Programmation Orientée Objet en PHP

-Niveau débutant & intermédiaire-

Introduction (1/3)

1. Objectif

- Découvrir un peu plus PHP
- Maîtriser les bases de la POO
- Vous permettre de répondre aux besoin de votre entreprise
- Gagner en confiance et motivation

2. Requis

- Version PHP >=7.4 (avec laragon, lamp, wamp, ...)
- Avoir les base de PHP (variable, if, for, while, ...)
- Éditeur de texte (vs-code, phpstorm, sublime-text, ...)
- Shell terminal / Powershell avec le client php-cli

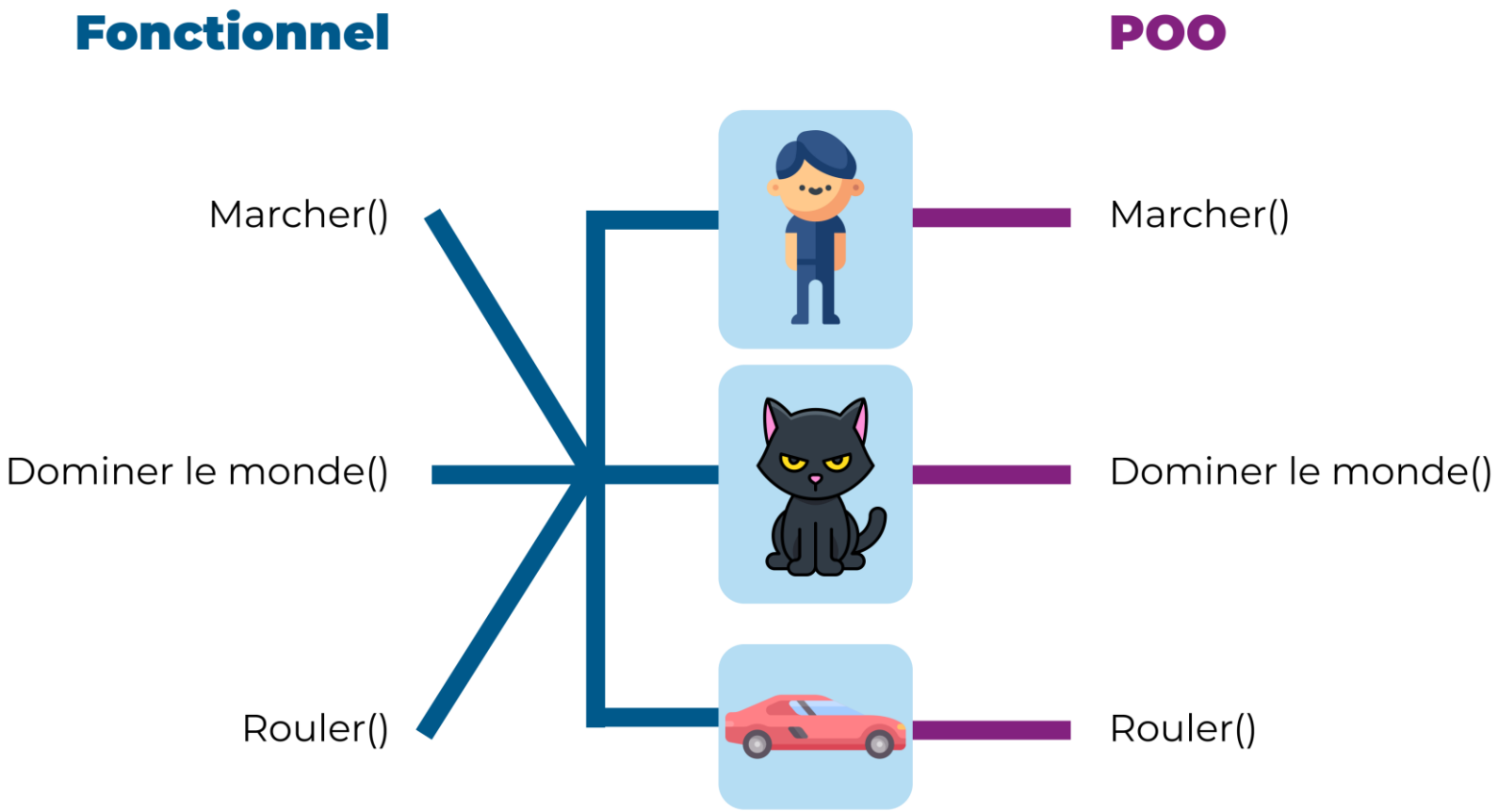
Introduction (2/3)

La programmation orientée objet (POO) est devenue **l'un des éléments constitutifs de la programmation**, qui remplace ou améliore la programmation procédurale. Alors que l'idée plus traditionnelle de la programmation procédurale place l'action et la logique au centre, **la POO utilise des objets et des données pour produire des résultats**.

Au niveau le plus simple, la POO se concentre sur les objets qu'un programmeur souhaite modifier plutôt que sur les actions nécessaires pour effectuer ce changement. Cela **facilite l'exécution d'analyses de code** par les programmeurs et ces objets sont **réutilisables dans d'autres projets**.

La majorité des langages de programmation modernes tels que C++, Object Pascal, Java, Python et PHP combinent à la fois la programmation orientée objet et la programmation procédurale, ce qui signifie que la programmation orientée objet est devenue une évolution très importante dans le monde de la programmation.

