R3.01 - ProgWeb
TP 3 - POO

Introduction

Comme pour les TD et TP précédents, vous lancerez un **`php -S localhost:8888`** pour servir de mini serveur local de test durant ce TP.

PHP vs le reste du monde

Vous avez déjà appréhendé la POO (Programmation Orientée Objet) avec Java en BUT1.

PHP ne révolutionne pas le concept mais apporte sa touche personnelle pour en simplifier certains aspects.

PHP est venu plutôt tardivement à la POO, un peu forcé sans doute par les attentes et les besoins de ses utilisateurs. De ce fait, comme il n'a pas été conçu au départ pour l'objet. On ressent bien le choix assumé par ses concepteurs d'adapter la POO à PHP et non l'inverse.

Il faut voir la POO dans PHP comme une fonctionnalité du langage et non pas un principe fondamental de fonctionnement. Alors qu'un langage comme Java est purement objet, PHP est tout à fait utilisable en premier lieu comme un simple langage procédural, et c'est d'ailleurs ainsi que vous l'avez expérimenté jusqu'à présent.

Terminologie Objet

Nous n'allons pas entrer en détail dans ce qui est commun aux principes et aux langages de POO, tels que Java que vous connaissez déjà. Nous nous arrêterons plutôt sur ce qui fait de PHP un langage de POO différent des autres.

Dans les exemples qui vont suivre, les portions écrites **`...`** représentent du code absent ou non détaillé ici. Si vous testez ces exemples, il faudra évidemment compléter la partie manquante car ces exemples de code ne sont pas exécutables en l'état.

Même si ce n'est pas indiqué dans le sujet, vous êtes invités à expérimenter les portions de code données en exemple et à solliciter votre enseignant si besoin.

Principes de base

Classe

Description de la nature (composition) des objets appartenant à cette classe : ses attributs et ses méthodes, qu'on va décrire un peu plus loin. Une classe possède un nom et doit être déclarée avant de pouvoir être utilisée, à l'aide du mot-clé **`class`**.

Une classe est donc un modèle servant à instancier des objets (voir l'instanciation plus loin).

Héritage

Une classe dérive toujours d'une autre classe, c'est ce qu'on appelle communément l'héritage en POO. En PHP, l'héritage se fait à l'aide du mot-clé **`extends`**. Si on ne précise rien, une classe hérite automatiquement de la superclasse **`Object`** qui est la classe ancêtre de toutes les autres.

Syntaxe simple, sans déclaration explicite d'héritage (donc héritage implicite de **`Object`**) :

```
class <nom> {
...
}
```

Syntaxe avec un héritage explicite :

```
class <nom> extends <autre_classe> {
...
}
```

En PHP il n'y a pas d'héritage multiple, une classe ne peut hériter que d'une seule autre classe. Évidemment l'héritage en cascade est possible : B hérite de A et C hérite de B.

Instanciation & Objet

L'instanciation permet de créer des objets à partir d'une classe. On utilise le mot-clé **`new`** pour instancier une classe en un objet et généralement on affecte une variable avec le résultat de cette instanciation (i.e. l'objet retourné par **`new`**) :

```
class Document {
...
}

$un_doc = new Document();
```

Il est aussi possible de créer un objet de la façon suivante, en instanciant la superclasse **`Object`** (qui existe nativement dans PHP) :

```
$un_obj = new Object();
```

On reparle de l'intérêt de cette écriture plus tard au paragraphe **À savoir > Structures**.

Attribut

Composant de la classe, représenté par une variable au sein de la classe et permettant de stocker un état ou des informations associées à chaque objet de cette classe. Le type peut être n'importe quel type possible dans PHP, y compris une classe ou un tableau.

Il faut déclarer les attributs d'une classe de la manière suivante :

```
<visibilité> <nom_var> = <éventuelle_valeur_par_défaut>;
```

où **<visibilité>** est un mot-clé spécifiant la portée, parmi :

- **public** : l'attribut est accessible par celui qui possède un objet de cette classe
- **protected** : l'attribut est accessible par les méthodes de la classe (on en parle juste après) ou par toute autre classe qui hérite de cette classe.
- **private** : l'attribut n'est accessible que par les méthodes de la classe.

Exemples :

```
class Etudiant_UR {
    public $nom;
    public $prenom;
    public $numero;
...
}

class Etudiant_Lannion extends Etudiant_UR {
    public $groupe_tp;
    protected $photo;
    private $notes = [];
...
}
```

En l'absence de mot-clé de **<visibilité>**, la valeur **public** est utilisée par défaut.

