⬅Revenir à la théorie.

Rappels

- 1. Introduction
- 2. Fonctions anonymes
    - 2.1 Auto-invoquer une fonction anonyme
    - 2.2 Utiliser une fonction anonyme comme paramètre d'une fonction.
    - 2.3: Affectation d'une fonction anonyme dans une variable.
    - 2.4: Passage d'une variable d'une autre portée (scope) à une fonction anonyme
    - 5.5: Un peu de tout 😄
- 3. La programmation orientée object (POO)
    - 3.1 Classe
        - 3.1.1 Constructeur
        - 3.1.2 Attribut
        - 2.1.3 Méthode
        - 3.1.4 L'encapsulation
        - 3.1.5 Visibilité d'une méthode/attribut: public, private, protected
    - 3.2 Objets / Instances
    - 3.3 Héritage

1. Introduction

Ici seront vite revues certaines notions PHP nécessaires pour aborder Laravel.
Nous n'irons pas dans le détail mais ces notions sont des prérequis pour ce cours.
Ces notions ont déjà été vues au cours de PHP.

2. Fonctions anonymes

Les fonctions anonymes sont des fonctions qui ne possèdent pas de nom.
Une fonction anonyme s'écrite de la même manière qu'une fonction normale sauf qu'elle n'a pas

de nom.

2.1 Auto-invoquer une fonction anonyme

```
(function(){
    echo 'Fonction anonyme bien exécutée';
})();
```

On entoure la fonction anonyme d'un premier couple de parenthèses et d'ajouter un autre couple de parenthèses à la suite du premier couple.

2.2 Utiliser une fonction anonyme comme paramètre d'une fonction.

```
$strings = ["APPLE", "oRaNgE", "BANANA", "COCONUT"];
$lengths = array_map(function ($item) { return strtolower($item);}, $strings);
print_r($lengths);
```

La sortie donnera ceci:

```
Array
(
    [0] => apple
    [1] => orange
    [2] => banana
    [3] => coconut
)
```

La fonction array_map() envoie chaque valeur d'un tableau à une fonction créée par l'utilisateur et renvoie un tableau avec de nouvelles valeurs, données par la fonction créée par l'utilisateur. Ici, on voit donc avec la fonction array_map que l'on a passé comme paramètre une fonction anonyme qui retourne une string en miniscules pour que valeur envoyée.
Il aurait aussi été possible de faire ceci sans fonction anonyme:

```php
declare(strict_types=1);

function tolower(string $string): string{
    return strtolower($string);
}
$strings = ["APPLE", "oRaNgE", "BANANA", "COCONUT"];
$lengths = array_map("tolower", $strings);
print_r($lengths);
```

Remarquez que dans le code précédent on a appelé la fonction tolower entre guillemets.

2.3: Affectation d'une fonction anonyme dans une variable.

```php
$salutation = function($name){
    echo "Hello $name !";
};

$salutation("Philip");
```

On a défini une variable $salutation qui contient une fonction anonyme qui accepte un paramètre $name.
Lorsque l'on va faire $salutation('Philip'), on va appeler la fonction anonyme contenue dans $salutation avec le paramètre Philip. Celle-ci affichera alors: Hello Philip !

2.4: Passage d'une variable d'une autre portée (scope) à une fonction anonyme

Comme vous le savez une variable est visible dans sa portée. En anglais, on utilise le terme de 'Scope'. Cette notion de portée existe dans tous les langages.
Pour qu'une fonction anonyme ait accès à une variable hors de sa portée on utilise le mot clef use suivit de la variable entre parenthèses.

```php
$name = "Johnny";
$salutation = function() use ($name){
    echo "Hello $name !";
}

$salutation();
```

Ici, la sortie de ce programme donnera: Hello Johnny !
On pourrait se dire que c'est exactement pareil que le programme précédent. A la différence que la fonction anonyme est appélée sans paramètre et que l'on accède à l'intérieure de la fonction à $name grâce à l'utilisation de 'use'. Car rappelons-le, $name n'est pas dans le même Scope que celui de la fonction anonyme.