Introduction (3/3)



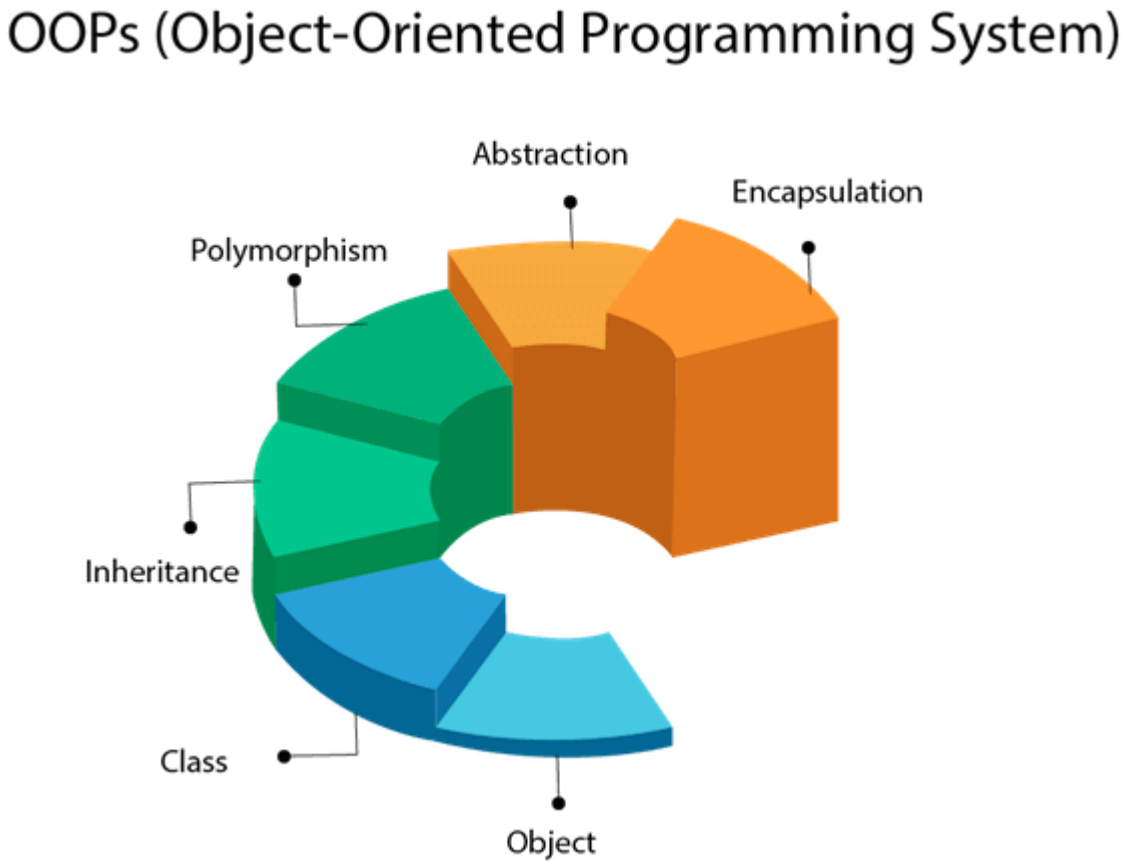
Au programme

Organisation du code
- Les classes
- Propriétés (attributs) d'une classe
- Méthodes d'une classe

L'encapsulation, Sécurité du code (intégrité)
- Visibilité Private
- Accesseurs & Mutateurs

Réutilisation du code et gain de temps
- Héritage
- Mot clé final
- Trait

Plus de contrôle sur votre développement
- Abstraction & interfaces
- Polymorphisme
- Namespace
- Exceptions
- Autoloader



Analyse de cas

La société R souhaite un programme afin d'organiser son personnel, elle fait appel à un développeur qui écrit le code suivant

```php
$nom1 = 'Hamada';
$prenom1 = 'Fahari';
$age1 = 35;
$anciennete1 = 12;

$nom2 = 'Dupont';
$prenom2 = 'Michel';
$age2 = 50;
$anciennete2 = 20;

function presentation($nom, $prenom, $age, $anciennete
{
  echo "Mon nom est $nom, mon prenom est $prenom,
    j'ai $age ans
    je travail depuis $anciennete ans.\n\r";
}

presentation($nom1, $prenom1, $age1, $anciennete1);
presentation($nom2, $prenom2, $age2, $anciennete2);
```

Analyse de cas

La société R souhaite un programme afin d'organiser son personnel, elle fait appel à un développeur qui écrit le code suivant

On remarque qu'avec le code présenté :

- Nous avons **beaucoup de variables** pour représenter les données d'un employé.
- Une **gestion** de nouveaux employés **qui se complique**.
- Il est aussi possible que nous puissions **mélanger les variables**.
- Au final, **un code surchargé et difficile à maintenir**

```php
$nom1 = 'Hamada';
$prenom1 = 'Fahari';
$age1 = 35;
$anciennete1 = 12;

$nom2 = 'Dupont';
$prenom2 = 'Michel';
$age2 = 50;
$anciennete2 = 20;

function presentation($nom, $prenom, $age, $anciennete
{
  echo "Mon nom est $nom, mon prenom est $prenom,
    j'ai $age ans
    je travail depuis $anciennete ans.\n\r";
}

presentation($nom1, $prenom1, $age1, $anciennete1);
presentation($nom2, $prenom2, $age2, $anciennete2);
```

Organisation du code (1/6)

Une classe est une définition ou une représentation de quelque chose. La classe va contenir des propriétés qu'on pourra manipuler ainsi que des méthodes.

- **Les propriétés (attributs)**, ce sont des variables internes à cette définition dans lesquelles on stocke des valeurs.
- Une classe a aussi des **méthodes**, il s'agit de fonctions internes à la classe. Les méthodes representent le comportement qu'aura notre définition.

Ainsi, **la classe détermine ce qu'il sera possible de faire** avec ce qu'elle représente.

