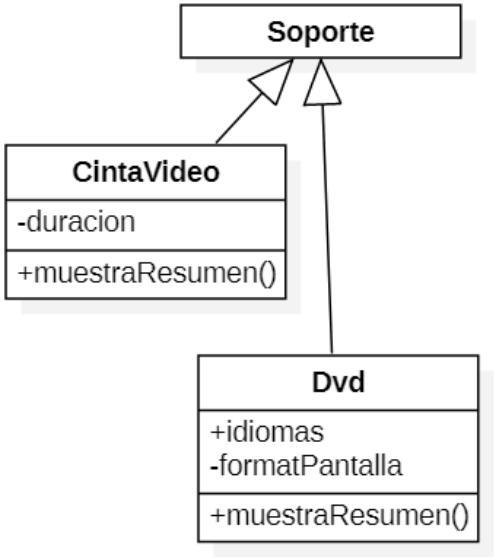
# Class inheritance exercise

The result should be:

```
Los cazafantasmas
Precio: 3.5 euros
Precio IVA incluido: 4.24 euros
Película en VHS:
Los cazafantasmas
3.5 € (IVA no incluido)
Duración: 107 minutos
```

# Class inheritance exercise

Create the Dvd class which inherits from Soporte. Add the languages and screenFormat attributes. Then override both the constructor and the showSummary method.




```
include "Dvd.php";

$miDvd = new Dvd("Origen", 24, 15, "es,en,fr", "16:9");
echo "<strong>" . $miDvd->titulo . "</strong>";
echo "<br>Precio: " . $miDvd->getPrecio() . " euros";
echo "<br>Precio IVA incluido: " . $miDvd->getPrecioConIva() . " euros";
$miDvd->muestraResumen();
```

# Class inheritance exercise

The result should be:

```
Origen
Precio: 15 euros
Precio IVA incluido: 18.15 euros
Película en DVD:
Origen
15 € (IVA no incluido)
Idiomas:es,en,fr
Formato Pantalla:16:9
```

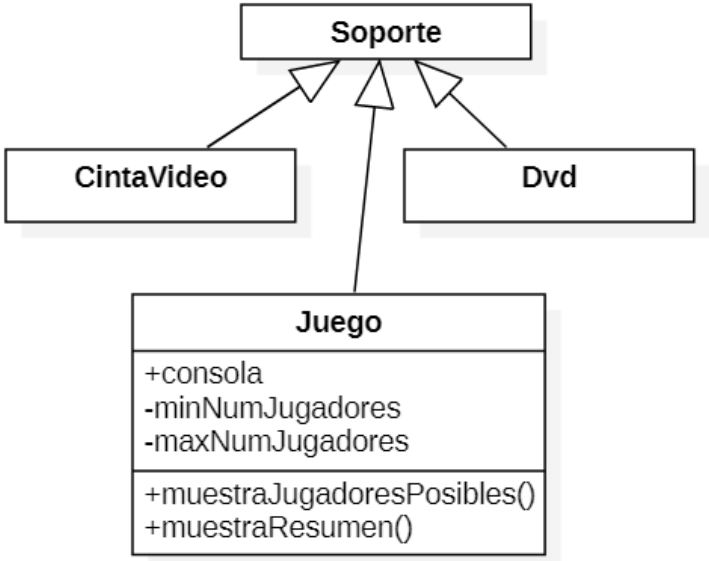
# Class inheritance exercise

Create the Game class which inherits from Soporte.

Add the console, minNumPlayers and maxNumPlayers attributes.

Add the method showPossiblePlayers , which should show 'For one player', 'For X players' or 'From X to Y players' depending on the values of the attributes created.

Finally, override both the constructor and the showSummary method.



# Class inheritance exercise

Test the class with the following code:

```
include "Juego.php";

$miJuego = new Juego("The Last of Us Part II", 26, 49.99, "PS4", 1, 1);
echo "<strong>" . $miJuego->titulo . "</strong>";
echo "<br>Precio: " . $miJuego->getPrecio() . " euros";
echo "<br>Precio IVA incluido: " . $miJuego->getPrecioConIva() . " euros";
$miJuego->muestraResumen();
```

# Class inheritance exercise

The result should be:

```
The Last of Us Part II
Precio: 49.99 euros
Precio IVA incluido: 60.49 euros
Juego para: PS4
The Last of Us Part II
49.99 € (IVA no incluido)
Para un jugador
```