5.5: Un peu de tout 😀

```php
declare(strict_types=1);

/**
    Fonction qui affiche une chaîne selon la fonction fournie en paramètre
    @param callable $format fonction à appeler.
    @param string $str chaîne de caractère à traîter.
*/
function printFormatted(callable $format, string $str): string
{
    echo "[ ".$format($str)." ] <br/>";
}

// Une fonction anonyme qui retourne les 5 premiers caractères (si chaine >=5 caractères)
$func1 = function (string $str) {
    if (strlen($str) >= 5)
        return substr($str, 0, 5);
    else
        return $str;
};

// Une fonction anonyme qui retourne une chaine écrite un caractère sur 2 en majuscule, l
$func2 = function (string $str) {
    $result = '';
    for ($x = 0; $x < strlen($str); $x++) {
        $current = $str[$x];
        $modulo = ($x + 1) % 2;
        //Si le reste de la division par 2 (modulo 2) est égale 0
        //Alors on ajoute la lettre courante en minuscule à $result
        //Sinon on ajoute la lettre courante en majuscule à $result
        $result .= ($modulo == 0) ? strtolower($current) : strtoupper($current);
    }
    return $result;
};

// Fonction anonyme qui retourne une chaîne en Leet Speak.
// Exemple: Leet Speak deviendra 133t s|*34X
$func3 = function (string $str) {
    return str_replace(
        array('a', 'b', 'c',  'd',  'e',  'f',  'g', 'h', 'i', 'j',  'k',  'l', 'm', 'n',
```

```
        array('4', '8', '(',  '[)',  '3', '|=', '9', '#', '1', '_|',  'X',  '1', '|v|', '
        strtolower($str)
    );
};

$chaine = "Hello Philip & Johnny";
printFormatted($func1, $chaine);
printFormatted($func2, $chaine);
printFormatted($func3, $chaine);

// Une chaîne contenant le nom de la fonction
printFormatted("strtoupper", $chaine);
printFormatted("strtolower", $chaine);
printFormatted("strrev", $chaine);

// On passe directement une fonction anonyme en paramètre
// Cette fonction anonyme retourne une chaine au format suivant: ### $str ### où les espa
printFormatted(
    function ($str) {
        return "### " . str_replace(' ', '', $str) . " ###";
    },
    $chaine
);
```

Voici le résultat de ce programme:

```
[ Hello ]
[ HeLlO PhIlIp & jOhNnY ]
[ #3110 |*#111|* & _|0#^/^/y ]
[ HELLO PHILIP & JOHNNY ]
[ hello philip & johnny ]
[ ynnhoJ & pilihP olleH ]
[ ### HelloPhilip&Johnny ### ]
```

Discutons-en ! 😀

3. La programmation orientée object (POO)

Ce que dit Wikipedia:

> *La programmation orientée objet (POO), ou programmation par objet, est un paradigme de programmation informatique. Elle consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle.*

> *La programmation par objet consiste à utiliser des techniques de programmation pour mettre en œuvre une conception basée sur les objets. Celle-ci peut être élaborée en utilisant des méthodologies de développement logiciel objet, dont la plus connue est le processus unifié (« Unified Software Development Process » en anglais), et exprimée à l'aide de langages de modélisation tels que le Unified Modeling Language (UML).*

