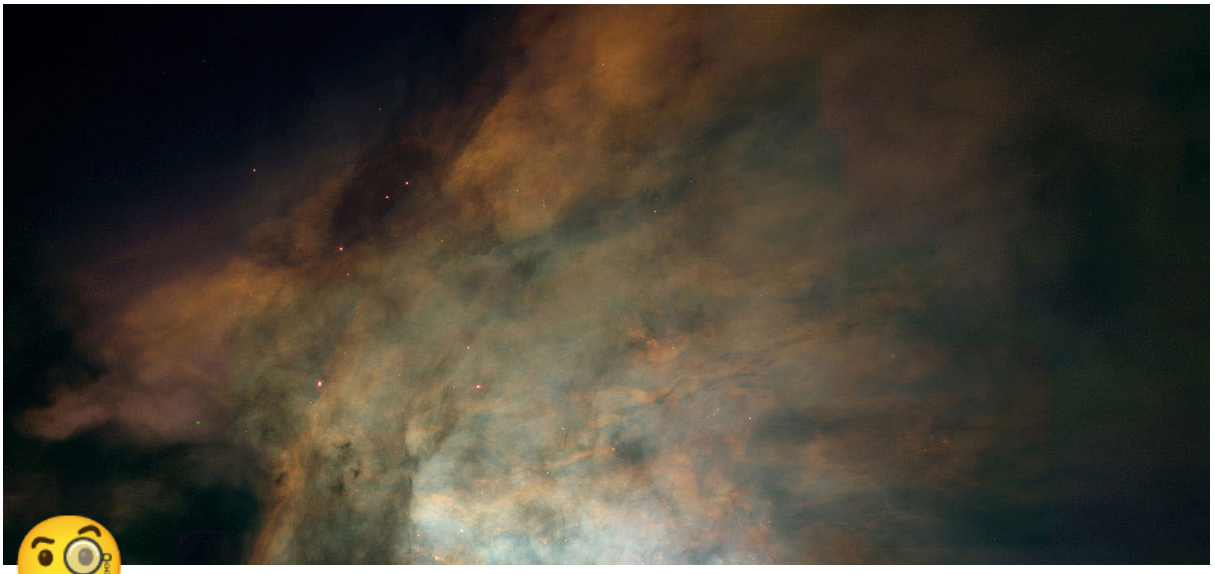

OOP Documentation

Introduction à la POO en PHP

L'orientation objet (POO) est un paradigme de programmation qui consiste à créer des objets, qui sont des entités autonomes regroupant des données (propriétés) et les actions qui peuvent être effectuées sur ces données (méthodes).

Avantages de l'utilisation de l'OOP en PHP

- **Meilleure organisation du code:** L'OOP vous aide à organiser votre code en unités logiques, ce qui le rend plus facile à comprendre et à maintenir.
- **Augmentation de la réutilisation du code:** Vous pouvez réutiliser du code en créant des classes et en les héritant d'autres classes.
- **Réduction de la redondance:** L'OOP vous aide à éviter d'écrire le même code plusieurs fois.
- **Amélioration de la maintenabilité:** Le code OOP est généralement plus facile à modifier et à étendre.
- **Amélioration de la modularité:** L'OOP vous permet de décomposer votre application en modules plus petits et indépendants.

En PHP, l'OOP introduit un moyen puissant d'organiser et de structurer votre code, le rendant plus maintenable, flexible et facile à comprendre. Voici quelques concepts

clés de l'OOP en PHP :

Classes et objets

Les classes et les objets sont les deux concepts fondamentaux de l'OOP.

Classes

Une classe est un modèle qui définit la structure d'un objet. Elle comprend les propriétés, qui sont les données associées à l'objet, et les méthodes, qui sont les actions que l'objet peut effectuer.

Objets

Un objet est une instance d'une classe. Il s'agit d'une entité concrète avec ses propres valeurs pour les propriétés définies dans la classe.