Elle va donc nous permettre de structurer et d'organiser l'ensemble de notre code pour le rendre plus solide et facile à entretenir ou à faire évoluer.

Organisation du code (2/6)

1. Les classes

1.1. Déclaration d'une classe:

Une classe se déclare de la manière suivante :

```
class Voiture
{

}
```

Note: Le mot clé class est suivi d'un nom en **PascalCase**, Prenez soin de nommer votre classe de manière pertinente.

Organisation du code (3/6)

1. Les classes

1.2. Instanciation d'une classe:

Créer une instance, c'est créer un objet à partir d'une classe. Nous utilisons le mot new pour instancier une classe. Cette instanciation sera alors affecté à une variable qui deviendra notre objet de type Voiture.

Exemple d'instanciation:

```php
$voiture = new Voiture();

var_dump($voiture);
```

À savoir qu'au moment de l'instanciation, une référence d'objet est mise en place par PHP.

Organisation du code (4/6)

1. Les classes

1.3. Référence d'objet :

Une « **référence** » en PHP ou plus précisément un **alias** est un moyen d'accéder au contenu d'une même variable en utilisant un autre nom. Plus simplement, créer un alias signifie déclarer un autre nom de variable qui va partager la même valeur que la variable de départ.

Notez qu'en PHP, le nom d'une variable et son contenu ou sa valeur sont identifiés comme deux choses distinctes par le langage. Cela permet donc de donner plusieurs noms à un même contenu.

Ainsi, lorsque nous modifions la valeur de l'alias, nus modifions également la valeur de la variable de base puisque ces deux éléments partagent la même valeur.

Déclaration par référence avec & :

```php
function foo(&$var) {
    $var = 2;
}
$a = 1;
foo($a);

// $a vaut 2 maintenant
var_dump($a);
```

Déclaration par référence avec l'instanciation :

```php
$date1 = new DateTime();
$date2 = $date1;
$date3 = clone $date1;
$date2->modify('+1 day');

var_dump($date1, $date2,
$date3);
```

`$date1` et `$date2` désignent le même objet en mémoire. Ils sont donc tous les deux au lendemain.

Organisation du code (5/6)

2. Propriétés (attributs) d'une classe

Les propriétés sont les variables membres de la classe. Ils constituent les caractéristiques de l'objet. En quelque sorte ce qui l'a définit.

```php
class Voiture
{
  $vitesse = 100;
  $carburant = 'diesel';
}
```

Pour déclarer un attribut, il faut le précéder par sa visibilité. La visibilité d'un attribut indique à partir d'où nous pouvons y avoir accès. public est un type de visibilité qui permet à l'attribut d'être accessible de partout (de l'intérieur de la classe dont il est membre comme de l'extérieur).

```php
class Voiture
{
  public $vitesse = 100;
  public $carburant = 'diesel';
}

$voiture = new Voiture();
$voiture->vitesse = 120;
$voiture->carburant = 'ethanol';
var_dump($voiture);
```

Nous utilisons l'opérateur `->` pour avoir accès **aux propriétés et méthodes** de l'objet instancié.

Organisation du code (6/6)

3. Méthodes d'une classe

Nous allons également pouvoir déclarer des fonctions à l'intérieur de nos classes. Les fonctions définies à l'intérieur d'une classe sont appelées des **méthodes**. Les méthodes de classes vont donc être ce qui défini le comportement de notre objet.

```php
class Voiture
{
  public $vitesse = 100;
  public $carburant = 'diesel';

  public function rouler()
  {
    echo 'Voiture '.$this->carburant.', roulant à
'.$this->vitesse.' KM/h';
  }
}

$voiture  = new Voiture();
$voiture->rouler();
```

Vous remarquez que, pour utiliser les propriété de la classe, nous utilisons le mot clé `$this`. La pseudo-variable `$this` est disponible lorsqu'une méthode est appelée depuis un contexte objet. `$this` est la valeur de l'objet appelant.

Note : Par défaut la méthode a une visibilité public, nous pourrons donc ne pas le mentionner mais par convention, il est préférable de le définir.

TD

Réalisez l’exercice 1

L’encapsulation (1/8)

1. Visibilité private

Nous avons défini les attributs avec une visibilité public, ces attributs peuvent être modifié par n'importe qui à un autre instant dans le code.

Pour remédier à cela, nous allons utiliser le concept d'**encapsulation** afin de protéger certaines données des interférences extérieures, en se forçant à utiliser les méthodes définies pour manipuler les données. L’encapsulation va ici être très intéressante pour empêcher que certaines propriétés ne soient manipulées depuis l’extérieur de la classe.
Nous allons donc utiliser la visibilité private au niveau de nos propriétés.

```php
class Voiture
{
  private $vitesse = 100;
  private $carburant = 'diesel';

  ...
}
```

private ne va être accessible uniquement depuis l’intérieur de la classe. Pour y avoir accès, nous utiliserons les accesseurs.

L'encapsulation (2/8)

2. Les accesseurs et mutateurs

Nous ne pouvons plus accéder directement aux attributs d'un objet. Pour lire et modifier leurs valeurs, nous passerons par des méthodes qui permettent de sécuriser leur utilisation.

Ces méthodes s'appellent **accesseurs** (**getter**) pour lire leurs valeurs et **mutateurs** (**setter**) pour modifier leurs valeurs.