Pour faire dans le concret, je vais partir directement depuis un exemple. Et les différents concepts seront brièvement expliqués depuis la classe Cupcake. ☺

```php
declare(strict_types=1);

class Cupcake
{
    public string $nom;
    private bool $_contientLait;
    private bool $_contientGluten;
    private array $_ingredients;
    public const FOUR = true;

    public function __construct(string $nom, array $ingredients, bool $contientLait, bool
    {
        $this->nom = $nom;
        $this->_contientLait = $contientLait;
        $this->_contientGluten = $contientGluten;
        $this->_ingredients = $ingredients;
    }

    public function getName(): string
    {
        return "Ce cupcake s'appelle <b>$this->nom</b>.<br/>";
    }

    public function displayName(): void
    {
        echo $this->getName();
    }

    private function _getIngredients(): string
    {
        $_ingredients = "<u>Liste des ingrédients</u>:";
        $_ingredients .= "<ul>";
        foreach ($this->_ingredients as $ingredient => $quantite) {
            $_ingredients .= "<li>$ingredient avec $quantite.</li>";
        }
        $_ingredients .= "</ul>";
        return $_ingredients;
    }
```

```php
    private function _getAllergenes(): string
    {
        return "<div style='color:red'>"
            . ($this->_contientLait ? 'Il contient du lait.<br/>' : '')
            . ($this->_contientGluten ? 'Il contient du gluten.<br/>' : '')
            . "</div>";
    }

    public function displayAllergenes(): void
    {
        echo $this->_getAllergenes();
    }

    public function displayIngredients(): void
    {
        echo $this->_getIngredients();
    }

    public function displayCuisson(): void
    {
        echo "$this->nom " . ($this::FOUR ? ' doit être cuit au four.' : ' ne doit pas êt
    }

    public function __toString(): string
    {
        return $this->getName()
            . $this->_getAllergenes()
            . $this->_getIngredients();
    }
}

$cupcake1 = new Cupcake("Pinky", array("sucre" => "12 gr", "lait" => "10 ml"), true, true
$cupcake1->displayName();
$cupcake1->displayAllergenes();
$cupcake1->displayIngredients();
echo '<hr/>';
$cupcake2 = new Cupcake("Blue", array("sucre" => "17 gr", "lait" => "13 ml", "Chocolat" =>
echo $cupcake2;
echo $cupcake2->displayCuisson();
```

Ce code affichera ceci:

```
Ce cupcake s'appelle Pinky.
Il contient du lait.
Il contient du gluten.
Liste des ingrédients:
sucre avec 12 gr.
lait avec 10 ml.
------------------------
Ce cupcake s'appelle Blue.
Il contient du lait.
Liste des ingrédients:
sucre avec 17 gr.
lait avec 13 ml.
Chocolat avec 14 gr.
farine avec 5 gr.
Blue doit être cuit au four.
```

3.1 Classe

Une classe est un type de donnée particulier. Elle commence par le mot class et est suivi d'une accolade ouvrante, d'un corps et d'une accolade fermante:

```
class Cupcake
{
    //blablabla
}
```

En son sein sont encapsulés:

- des méthodes qui sont en fait des fonctions: displayName(), displayIngredients(), etc.
- des attributs qui ont un type: $nom, $ingredients, $contientLait, etc.
- un constructeur: qu'a-t-on besoin pour créer un objet du type Cupcake ?
  public function __construct(string $nom, array $ingredients, bool $contientLait, bool $contientGluten)
- des constantes: valeurs qui ne peuvent être changées.

public const FOUR = true;

3.1.1 Constructeur

Comme son nom l'indique, il permet de construire notre futur objet.
On fournit au constructeur, les paramètres nécessaires à sa construction.
Dans le constructeur suivant: public function __construct(string $nom, array $ingredients, bool $contientLait, bool $contientGluten)
Nous aurons besoin:

- du nom (string)
- des ingrédients (array)
- s'il contient du lait (bool)
- s'il contient du gluten (bool)

On utilise le mot clef new pour créer un objet avec son constructeur:
$cupcake1 = new Cupcake("Pinky", array("sucre" => "12 gr", "lait" => "10 ml"), true, true);

Dans ce constructeur qui est une fonction particulère, on peut y faire des affectations et diverses opérations nécessaires en fonction du contexte.

Si la classe n'a pas besoin de paramètres pour construire un objet, on n'est pas obligé d'indiquer un constructeur. PHP fait ce qu'on appelle un constructeur par défaut que vous pourrez utiliser directement.

Si notre classe Cupcake n'avait pas besoin de constructeur avec tous ces paramètres, on aurait pu écrire lors de la création de l'objet:

```
$cupcake1 = new Cupcake();
```

3.1.2 Attribut

Les attributs sont des variables qui définissent notre futur objet:

- $nom (string)
- $contientLait (bool)
- $contientGluten (bool)
- $ingredients (array)

Pour accéder à un attribut on utilise ->

```
public function getName(): string
{
    return "Ce cupcake s'appelle <b>$this->nom</b>.<br/>";
}
```

2.1.3 Méthode

Les méthodes, voyez-les comme des fonctions qui retourne ou non une valeur:

- getName retourne une string.
- displayName ne retourne pas de valeur (void): Affiche le nom du cupcake.
- getIngredients retourne une string.
- displayIngredients ne retourne pas de valeur (void): Affiche la liste des ingrédients du cupcake.