Par exemple :

```
// Example of a simple class in PHP
class Car {
    // Properties
    public $brand;
    public $model;

    // Methods
    public function startEngine() {
        return "Engine started for $this->brand $this->model.
    }
}

// Creating an object (instance) of the Car class
$myCar = new Car();
$myCar->brand = "Toyota";
$myCar->model = "Corolla";
echo $myCar->startEngine(); // Output: "Engine started for To
```

Encapsulation :

L’encapsulation est le concept de regroupement des données (propriétés) et des méthodes (fonctions) qui fonctionnent sur les données dans une seule unité (classe).

Il permet de restreindre l’accès à certains composants et de masquer l’état interne des objets.

```
class BankAccount {
    private $balance = 0; // Encapsulated property

    public function deposit($amount) {
        $this->balance += $amount;
    }

    public function getBalance() {
        return $this->balance;
    }
}

$account = new BankAccount();
$account->deposit(100);
echo $account->getBalance(); // Output: 100
// echo $account->balance; // Error: Cannot access private pr
```

PHP Access Modifiers :

- `public` : L'accès est autorisé depuis l'extérieur de la classe.
- `private` : L'accès est autorisé uniquement depuis l'intérieur de la classe.
- `protected` : L'accès est autorisé depuis l'intérieur de la classe et depuis les classes héritées.

Héritage :

L’héritage permet à une classe (enfant/sous-classe) d’hériter des propriétés et des méthodes d’une autre classe (parent/superclasse). Il favorise la réutilisabilité du code et la création d’une hiérarchie de classes.

```
class Animal {
    public function makeSound() {
        return "Some sound";
    }
}
```

```
class Dog extends Animal {
    public function eat() {
        return "Woof!";
    }
}

$animal = new Animal();
echo $animal->makeSound(); // Output: "Some sound"

$dog = new Dog();
echo $dog->makeSound(); // Output: "Woof!"
```

▼ Differente types in Inheritance :

Inheritance in object-oriented programming allows a class (known as a subclass or derived class) to inherit properties and behaviors from another class (known as a superclass or base class). There are several types or variations of inheritance, each defining how properties and behaviors are inherited from the parent class. The main types of inheritance include:

Single Inheritance

Single inheritance involves one class inheriting properties and behaviors from a single parent class. Each subclass has only one immediate superclass.

Example:

```
class Animal {
    // Animal properties and methods
}

class Dog extends Animal {
    // Dog inherits from Animal
}
```

Multilevel Inheritance

Multilevel inheritance involves a chain of inheritance where a class inherits properties and behaviors from a superclass, and then another class further inherits from that subclass.

Example:

```
class Animal {
    // Animal properties and methods
}

class Mammal extends Animal {
    // Mammal inherits from Animal
}

class Dog extends Mammal {
    // Dog inherits from Mammal
}
```

Hierarchical Inheritance

Hierarchical inheritance involves multiple subclasses inheriting from the same superclass. It creates a hierarchical structure where one superclass has multiple subclasses.

Example:

```
class Animal {
    // Animal properties and methods
}

class Cat extends Animal {
    // Cat inherits from Animal
}

class Dog extends Animal {
    // Dog inherits from Animal
}
```

Multiple Inheritance (Not supported in all programming languages)

Multiple inheritance occurs when a class inherits properties and behaviors from more than one superclass. However, some programming languages do not

support multiple inheritance due to complexities related to ambiguity and the diamond problem.

Example (conceptual, not supported in all languages):

```
class A {
    // Class A properties and methods
}

class B {
    // Class B properties and methods
}

class C extends A, B {
    // Multiple inheritance (conceptual, not supported in
}
```

Hybrid/Combined Inheritance

Hybrid or combined inheritance is a combination of multiple inheritance types. It combines various types of inheritance mechanisms.

▼ **Superclass & Subclass :**

In object-oriented programming, specifically in the context of inheritance, the terms "Derived Class" and "Super Class" refer to different classes within an inheritance hierarchy.