Une des conventions souvent utilisées, est de reprendre le nom de l'attribut pour créer les méthodes, en ajoutant get pour les accesseurs et set pour les mutateurs.

```php
class Voiture
{
  ...
  public function setVitesse($vitesse)
  {
    $this->vitesse = $vitesse;
  }

  public function getVitesse()
  {
    return $this->vitesse;
  }

  public function setCarburant($carburant)
  {
    $this->carburant = $carburant;
  }

  public function getCarburant()
  {
    return $this->carburant;
  }
  ...
}
```

L'encapsulation (3/8)

2. Les accesseurs et mutateurs

Pour pouvoir donc modifier les information de notre objet, nous passerons par les accesseurs et mutateurs, qui sont des méthodes déclarées public de l'objet.

```php
$voiture = new Voiture();
$voiture->setCarburant('essence');
$voiture->setVitesse(120);
$voiture->rouler();
```

Nous pouvons ainsi sécuriser la modification de nos propriétés.

Exemple : Nous ne souhaitons modifier le carburant que si la nouvelle valeur est parmi "diesel", "essence" ou "ethanol".

```php
class Voiture
{
  ...
  public function setCarburant($carburant)
  {
    if(in_array($carburant, ['diesel', 'essence', 'ethanol'])) {
      $this->carburant = $carburant;
    }
  }
}

$voiture = new Voiture();
$voiture->setCarburant('eau');
var_dump($voiture);
```

L'encapsulation (4/8)

3. Le constructeur

PHP permet de déclarer des constructeurs pour les classes via la fonction magique `__construct()`. Les classes qui possèdent cette méthode l'appellent à chaque création d'une nouvelle instance d'objet, ce qui est intéressant pour toutes les initialisations, dont l'objet a besoin avant d'être utilisé.

```php
class Voiture
{
  private $vitesse = 100;
  private $carburant = 'diesel';

  public function __construct($vitesse, $carburant)
  {
    $this->vitesse = $vitesse;
    $this->setCarburant($carburant);
  }
  ...
}

$voiture = new Voiture(90, 'essence');
var_dump($voiture);
```

TD

Réalisez l’exercice 2

L’encapsulation (5/8)

4. Méthodes magiques

Les méthodes magiques sont des méthodes qui, si elles sont déclarées dans une classe, ont une fonction déjà prévue par le langage.

__construct() : Constructeur de la classe.

__destruct() : Destructeur de la classe.

__set() : Déclenchée lors de l'accès en écriture à une propriété de l'objet.

__get() : Déclenchée lors de l'accès en lecture à une propriété de l'objet.

__call() : Déclenchée lors de l'appel d'une méthode inexistante de la classe (appel non statique).

__callstatic() : Déclenchée lors de l'appel d'une méthode existante de la classe (appel statique), disponible depuis PHP 5.3.

__isset() : Déclenchée si nous appliquons isset() à une propriété de l'objet.

__unset() : Déclenchée si nous appliquons unset() à une propriété de l'objet.

L’encapsulation (6/8)

4. Méthodes magiques

__sleep() : Exécutée si la fonction serialize() est appliquée à l'objet.

__wakeup() : Exécutée si la fonction unserialize() est appliquée à l'objet.

__toString() : Appelée lorsque l'on essaie d'afficher directement l'objet : echo $object.

__set_state() : Méthode statique lancée lorsque l'on applique la fonction var_export() à l'objet

__clone() : Appelés lorsque l'on essaie de cloner l'objet ;

__autoload() : Cette fonction n'est pas une méthode, elle est déclarée dans le scope global et permet d'automatiser les "include/require" de classes PHP.

L'encapsulation (7/8)

5. Attributs et Méthodes static

Les attributs et méthodes statiques appartiennent à la classe et non à l'objet. Par conséquent, nous ne pouvons pas y accéder par l'opérateur `->` mais plutôt par l'opérateur de résolution de portée `::` précédé par le nom de la classe dans laquelle ils sont définis. Pour spécifier si une propriété est statique, nous déclarons le mot clé `static` après la visibilité. La valeur d'une propriété statique peut être modifiée, tout au long du processus, après l'avoir défini dans la classe.

Exemple :

```php
class Voiture
{
  public static $nbPortes = 5;
  ...

  public static function ajouterPortes()
  {
    self::$nbPortes++;
    echo sprintf("Une voiture avec %d portes", self::$nbPortes);
  }
}

var_dump(Voiture::$nbPortes, Voiture::ajouterPortes());
```

Note : On utilise un nouveau mot clé **self**. Dans la définition d'une classe, « `$this` » se réfère à l'objet actuel, tandis que « `self` » se réfère à la classe actuelle.

L'encapsulation (8/8)

6. Les constantes

Une constante de classe est un élément statique par défaut. Son rôle est le même que celui d'une constante classique déclarée à l'aide de la fonction `define()`. **Sa valeur est inchangée, contrairement à la propriété statique** et elle appartient aussi à la classe dans laquelle elle est déclarée et non à l'objet qui constitue l'instance de classe.

Pour définir une constante, nous utilisons le mot clé `const` suivi du nom de la constante à laquelle nous affectons la valeur souhaitée. Par convention, l'identifiant de la constante est déclaré en **majuscule**.
La constante de classe peut être appelée de l'intérieur comme de l'extérieur de la classe grâce à l'opérateur de résolution de portée `::`

```php
class Voiture
{
  const NB_ROUES = 4;
  ...

  public function rouler()
  {
    echo 'Voiture '.$this->carburant.', roulant à '
      .$this->vitesse.' KM/h avec '.self::NB_ROUES.' roues';
  }
}

var_dump(Voiture::NB_ROUES);
```

TD

Réalisez l’exercice 3

Réutilisation du code (1/8)