Note : la **<éventuelle_valeur_par_défaut>** ne peut être qu'une valeur constante, pas une expression impliquant une variable ou un appel de fonction.

Pour accéder à un attribut, sous réserve de disposer du droit de le faire (visibilité **public** de l'attribut), on utilise la syntaxe suivante :

```
$objet->attribut
```

où **attribut** est le nom d'un attribut de la classe, sans indiquer le **$** habituel des variables ! Ceci s'explique car le **$** est déjà présent devant l'expression.

Exemple :

```
$etu = new Etudiant_UR();
$etu->prenom = "LULU";
echo "Voici un étudiant qui se prénomme " . $etu->prenom;
```

PHP étant connu pour sa permissivité, accéder à un attribut inexistant ne pose aucun problème, comme pour une variable classique. Affecter une valeur à un attribut

inexistant ajoute même cet attribut à l'objet ! Attention, il est ajouté à l'objet et non pas à la classe. Les autres objets de cette classe ne possèdent donc pas cet attribut supplémentaire ajouté manuellement.

Exemple :

```
$etu = new Etudiant_UR();
$etu->prenom = "PIPO";
$etu->secret = "❤️LULU"; // Attribut inexistant dans la classe
echo "Je sais que $etu->prenom $etu->secret";
```

Méthode

C'est aussi un composant de la classe, représenté par une fonction au sein de la classe. Les méthodes décrivent des actions qui peuvent être exécutées sur un objet de la classe et qui agissent sur ou en relation avec l'objet. Elles n'ont d'usage que parce qu'elles sont liées à un objet.

Il faut déclarer les méthodes, comme pour les attributs, de la manière suivante :

```
<visibilite> fonction <nom_fonc>(...) {...}
```

où **<visibilité>** est le même mot-clé que pour les attributs et spécifie la portée parmi **public**, **protected** et **private**.

Pour appeler une méthode, sous réserve de disposer du droit de le faire (visibilité **public** de la méthode), on utilise la syntaxe suivante :

```
$objet->méthode();
```

où **méthode()** est le nom d'une méthode de la classe.

La méthode s'exécute donc dans le contexte d'un objet auquel elle est attachée. Cet objet est accessible, dans toutes les méthodes de la classe, par la pseudo-variable **$this**. On l'utilise ainsi :

```
$this->attribut
```

Cette pseudo-variable **$this** n'a d'existence qu'au sein d'un objet, et il y a donc autant de **$this** que d'objets. C'est un peu comme dire qu'il y a autant de "moi" qu'il y a d'êtres vivants, mais ce "moi" n'est utilisable (prononçable) que par un être vivant "depuis l'intérieur de lui-même".

Exemple :

```
class Horodateur {
    public $maintenant;

    function setNow() {
        $this->maintenant = time();
    }

    function fmtDate() {
        return date("Y-m-d", $this->maintenant);
```

```
        }

        function fmtHeure() {
            return date("H:i:s", $this->maintenant);
        }
    }
    $now = new Horodateur();
    $now->setNow();
    echo "Objet créé le " . $now->fmtDate() . " à " . $now->fmtHeure();
```

Par défaut, en l'absence de mot-clé de **<visibilité>** c'est la valeur **public** qui est utilisée.

Exercice 1

Vous connaissez d'autres langages de POO et vous savez qu'il existe des constructeurs dans ces langages. En PHP aussi et nous allons le voir juste après, mais à ce stade n'utilisez que ce que nous venons de voir, donc pas d'usage de constructeur et pas de Google non plus, tout ce dont vous avez besoin est dans les pages précédentes ou la documentation officielle.

Sur la base de la classe **Horodateur**, ajoutez :

- Un attribut **label** permettant d'associer un nom à l'objet.
- Une méthode **difference($dt)** qui accepte un timestamp Unix (rappel : c'est le nombre de secondes écoulées depuis le 1er janvier 1970) en paramètre **$dt** et qui retourne la différence entre la valeur $dt et la valeur $maintenant.

Testez votre classe et votre code ainsi :

- Créez deux objets
- Affectez-leur un label quelconque à chacun
- Avec chacun des objets, affichez le label et la différence de temps (en secondes) par rapport aux valeurs :
    - **1664524330**
    - **2074751530**

Exercice 2

Sur la base de l'exercice 1, ajoutez une méthode **estFutur($dt)** qui retourne une valeur booléenne indiquant si le timestamp **$dt** est dans le passé ou dans le futur par rapport au timestamp de l'objet. Testez avec les mêmes valeurs que précédemment (pour info la 1ère est dans la passé, la seconde dans le futur).