- **Superclass (Base Class or Parent Class):** This is the class that is extended or inherited from. It contains properties and methods that are shared by one or more subclasses.
- **Subclass (Derived Class or Child Class):** This is the class that inherits from another class. It extends the functionality of the superclass by adding new properties or methods or by overriding existing methods.

Here's a breakdown of these terms:

- **Superclass (or Base Class or Parent Class):** This is the original class that serves as the foundation. It provides a blueprint for properties and behaviors that can be inherited by other classes.

Example:

```
// Superclass
class Animal {
    // Properties and methods common to all animals
}
```

- **Subclass (or Derived Class or Child Class):** This is a class that inherits properties and methods from its superclass. It can extend the functionality of the superclass by adding new methods or properties and can override methods from the superclass.

Example:

```
// Subclass (derived from Animal)
class Dog extends Animal {
    // Additional properties and methods specific to dogs
}
```

So, to clarify:

- The **superclass** is the base class that provides the common functionality.
- The **subclass** is the derived class that extends or specializes the functionality of the superclass.

The terms "superclass" and "base class" are often used interchangeably, similarly, "subclass" and "derived class" are used interchangeably to refer to the classes in an inheritance relationship.

Polymorphisme :

Le polymorphisme permet de traiter des objets de différentes classes comme des objets d’une superclasse commune. Il permet l’utilisation d’une interface unique pour différents types de données.

```
interface Shape {
    public function calculateArea();
}

class Circle implements Shape {
```

```
    private $radius;

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

    public function calculateArea() {
        return pi() * $this->radius * $this->radius;
    }
}

class Square implements Shape {
    private $side;

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

    public function calculateArea() {
        return $this->side * $this->side;
    }
}

function getArea(Shape $shape) {
    return $shape->calculateArea();
}

$circle = new Circle(5);
echo getArea($circle); // Output: Area of the circle

$square = new Square(4);
echo getArea($square); // Output: Area of the square
```

Il existe deux principaux types de polymorphisme :

▼ **Compile-time polymorphism (also known as static polymorphism or method overloading)**:

Method overriding :permet à une classe d'avoir plusieurs méthodes avec le même nom mais des paramètres différents. La méthode correcte à exécuter est déterminée par le nombre ou le type de paramètres au moment de la compilation.

Exemple (Java) :

```
class Animal {
    public void emettreSon() {
        System.out.println("Un son");
    }
}

class Chien extends Animal {
    public void emettreSon() {
        System.out.println("Aboiement");
    }
}

class Chat extends Animal {
    public void emettreSon() {
        System.out.println("Miaulement");
    }
}
```

▼ IN PHP

PHP does not support method overloading directly as some other languages do (like Java). Method overloading is the ability to define multiple methods with the same name but with different parameter lists within the same class.

However, in PHP, you can achieve something similar using variable-length argument lists and default parameter values.

Example:

```
class Example {
    public function add(...$args) {
        $sum = 0;
        foreach ($args as $num) {
            $sum += $num;
```

```
        }
        return $sum;
    }
}

$obj = new Example();
echo $obj->add(2, 3); // Output: 5
echo $obj->add(2, 3, 4); // Output: 9
```

▼ **Run-time polymorphism (also known as dynamic polymorphism or method overriding)**:

Method overriding :est réalisée en créant une méthode dans la sous-classe avec la même signature (nom de méthode et paramètres) qu'une méthode dans la superclasse.

```
class Animal {
    public function makeSound() {
        echo "Some sound";
    }
}

class Dog extends Animal {
    public function makeSound() {
        echo "Bark";
    }
}

class Cat extends Animal {
    public function makeSound() {
        echo "Meow";
    }
}

$dog = new Dog();
$cat = new Cat();
```

```
$dog->makeSound(); // Output: "Bark"
$cat->makeSound(); // Output: "Meow"
```

Traits :