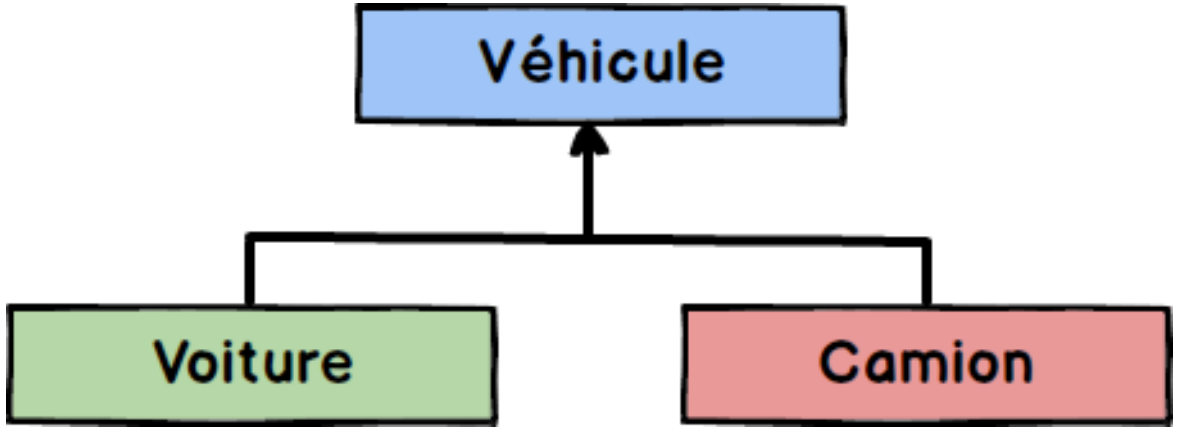
Héritage

L'héritage est un concept fondamental de la POO. C'est d'ailleurs l'un des plus importants, puisqu'il permet de réutiliser le code d'une classe autant de fois que l'on souhaite tout en ayant la liberté de modifier certaines parties.

Voyons notre exemple :

Nous disposons d'une classe du nom de **Vehicule**. Si nous créons une classe du nom de **Voiture** qui hérite de **Vehicule**, alors **Voiture** hérite de tous les membres (attributs et méthodes) qui constituent **Vehicule**.

Autrement dit, si nous instancions **Voiture**, alors tous les attributs et méthodes de **Vehicule** peuvent être appelés à partir de l'objet créé (l'instance de **Voiture**). Bien entendu, il faut que les membres appelés soient publiques.
Dans ce cas, **Vehicule** est la classe **mère** et **Voiture** est la classe **fille**.



Réutilisation du code (2/8)

Héritage:

1. Mot clé « extends »

Pour procéder à l'héritage, nous faisons appel au mot clé **extends** après le nom de la classe, comme ceci:

```
class Vehicule
{
  ...
}

class Voiture extends Vehicule
{
  ...
}
```

Notre classe Voiture étend la classe Vehicule. Elle hérite et va pouvoir accéder à toutes les méthodes et aux propriétés de notre classe Vehicule qui n'ont pas été définies avec le mot clé private.

Réutilisation du code (3/8)

Héritage:

2. Visibilité « protected »

Avec la visibilité **protected:** l'attribut est accessible seulement de l'intérieur de la classe dont il est membre ainsi que de l'intérieur des classes fille qui héritent de cette classe.

```
class Vehicule
{
  private $vitesse = 120;
  protected $carburant = 'diesel';
  ...
}

class Voiture extends Vehicule
{
  private $vitesse = 90; // Error
  protected $carburant = 'diesel';
  ...
}
```

Nous allons avoir une erreur car $vitesse est private et ne peut être accessible hors de la classe Voiture contrairement à $carburant.

Réutilisation du code (4/8)

Héritage:

2. Visibilité « protected »

Maintenant, nous allons ajouter quelques méthodes à notre classe Vehicule et regardons comment nous pouvons les utiliser dans les classes filles Voiture et Camion.

```php
class Vehicule
{
  const NB_ROUES = 4;
  protected $vitesse = 100;
  protected $carburant = 'diesel';
  public static $nbPortes = 5;

  public function __construct($vitesse, $carburant)
  {
    $this->vitesse = $vitesse;
    $this->carburant = $carburant;
  }

  public function rouler() { ... }
  public static function ajouterPortes() { ... }
}

class Voiture extends Vehicule { }

class Camion extends Vehicule { }
```

Réutilisation du code (5/8)

Héritage:

2. Visibilité « protected »

Instanciation et utilisation :

```php
$vehicule = new Vehicule(120, 'diesel');
$camion = new Camion(150, 'essence');
$voiture = new Voiture(90, 'ethanol');

$vehicule->rouler();
$camion->rouler();
$voiture->rouler();

var_dump($vehicule, $camion, $voiture);
```

Lorsqu'une **classe fille hérite d'une classe mère**, elle peut accéder et utiliser tous les membres non privés de la classe mère. Nous le savons, mais est-ce qu'une classe fille peut avoir ses propres propriétés et méthodes ? Oui, elle peut en avoir.

```php
class Vehicule { ... }

class Voiture extends Vehicule
{
  public function conduire()
  {
    echo 'Bonne conduite à '.$this->vitesse.' Km/h  ...\n';
  }
}

$voiture = new Voiture(90, 'ethanol');
$voiture->conduire();
```

Réutilisation du code (6/8)

Héritage:

3. Mot clé final

Le mot-clé **final** empêche les classes enfants de surcharger une méthode en préfixant la définition avec final. Si la classe elle-même est définie comme **finale**, elle ne pourra pas être étendue.

Cela peut être utile si vous souhaitez empêcher explicitement certains développeurs de surcharger certaines méthodes ou d'étendre certaines classes dans le cas d'un projet Open Source par exemple.

```
class Vehicule
{
  final public function rouler() { ... }
}

final class Voiture extends Vehicule
{

}
```

La méthode `Vehicule::rouler()` ne pourra pas être redéfini par les classes enfants

La classe `Voiture` ne pourra pas étendre à d'autres classes

Réutilisation du code (7/8)

Héritage:

4. Trait

PHP ne supporte que l'héritage simple, c'est à dire, une classe fille ne peut hériter que d'une seule classe mère. Que faire si une classe a besoin d'hériter de plusieurs classes mères ? Les **traits** résolvent ce problème.