Vous pouvez aller sur le site https://www.unixtimestamp.com/ pour vous fabriquer des timestamps Unix à partir d'une date donnée.

Constructeur

Un constructeur permet de préciser des valeurs d'initialisation au moment de l'instanciation.

Il fait partie des méthodes "magiques" de PHP. C'est la terminologie officielle.

Toutes les méthodes magiques commencent par **_ _** (deux underscores)

Un constructeur :

- Se nomme toujours **__construct(...)**
- Est appelé automatiquement par le mot-clé **new**
- Est toujours **public**
- Peut recevoir des paramètres (ceux passés par le **new**)
- Ne doit jamais être appelé directement (rappel : c'est **new** qui s'en occupe)
- Est unique. Pas de constructeurs multiples

Exemple :

```
class Etudiant_UR {
     public $nom;
     public $prenom;
     private $numero;

     function __construct($name, $surname, $id) {
        $this->nom = $name;
        $this->prenom = $surname;
        $this->numero = $id;
     }
...
}
$lulu = new Etudiant_UR("L'ASTICOT", "LULU", 12345);
```

Comme dans cet exemple, le constructeur permet d'initialiser des attributs (ici **$numero**) qui peuvent être **protected** ou **private** et donc impossible à initialiser autrement.

Destructeur

Un destructeur est appelé quand l'objet (i.e. la variable) est supprimé de la mémoire.

Ca peut être :

- quand le programme s'arrête
- quand la variable sort de portée comme une variable locale à une fonction quand le code de la fonction a terminé de s'exécuter
- par l'usage de l'instruction **unset()**.

Il s'agit encore d'une méthode magique. Un destructeur :

- Se nomme toujours **__destruct()**

- Est appelé automatiquement à la suppression de l'objet
- Est toujours **public**
- Ne peut jamais recevoir de paramètre
- Ne doit jamais être appelé directement

Exemple :

```
class Etudiant_UR {
...
     function __destruct() {
        ...
     }
}
$lulu = new Etudiant_UR("L'ASTICOT", "LULU", 12345);
unset($lulu);   // Le destructeur sera appelé automatiquement
```

Exercice 3

Au lieu d'obliger un appel à **setNow()** par l'utilisateur de la classe, ajoutez plutôt un constructeur à votre classe **Horodateur**, qui va permettre l'initialisation de la variable **$maintenant** soit au temps Unix de l'instant présent (si pas de paramètre passé au constructeur), soit au temps Unix passé en paramètre au constructeur. Profitez-en pour supprimer cette méthode **setNow()** et adaptez votre code de l'exercice 2 en conséquence.

Exercice 4

Ajoutez un destructeur à votre classe **Horodateur** qui affiche **Au revoir <label>** où **<label>** est la valeur de l'attribut **label**. Testez votre programme dans les mêmes conditions que l'exercice 2.

Observez que le destructeur est bien appelé, pour chaque objet, quand le script se termine.

Observer l'ordre dans lequel s'affichent les deux destructions.

Créer un destructeur est une façon de pouvoir coder le nettoyage avant de quitter. Par exemple, si votre objet a créé un fichier temporaire, c'est un moyen pratique de pouvoir le supprimer automatiquement peu importent les raisons et à quel endroit du code l'objet est supprimé.

Cas particulier

Il a été dit qu'on n'appelle jamais un constructeur ou un destructeur dans son code.

Il existe cependant un cas exceptionnel avec les classes héritées.

Si on met en place un constructeur (et uniquement dans ce cas) dans une classe héritée on ~~peut~~ doit appeler le constructeur de la classe parente. Il ne faut rien supposer de ce que fait le constructeur parent, même si à un instant T il n'y a pas de constructeur

parent, ça peut évoluer dans le futur. Il est donc une bonne pratique que de systématiquement appeler le constructeur parent.

Il en va de même avec le destructeur.

Notez bien la syntaxe particulière de ces appels :

```
function __construct(...) {
     parent::construct(...);   // Passer les paramètres adaptés
     ...
}
function __destruct() {
     ...
     parent::destruct();
}
```

Getter & Setter

Choisir de définir un attribut en **public** permet d'y accéder (lecture et écriture) depuis l'extérieur de la classe.