Les traits en programmation orientée objet (POO) en PHP sont un mécanisme permettant la réutilisation de code, offrant une forme de composition horizontale. Ils fournissent une manière de regrouper des fonctionnalités de manière réutilisable dans les classes, sans utiliser l'héritage multiple.

```
trait Log {
    public function logInfo($message) {
        echo 'Log: ' . $message;
    }
}

class User {
    use Log;

    public function displayMessage() {
        $this->logInfo('User information displayed.');
        // ... Autres fonctionnalités spécifiques à la classe
    }
}

class Admin {
    use Log;

    public function displayMessage() {
        $this->logInfo('Admin information displayed.');
        // ... Autres fonctionnalités spécifiques à la classe
    }
}
```

Concepts avancés de POO :

▼ Interfaces:

Une interface dans OOP définit un contrat pour les classes qui l'implémentent. Il spécifie un ensemble de méthodes qui doivent être implémentées par toute classe qui adhère à cette interface.
Les interfaces garantissent un comportement cohérent entre les différentes classes et favorisent la réutilisabilité du code en fournissant un plan directeur que les classes peuvent suivre.

```
interface CookieDecorator {
    public function decorate();
}

class ChocolateChipCookie implements CookieDecorator {
    public function decorate() {
        // Implementation specific to Chocolate Chip Cooki
    }
}

class SugarCookie implements CookieDecorator {
    public function decorate() {
        // Implementation specific to Sugar Cookie
    }
}
```

▼ Abstract Classes

Les classes abstraites en programmation orientée objet (OOP) sont des classes qui ne peuvent pas être instanciées seules.
Ils servent de plans ou de modèles à d'autres classes.
Les classes abstraites peuvent contenir à la fois des méthodes implémentées (méthodes avec un corps) et des méthodes abstraites (méthodes sans corps).

```
abstract class Shape {
    protected $name;

    public function __construct($name) {
        $this->name = $name;
```

```
    }

    abstract public function calculateArea(); // Abstract m

    public function getName() {
        return $this->name;
    }
}

class Square extends Shape {
    private $side;

    public function __construct($name, $side) {
        parent::__construct($name);
        $this->side = $side;
    }

    public function calculateArea() {
        return $this->side * $this->side;
    }
}

class Circle extends Shape {
    private $radius;

    public function __construct($name, $radius) {
        parent::__construct($name);
        $this->radius = $radius;
    }

    public function calculateArea() {
        return pi() * $this->radius * $this->radius;
    }
}

$square = new Square("Square", 4);
echo $square->getName() . " Area: " . $square->calculateAr
```

```
$circle = new Circle("Circle", 3);
echo $circle->getName() . " Area: " . $circle->calculateAr
```

▼ Static Methods and Properties

Les méthodes et propriétés statiques appartiennent à la classe elle-même plutôt qu'à toute instance spécifique de la classe. Ils sont accessibles en utilisant le nom de la classe directement, sans avoir besoin d'une instance d'objet.
Les méthodes statiques peuvent effectuer des actions qui ne sont liées à aucune instance spécifique et peuvent être utiles pour les fonctions utilitaires ou les instances de comptage.

```
class Cookie {
    public static $totalBaked = 0;

    public static function bakeCookie() {
        // Baking process
        self::$totalBaked++; // Increment total baked cook
    }
}

Cookie::bakeCookie(); // Invoking the static method
echo Cookie::$totalBaked; // Accessing the static property
```

Constructor and Destructo:

Les constructeurs et les destructeurs sont des types spéciaux de méthodes utilisées dans les classes pour initialiser les objets et effectuer des opérations de nettoyage, respectivement.

Constructor :

A destructor is a special method that is automatically called when an object is destroyed or goes out of scope. It's used for performing cleanup tasks, releasing resources, or doing any necessary finalization before an object is removed from memory.

En PHP, la méthode constructeur est nommée `__destruct()` :

```
class MyClass {
    private $name;

    public function __construct($name) {
        $this->name = $name;
        echo "Initializing object with name: $this->name <br>
    }
}
// Creating objects
$obj1 = new MyClass("Object 1");

// Output:
// Initializing object with name: Object 1

// Object destruction happens when they go out of scope
unset($obj1);

// Output:
// Destroying the object
```

Destructo :

Un constructeur est une méthode spéciale qui est appelée automatiquement lorsqu'un objet est créé. Il est utilisé pour initialiser les propriétés de l'objet ou effectuer toute configuration nécessaire avant que l'objet ne soit prêt à être utilisé.