Les **traits** sont utilisés pour déclarer les méthodes qui peuvent être utilisées dans plusieurs classes. Les **traits** peuvent avoir des méthodes abstraites qui peuvent être utilisées dans plusieurs classes. Les méthodes peuvent avoir n'importe quel modificateur d'accès (public, private, or protected).
Les traits sont déclarés avec le mot-clé **trait** :

```php
trait VitesseTrait
{
  protected $vitesse = 120;

  public function setVitesse($vitesse)
  {
    $this->vitesse = $vitesse;
  }

  public function getVitesse()
  {
    return $this->vitesse;
  }
}
```

Réutilisation du code (8/8)

Héritage:

4. Trait

Utilisation de trait dans une classe:

```php
class Vehicule
{
  use vitesseTrait;
  ...
}

$vehicule = new Vehicule();
$vehicule->setVitesse(170);
```

Si d'autres classes ont besoin d'utiliser la propriété **vitesse** et ses **accesseurs**, utilisez simplement le trait «VitesseTrait» dans ces classes. Cela réduit la duplication du code, car il n'est pas nécessaire de déclarer à nouveau la même propriété et méthodes encore et encore.

TD

Réalisez l’exercice 4

Plus de contrôle (1/13)

1. Interface

Une interface permet aux utilisateurs de créer des programmes, en spécifiant les méthodes publiques qu'une classe doit implémenter, sans impliquer la complexité et les détails de l'implémentation des méthodes. Une interface est définie comme une classe, mais avec le mot-clé interface. L'interface ne contient pas de propriétés ou de variables comme c'est le cas dans une classe.

```php
interface VehiculeInterface
{
  public function rouler();
  public static function afficherNbPortes();
}

class Vehicule implements VehiculeInterface
{ ... }
```

Une classe peut implémenter plusieurs interfaces, séparées par des virgules.

```php
interface VoitureInterface
{
  public function conduire();
}

class Voiture extends Vehicule implements VehiculeInterface, VoitureInterface
{ ... }
```

TD

Réalisez l’exercice 5

2. Abstraction

PHP a des classes et méthodes abstraites. Les classes définies comme abstraites ne peuvent pas être instanciées, et toute classe contenant au moins une méthode abstraite doit elle aussi être abstraite. Les méthodes définies comme abstraites déclarent simplement la signature de la méthode ; elles ne peuvent définir son implémentation.
Lors de l'héritage d'une classe abstraite, toutes les méthodes marquées comme abstraites dans la déclaration de la classe parente doivent être définies par la classe enfant et suivre les règles habituelles d'héritage et de compatibilité de signature.

Définition d'une classe abstraite

```php
abstract class AbstractVehicule
{
    private $vitesse;
    protected $carburant;
    abstract public function rouler();
}
```

Plus de contrôle (3/13)

Héritage de la classe abstraite

```php
class Voiture extends AbstractVehicule
{
  private $vitesse = 90; // Error
  protected $carburant = 'diesel';
  public function rouler()
  {
    echo 'Voiture '.$this->carburant.', roulant à '.
        $this->vitesse.' KM/h avec '.
        self::NB_ROUES.' roues';
  }
}
```

TD

Réalisez l’exercice 6

Plus de contrôle (4/13)

3. Polymorphisme

Le polymorphisme est un outil puissant et fondamental dans la programmation orientée objet. Il décrit un modèle dans lequel **les classes ont des fonctionnalités différentes tout en partageant une interface commune.**

La beauté du polymorphisme réside dans le fait que le code travaillant avec les différentes classes n'a **pas besoin de savoir quelle classe il utilise**, car elles sont toutes utilisées de la même manière.

Dans le monde de la programmation, le polymorphisme est utilisé pour rendre les applications plus modulaires et extensibles. Au lieu d'utiliser des instructions conditionnelles compliquées décrivant différents plans d'action, vous créez des objets interchangeables que vous sélectionnez en fonction de vos besoins. C'est l'objectif de base du polymorphisme.

Plus de contrôle (5/13)

3. Polymorphisme

Exemple:

```php
interface Transport {
  public function voyager(Voyageur $voyageur);
}

class Voiture implements Transport {
  public function voyager(Voyageur $voyageur) {
    return $voyageur->getNom().' voyage en voiture' ;
  }
}

class Avion implements Transport {
  public function voyager(Voyageur $voyageur) {
    return $voyageur->getNom().' voyage en avion' ;
  }
}

class Voyageur {
  private $nom;
  public function __construct($nom){
      $this->nom = $nom;
  }
  public function getNom(){
      return $this->nom;
  }
  public function voyager(Transport $transport){
      return $transport->voyager($this);
  }
}
```

3. Polymorphisme

Exemple:

```php
$dany = new Voyageur('Daniel');
$voiture = new Voiture();
$dany->voyager($voiture);
$avion = new Avion();
$dany->voyager($avion);
```

Cet exemple illustre le polymorphisme.

Un voyageur (`$dany`) a le choix entre deux transports : voiture ou avion. Quel que soit le transport, l'action sera appelée par la même méthode : dans notre cas, `voyager()`.

La méthode ne se soucie pas des détails de chaque voyage. En effet, chaque type de transport devient une classe qui définit les données du voyage.