Choisir de définir un attribut en **protected** ou **private** permet de le masquer s'il n'est pas utile de l'extérieur ou d'en protéger sa modification.

Cependant, il existe des situations qui ne sont satisfaites par aucun de ces deux modes. Quelques exemples :

- Pouvoir lire un attribut tout en protégeant sa modification
- Pouvoir affecter un attribut tout en contrôlant la valeur d'affectation
- Pouvoir affecter un attribut une seule fois ou un nombre limité de fois

Une solution est de protéger les attributs des accès externes en les définissant en **private**, et de proposer des méthodes d'accès en lecture et/ou en écriture, en fonction de ce qui est recherché.

Exemple :

```
class Etudiant_UR {
     ...
     private $numero;

     function setNumero($val) {
        $this->numero = $val;
     }
}
$etu = new Etudiant_UR();
$etu->numero = 12345;    // Ne fonctionne pas car private
$etu->setNumero("12345");// OK via la méthode setNumero()
echo $etu->numero;       // Ne fonctionne pas car private
```

PHP a encore une fois une solution à lui pour rendre l'écriture du code plus sympa : les Getters et les Setters.

L'idée est de conserver une écriture de code identique à ce qu'on écrirait si l'attribut était **public**, mais en passant par une tuyauterie interne à l'objet à base de méthodes comme le **setNumero(...)** précédent.

Ce mécanisme utilise les méthodes magiques **__get()** et **__set()**. qui permet de traiter tous les attributs qu'on souhaite dans ces deux seules méthodes.

Voici un exemple :

```
class Etudiant_UR {
    private $nom;
    private $prenom;
    private $numero;

    function __set($nom_att, $val) {
        if ($nom_att === 'nom') {
            $this->nom = strtoupper($val);          ①
        } else if ($nom_att === 'prenom') {
            $this->prenom = substr($val, 0, 10);    ②
        } else if ($nom_att === 'numero') {
            $this->numero = $val;
        }
    }
    function __get($nom_att) {
        if ($nom_att === 'nom') {
            return $this->nom;
        } else if ($nom_att === 'prenom') {
            return $this->prenom;
        }                                           ③
    }
}
$etu = new Etudiant_UR();
$etu->nom = "l'asticot";
$etu->prenom = "LULU PAULO DELLA CASA BIANCA";
$etu->numero = 98765;
echo "$etu->nom $etu->prenom\n";
echo $etu->numero;                                  ④
```

Dans ce code, les accès (lecture et modification) des attributs **nom**, **prenom** et **numero** se font par une écriture de code indépendante du fait de l'utilisation des Getters et des Setters, ce qui est agréable à coder et plus lisible.

Et les Getters et Setters apportent en plus que le **nom** sera toujours stocké en majuscules grâce à ① , que le **prenom** sera au maximum de 10 caractères grâce à ② , et que le **numero** pourra être affecté, mais grâce à ③ il ne pourra jamais être lu en ④.

À savoir

Voici maintenant quelques informations à connaître sur la POO “à la sauce PHP”, mais qui ne seront pas détaillées dans ce TP.

Copie vs Clonage

La copie d’un objet se fait par l’expression **`$obj2 = $obj1`**. Attention, c’est un terme abusif car ça ne copie pas réellement l’objet, ça crée juste une référence sur le même objet. Toute modification faite sur **`$obj2`** sera visible aussi sur **`$obj1`** puisque c’est le même objet.

Pour effectuer une réelle copie sous forme d’un autre objet indépendant, il faut utiliser l’expression **`$obj2 = clone $obj1`**. Les deux objets seront alors distincts mais attention, même si les valeurs des attributs sont dupliquées, le constructeur n’est pas appelé lors du clonage. Si le constructeur fait un traitement particulier à l’instanciation, il faudra prévoir un mécanisme pour faire ce même traitement après une copie par **`clone`** !

Surcharge

Dans une classe **`B`** héritée d’une classe **`A`**, il est possible de redéfinir une méthode qui existe déjà dans la classe parente **`A`**. En fonction de la nature de l’objet (est-il de la classe **`A`** ou de la classe **`B`** ?), la méthode appelée sera soit celle de la classe **`A`** soit celle de la classe **`B`**.

Si la méthode surchargée dans **`B`** veut appeler la méthode de la classe **`A`**, il faut préfixer l’appel par un **`parent::`**

Attributs et méthodes de classe

Comme avec Java et d’autres langages de POO, PHP propose aussi des attributs et des méthodes de classe qui sont attachés aux classes et non aux objets, et qui existent donc indépendamment des objets.