Pour accéder à une méthode on utilise ->

```
public function displayIngredients()
{
    echo $this->_getIngredients();
}
```

3.1.4 L'encapsulation

L’encapsulation va ici être très intéressante pour empêcher que certaines propriétés ne soient manipulées depuis l’extérieur de la classe. Pour définir qui va pouvoir accéder aux différents attributs, méthodes et constantes de nos classes, nous allons utiliser des limiteurs d’accès ou des niveaux de visibilité qui vont être représentés par les mots clefs public, private et protected.

3.1.5 Visibilité d'une méthode/attribut: public, private, protected

C'est très simple qu'est-ce qui peut être visible (attributs/méthodes) depuis l'extérieur de la classe ?

- Si c'est visible: c'est public. On peut accéder soit aux attributs publics ou aux méthodes publiques.

- Si c'est invisible: c'est private. On ne peut accéder à ces méthodes que dans la classe.

Alors pour l'écriture des membres privés d'une classe (méthodes/attributs), certains ont pris la convention d'écrire tout membre public avec un underscore. Pour l'exemple, j'ai appliqué cette convention mais personnelement je ne la pratique pas. Mais vous donner/imposer des 'Naming Conventions' n'est pas mauvais. ☺

```
public string $nom;
private bool $_contientLait;
```

C'est le dev qui décide ou non de la pertinence de mettre public ou private. On peut par exemple mettre une fonction en private pour éviter d'appeler cette fonction qui supprimerait un fichier.

Il existe aussi protected.
C'est comme private, on ne pourra accéder aux méthodes et attributs qu'à l'extérieur de la classe. La différence ?? Tout sera accessible depuis la classe ET les classes dérivées. Si vous mettez un membre privé, il ne sera pas accessible depuis une classe enfant.

Un avis personnel, si vous êtes certain que vous allez avoir des classes enfants de votre classe, je vous conseille de mettre protected au lieu de private. Comme ça vous pourrez avoir accès à ces membres protected. En effet, si vous avez des attributs/méthodes qui sont privées, il ne vous sera pas possible d'y accéder dans vos classe enfants.

3.2 Objets / Instances

Un objet, c'est la "matérialisation" d'une classe. D'un type théorique, la classe, on la concrétise en faisant un objet.
On utilise le mot clef new() et à l'intérieur des parenthèses on fournit les éventuels paramètres nécessaires au constructeur de la classe.

```
$cupcake1 = new Cupcake("Pinky", array("sucre" => "12 gr", "lait" => "10 ml"), true, true
```

On a créé ici un objet nommé $cupcake1 de type Cupcake.

3.3 Héritage

L'héritage c'est le fait de créer une classe en héritant des fonctions et attributs de notre classe parente si elles sont qualifiées de public ou protected.
On possède tous les avantages de notre parent et on peut ajouter/redéfinir des méthodes/ attributs.
On utilise le mot clef extends.

```
class Cake extends Cupcake
{
    public float $hauteur;

    public function displayHauteur(): void
    {
        echo "La hauteur de $this->nom est de $this->hauteur cm.<br/>";
    }

    public function displayName(): void
    {
        echo "Ce cake s'appelle <b>$this->nom</b>.<br/>";
    }

    public function __construct(string $nom, float $hauteur, array $ingredients, bool $co
    {
        parent::__construct($nom, $ingredients, $contientLait, $contientGluten);
        $this->hauteur = $hauteur;
    }
}

$cake = new Cake("Miam-Miam", 5.3, array("sucre" => "12 gr", "lait" => "10 ml"), true, tr
$cake->displayName();
$cake->displayIngredients();
$cake->displayHauteur();
```

Ce code affichera:

```
Ce cake s'appelle Miam-Miam.
Liste des ingrédients:
sucre avec 12 gr.
lait avec 10 ml.
La hauteur de Miam-Miam est de 5.3 cm.
```

Notre nouvelle classe Cake va hériter des méthodes et attributs de la classe parente si celles-ci sont public ou protected.

⬅Revenir à la théorie.