TD

Réalisez l’exercice 7

4. Namespace

Dans ce tutoriel, nous allons apprendre les espaces de noms. En PHP, lorsque nous créons de grandes applications ou lorsque nous intégrons des applications/bibliothèques tierces, il peut y avoir des risques de conflits entre les noms de classes et les noms de fonctions. Pour éviter donc ces problèmes, les « espaces de noms » en PHP fournissent un moyen de regrouper les classes, interfaces, fonctions et constantes.

Pour définir un namespace, rien de plus simple, nous allons utilier le mot clé `namespace` juste avant la définition de la classe :

```php
# Interfaces/Vehicule.php
namespace Interfaces;
interface Vehicule {}

# Classes/Vehicule.php
require 'Interfaces/Vehicule.php';
namespace Classes;
use Interfaces\Vehicule as VehiculeInterface;
class Vehicule implements VehiculeInterface {}
```

L’espace de noms est utilisé pour éviter des conflits et introduire plus de flexibilité et d’organisation dans le code. Tout comme les répertoires, l’espace de noms peut contenir une hiérarchie connue sous le nom de sous-espaces. PHP utilise la barre oblique inversée « `\` » comme séparateur d’espace de noms.

Plus de contrôle (8/13)

4. Namespace

Conclusion

- Un espace de noms peut être considéré comme un concept abstrait. Il permet de re-déclarer les mêmes fonctions/classes/interfaces dans un espace de noms séparé sans obtenir l'erreur fatale.
- Un espace de noms est un bloc de code hiérarchiquement étiqueté contenant un code PHP régulier.
- Un espace de noms peut contenir du code PHP valide.
- Un espace de noms concerne les types de code suivants : classes (y compris les classes abstraits et les traits), interfaces, fonctions et constantes.
- Les espaces de noms sont déclarés en utilisant le mot-clé **namespace**.

TD

Réalisez l’exercice 8

5. Les exceptions

PHP a introduit une nouvelle façon de gérer la plupart des erreurs en utilisant les **exceptions**. Cette nouvelle façon de procéder se base sur le PHP orienté objet et sur la classe **\Exception**.

L'idée ici va être de créer ou de **« lancer (throw)»** un nouvel objet \Exception lorsqu'une erreur spécifique est détectée. Dès qu'une exception est lancée, le script va suspendre son exécution et le PHP va chercher un endroit dans le script où l'exception va être **« attrapée (catch) ».**

Utiliser des exceptions va nous permettre de gérer les erreurs de manière plus fluide et de personnaliser la façon dont un script doit gérer certaines erreurs.

Notez que quasiment tous les langages serveurs utilisent le concept d'exceptions pour prendre en charge les erreurs car c'est la meilleure façon de procéder à ce jour.

Comment utiliser les exceptions ?

Une exception peut être lancée throw et attrapée catch dans PHP. Le code devra être entouré d'un bloc try pour faciliter la saisie d'une exception potentielle.

Plus de contrôle (10/13)

5. Les exceptions

Nous allons améliorer l'accesseur setCarburant de notre classe Vehicule, en utilisant la gestion des erreurs via les exceptions.

```php
class Voiture
{
  ...
  public function setCarburant($carburant)
  {
    if(!in_array($carburant, ['diesel', 'essence', 'ethanol'])) {
      throw new \Exception("La voiture ne supporte pas le carburant ".$carbrant, 1);
    }
    $this->carburant = $carburant;
  }
}

try {
  $voiture = new Voiture();
  $voiture->setCarburant('eau');
} catch (\Exception $e) {
  echo 'Message d\'erreur : ' .$e->getMessage();
  echo 'Code d\'erreur : ' .$e->getCode();
  echo $e->getFile();
}
```

L’idée derrière les exceptions va être d’anticiper les situations problématiques (situations qui vont pouvoir causer une erreur) et de lancer une exception si la situation est rencontrée.

TD

Réalisez l’exercice 9

6. L'autoloader

Jusqu'ici, nous avons défini les classes et nous les avons instancié pour nous servir des objets qui en sont les instances, et tout ceci dans la même page PHP. Bien que ça marche, ce n'est cependant pas une méthode de travail propre et organisée.

Nous conseillons donc de toujours séparer les classes dans différents fichiers.

Le problème c'est que nous sommes obligés ensuite de faire beaucoup de **require** pour charger nos différentes classes. Heureusement, l'**autoloading** nous permet de remédier à ce problème en incluant les classes dès que l'on en a besoin.
Le principe de base est de créer une fonction **__autoload** qui permettra à PHP de savoir comment include nos classes.

```php
function __autoload($className){
    require_once './' . str_replace('\\', '/', $className) . '.php';
}
```

Le problème de cette méthode c'est que nous ne pouvons pas avoir plusieurs fois la même fonction et nous ne pouvons donc pas créer plusieurs autoloader.

Plus de contrôle (12/13)

6. L'autoloader

Heureusement, il est possible de créer et d'enregistrer des fonctions manuellement en utilisant **spl_autoload_register**

```php
# Autoloader.php
class Autoloader
{
  /** Enregistre notre autoloader */
  public static function register()
  {
      spl_autoload_register(array(__CLASS__, 'autoload'));
  }

  /**
   * Inclue le fichier correspondant à notre classe
   * @param string $class Le nom de la classe à charger
   */
  public static function autoload($className)
  {
      require_once str_replace('\\', '/', $className) . '.php';
  }
}
```

Pour une meilleure organisation, nous utilisons ici une classe avec des méthodes statiques. Nous pouvons alors simplement lancer notre autoloader.

```php
# index.php
require 'Autoloader.php';
Autoloader::register();
```

Plus de contrôle (13/13)

7. Classe Anonymes

Lors de la création d'un nouvel objet, une classe est d'abord définie, puis un objet de cette classe est créé. En PHP 7, une classe dite anonyme a été introduite pour permettre aux classes d'être définies sans nom à la volée.