En PHP, la méthode constructeur est nommée `__construct()` :

```
class MyClass {
    public function __destruct() {
        echo "Destroying the object <br>";
    }
}
// Creating objects
$obj2 = new MyClass("Object 2");

// Output:
```

```
// Initializing object with name: Object 2

// Object destruction happens when they go out of scope
unset($obj2);

// Output:
// Destroying the object
```

MAGIC METHODS

Les méthodes magiques de PHP telles que `__toString()`, `__call()`, `__callStatic()`, `__invoke()` et `__clone()` sont des fonctionnalités intégrées à PHP pour permettre un comportement spécifique dans des situations particulières lors de la création et de la manipulation des objets.

`__toString()`

- **Description :** Convertit un objet en chaîne lorsqu'il est utilisé dans un contexte de chaîne, comme avec `echo` ou `print`.
- **Utilisation :** Permet de définir la représentation de l'objet sous forme de chaîne.
- **Exemple :**

```
class MaClasse {
    public function __toString() {
        return "Ceci est un objet de la classe MaClasse.";
    }
}

$obj = new MaClasse();
echo $obj; // Affiche : Ceci est un objet de la classe MaC
```

`__call()`

- **Description :** Déclenchée lors de l'appel de méthodes inaccessibles dans un contexte d'objet.
- **Utilisation :** Permet de gérer dynamiquement les appels de méthodes qui ne sont pas accessibles ou définies dans la classe.

- **Exemple :**

```
class MaClasse {
    public function __call($nom, $arguments) {
        echo "La méthode $nom n'existe pas.";
    }
}

$obj = new MaClasse();
$obj->methodeInexistante(); // Affiche : La méthode method
```

`__callStatic()`

- **Description :** Déclenchée lors de l'appel de méthodes statiques inaccessibles dans un contexte de classe.
- **Utilisation :** Permet de gérer dynamiquement les méthodes statiques inaccessibles ou non définies dans la classe.
- **Exemple :**

```
class MaClasse {
    public static function __callStatic($nom, $arguments)
        echo "La méthode statique $nom n'existe pas.";
    }
}

MaClasse::methodeStatiqueInexistante(); // Affiche : La mé
```

`__invoke()`

- **Description :** Appelée lorsqu'un objet est utilisé comme une fonction.
- **Utilisation :** Permet de rendre les objets appelables comme des fonctions.
- **Exemple :**

```
class MaClasse {
    public function __invoke($arg) {
        echo "Appelé avec l'argument : $arg";
    }
```

```
}

$obj = new MaClasse();
$obj("Bonjour"); // Affiche : Appelé avec l'argument : Bon
```

`__clone()`

- **Description :** Déclenchée lorsqu'un objet est cloné en utilisant le mot-clé `clone`.
- **Utilisation :** Permet de personnaliser le comportement lors du clonage d'objets, notamment pour la gestion des propriétés.
- **Exemple :**

```
class MaClasse {
    public function __clone() {
        echo "Objet cloné.";
    }
}

$obj1 = new MaClasse();
$obj2 = clone $obj1; // Affiche : Objet cloné.
```

`__get()`

- **Description :** Déclenchée lorsqu'on essaie d'accéder à des propriétés inaccessibles ou inexistantes d'un objet.
- **Utilisation :** Elle permet d'intercepter les tentatives d'accès à des propriétés non définies ou inaccessibles et de définir un comportement personnalisé pour gérer ces cas.
- **Exemple :**

```
class MaClasse {
    private $donnees = [];

    public function __get($nom) {
        return $this->donnees[$nom] ?? null;
    }
}
```

```
$obj = new MaClasse();
$obj->proprieteInexistante; // Appelle la méthode __get()
```