Nous définissons une classe anonyme en utilisant le mot-clé new class.

Exemple:

```php
$obj1 = new class() {};
$obj2 = new class($x, $y) {
   private $x;
   private $y;
   public function __construct($x, $y) {
         $this->x = $x;
         $this->y = $y;
   }
};
```

Elles peuvent prendre des arguments via le constructeur, hériter d'autres classes, implémenter des interfaces, et utiliser des traits comme dans une classe normale.

Les objets instanciés par une classe anonyme ne sont pas différents de ceux instanciés par une classe normale.

```php
$classX = new class extends Avion implements Transport {
  use VitesseTrait;
};

$classX->setVitesse(120);
var_dump($classX);
```

PROJET

Voir ressource Projet

1. Composer

1.1. Started

url : https://getcomposer.org/

Installer composer

```
curl -sS https://getcomposer.org/installer | php -- \
    --install-dir=/usr/local/bin \
    --filename=composer
```

La commande suivante permettra d'initialiser composer avec le répertoire vendor

```
composer init
```

Nous avions précédemment fait notre propre autoloader, mais composer peut en générer un aussi, qui comprendra tout aussi les namespaces des composants que l'on pourrait installer. Ajouter une section autoload au fichier composer.json

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

Après avoir ajouter notre section d'autoload vers notre répertoire src, faire générer un fichier autoload.

Bonus (2/4)

1. Composer

1.2. Using

Dans votre fichier index.php, il faut ajouter la ligne suivante:

```php
<?php

require_once 'vendor/autoload.php';
```

Installer un package

```
# install package http-client
composer require symfony/http-client

# install package to debug
composer require symfony/var-dumper
```

Vous pouvez retrouver la liste de tous les packages sur https://packagist.org/

Bonus (3/4)

2. PHP Doc

C'est une transposition de Javadoc au langage PHP. Il s'agit d'un standard formalisé pour commenter le code PHP. Il permet aussi à certains IDE de connaître le type des variables et de lever d'autres ambiguïtés dues au typage faible, améliorant ainsi la complétion de code.

```php
class Vehicule
{
  ...
  /**
    * @var int Vitesse du véhicule
    */
  protected $vitesse = 100;

  /**
    * @param int $vitesse Vitesse du véhicule
    * @param string $carburant Carburant du véhicule
    */
  public function __construct($vitesse, $carburant)
  { ... }

  /**
    * @return int Vitesse du véhicule
    */
  public function getVitesse()
  { ...  }
  ...
}
```

3. Typage de données

Si vous souhaitez utiliser cette fonctionnalité, il est conseillé de **toujours activer** le **declare(strict_types=1);** dans vos fichiers.

```php
class Vehicule
{
  private ?int $vitesse = null;
  private ?string $carburant = null;

  public function rouler(?int $hour) : int
  {
    return $hour ? $hour * $vitesse : null;
  }
}

class Conducteur
{
  private ?Vehicule $vehicule = null;

  public function __construct(Vehicule $vehicule)
  {
    $this->vehicule = $vehicule;
  }
}
```

Introduction

Cette nouvelle mise à jour majeure apporte tout un tas d'optimisations et de puissantes fonctionnalités au langage. Ce sont des changements très intéressants qui nous permettront d'écrire un meilleur code et de construire des applications plus puissantes.

1. Type d'union

Vous pouvez utiliser plusieurs types d'entrée pour la même fonction au lieu d'un seul, ce qui permet un plus grand degré de réutilisation du code.

```php
class Vehicule
{
  private int|float|null $vitesse = null;
  public function setVitesse(int|float|null $vitesse) : self
  {
    $vitesse = $vitesse;
    return $this;
  }
}
```

Vous pouvez aussi utiliser des classes et interfaces définies.

```php
class Conducteur
{
  public function conduire(Voiture|Camion $vehicule)
  {
    ...
  }
}
```

2. Promotion de propriétés de constructeur

Cette fonction devrait vous aider à accélérer votre processus de développement et à réduire les erreurs. En effet, moins de code redondant pour définir et initialiser les propriétés.

```php
class Vehicule
{
  public function __construct(
    private ?int $vitesse = 120,
    protected ?string $carburant = 'essence'
  ) {
    ...
  }
}
```

3. Visibilité pour les constantes

```php
class Vehicule
{
  private const CONST_PRIVATE = 'private';
  protected const CONST_PROTECTED = 'protected';
  public const CONST_PUBLIC = 'public';
  const CONST_DEFAULT_PUBLIC = 'default_public';
  ...
}
```

4. Arguments nommés

Les arguments nommés vous donnent plus de souplesse pour appeler les fonctions. Jusqu'à présent, vous deviez appeler une fonction et passer chaque argument dans l'ordre spécifié par la fonction.

```
// Using positional arguments:
array_fill(0, 100, 50);
```

Les arguments nommés vous permettent de définir un nom pour chaque paramètre. Et maintenant, ils peuvent être rappelés à l'ordre, comme décrit ci-dessous :

```
// Using named arguments:
array_fill(start_index: 0, num: 100, value: 50);
```

5. Fonction str_contains

Cette nouvelle fonction plutôt sympathique renvoie une valeur booléenne (vrai/faux) si une chaîne est trouvée dans une autre chaîne. Il faut deux arguments et la chaîne de caractères à rechercher.

```
str_contains('php8', '8'); // true

str_contains('php8', 'wordpress'); // false
```

Pour des filtres de chaînes encore plus utiles, découvrez ces nouvelles fonctionnalités :

```
str_starts_with('haystack', 'hay'); // true

str_ends_with('haystack', 'stack'); // true
```