`__set()`

- **Description :** Déclenchée lorsqu'on essaie d'assigner une valeur à une propriété inaccessible ou inexistante d'un objet.
- **Utilisation :** Elle permet d'intercepter les tentatives d'assignation de valeurs à des propriétés non définies ou inaccessibles et de définir un comportement personnalisé pour gérer ces assignations.
- **Exemple :**

```
class MaClasse {
    private $donnees = [];

    public function __set($nom, $valeur) {
        $this->donnees[$nom] = $valeur;
    }
}

$obj = new MaClasse();
$obj->proprieteInexistante = "Valeur"; // Appelle la métho
```

`__isset()`

- **Description :** Appelée lorsqu'on utilise `isset()` ou `empty()` sur des propriétés inaccessibles ou inexistantes d'un objet.
- **Utilisation :** Elle permet de définir un comportement personnalisé lors de la vérification de l'existence de propriétés non définies ou inaccessibles.
- **Exemple :**

```
class MaClasse {
    private $donnees = [];

    public function __isset($nom) {
        return isset($this->donnees[$nom]);
```

```
    }
}

$obj = new MaClasse();
isset($obj->proprieteInexistante); // Appelle la méthode _
```

`__unset()`

- **Description :** Déclenchée lorsqu'on utilise `unset()` sur des propriétés inaccessibles ou inexistantes d'un objet.
- **Utilisation :** Elle permet de définir un comportement personnalisé pour supprimer des propriétés non définies ou inaccessibles.
- **Exemple :**

```
class MaClasse {
    private $donnees = [];

    public function __unset($nom) {
        unset($this->donnees[$nom]);
    }
}

$obj = new MaClasse();
unset($obj->proprieteInexistante); // Appelle la méthode _
```

Namespaces :

Namespaces are used to avoid naming conflicts between classes, functions, and constants. They provide a way to organize code by grouping related classes, interfaces, functions, and constants under a specific namespace.

Defining a Namespace

```
// Define a namespace at the beginning of the PHP file
namespace MyNamespace;

class MyClass {
```

```
    // Class definition
}

function myFunction() {
    // Function definition
}
```

Using Namespaces

```
// Using classes/functions from a namespace
use MyNamespace\\MyClass;
use MyNamespace\\myFunction;

$obj = new MyClass();
myFunction();
```

Namespaces help prevent naming collisions by allowing multiple classes with the same name to exist under different namespaces. They improve code readability and maintainability by organizing code into logical groups.

Autoloading in OOP (Object-Oriented Programming):

Autoloading is a mechanism that automatically includes (or "loads") PHP files containing class definitions when they are needed, without the need for manual inclusion using `require` or `include` statements for each class file.

PSR-4 Autoloading

The PSR-4 autoloading standard defines a way to map namespaces to directory structures. It's commonly used in modern PHP applications.

Example directory structure:

```
- project
  - src
    - MyNamespace
      - MyClass.php
  - vendor
  - autoload.php
```

Autoloading Example (using Composer)

1. First, define the PSR-4 autoload mapping in the `composer.json` file:

```
{
    "autoload": {
        "psr-4": {
            "MyNamespace\\\\": "src/"
        }
    }
}
```

1. Run `composer install` or `composer dump-autoload` to generate the autoloading configuration.
2. Use the autoloaded classes in your code:

```
// In your PHP file
require_once 'vendor/autoload.php';

use MyNamespace\\MyClass;

$obj = new MyClass();
```

Composer's autoloader loads classes based on the PSR-4 autoloading standard, mapping namespaces to directories. When a class is instantiated, Composer's autoloader automatically includes the respective class file based on the namespace and class name.

Combining namespaces and autoloading with Composer greatly simplifies managing and loading classes in larger PHP projects, making code organization and maintenance more efficient.

OOP question :

▼ **How is abstraction different from encapsulation?**

Abstraction is about hiding the internal details, while encapsulation is about bundling data and methods that operate on the data.

▼ Abstraction:

Allows to hide unnecessary data from the user. This reduces program complexity efforts.

it displays only the necessary information to the user and hides all the internal background details.

it deals with the outside view of an object (Interface).