La déclaration se fait ainsi avec le mot-clé **`static`** :

```
class Math {
     static public $pi = 3.1415;
     static function carre($val) {
          return $val * $val;
     }
}

echo "PI vaut " . Math::$pi;
echo "5^2 vaut " . Math::carre(5);
```

Il n'a pas été nécessaire d'instancier des objets de la classe Math pour accéder à **`$pi`** et appeler la méthode **`carre()`**.

Notez la présence de **`Math::$`** et non pas **`$obj->`**. C'est une écriture similaire à celle déjà rencontrée précédemment avec **`parent::construct(...)`** et **`parent::destruct()`**.

Tout comme un objet qui veut accéder à lui-même doit utiliser **`$this`**, une classe qui veut faire référence à elle-même peut le faire avec **`self::`**.

Classe abstraite

Une classe abstraite est une classe dont la définition est incomplète et ne peut servir qu'à être héritée par une autre classe qui devra combler les "trous laissés dans la requête" de la classe parente.

Le mot-clé **`abstract`** permet d'indiquer qu'il s'agit d'une classe abstraite et quelles sont les méthodes à définir dans la classe héritée. Exemple :

```
abstract class Nombre {
     public $val1, $val2;
...
     abstract function division();
}

class Entier extends Nombre {
     function division() {
          return intval($this->val1) / intval($this->val2);
     }
}

class Reel extends Nombre {
     function division($val1, $val2) {
          return floatval($this->val1) / floatval($this->val2);
     }
}
```

Il est possible d'instancier les classes **`Entier`** et **`Reel`** mais pas **`Nombre`** qui est **`abstract`**.

Nous n'aborderons pas plus en détail ce type de classe dans ce TP.

Sérialisation

PHP ne sait pas sérialiser des objets nativement.

Cependant il est possible de sérialiser des objets PHP en lui donnant un petit coup de main via les méthodes magiques **`__sleep()`** et **`__wakeup()`**.

Ce TP n'abordera pas ces notions.

Structure

Les structures n'existent pas en PHP mais vous pouvez aisément détourner l'usage des classes pour combler ce manque. Avec la superclasse **`Object`** et le fait qu'on peut créer des attributs à la volée dans des objets, il est possible de simuler des objets-structure ainsi :

```
$une_personne = new Object();
$une_personne->adresse = "12, rue Victor Hugo, 22300 Lannion";
$une_personne->tel = "0605029187";
$une_personne->email = "lulu@pipo.com";
```

Mise en application

Vous trouverez sur Moodle un fichier **`EQUIPES_TDF_2023`** contenant une liste d'équipes participant au Tour de France 2023.

Vous trouverez aussi des fichiers portant les noms des équipes et contenant la liste des coureurs de chaque équipe. Notez que les fichiers portent les noms de chaque équipe trouvée dans **`EQUIPES_TDF_2023`** avec un **`_`** à la place des espaces. Exemple : l'équipe **`GROUPAMA - FDJ`** trouvée dans la liste correspond au fichier **`GROUPAMA_-_FDJ`**.

Les fichiers liste d'équipe contiennent tous une 1ère ligne qui est le nom de l'équipe, une information que vous avez déjà dans la liste. À vous de ne pas traiter cette 1ère ligne.

Objectif

L'objectif est d'afficher, sur une seule page HTML/PHP, une liste des équipes sous la forme présentée page suivante.

Votre code sera constitué du chargement des fichiers, de la création des objets et de la construction de la table à partir des objets créés. Le tout séquentiellement dans le même script.

Exercice 5

Votre script PHP doit :

- Créer une classe **`Coureur`** permettant de stocker les informations d'un coureur
- Créer une classe **`Equipe`** permettant de stocker les informations d'une équipe, notamment une liste de coureurs sous la forme d'un tableau associatif (clé = numéro du coureur, valeur = objet **`Coureur`**)
- Créer un tableau global, associant chaque numéro d'équipe à un objet **`Equipe`**.
- Lire la liste des équipes dans le fichier **`EQUIPES_TDF_2023`**
- Remplir le tableau d'équipes sans s'occuper des coureurs qui les composent. C'est l'étape ci-dessous qui va s'en charger dans un second temps.

- Une fois le tableau d'équipes créé et rempli, peupler le tableau des coureurs de chaque objet **`Equipe`** avec les coureurs lus dans le fichier idoine.
  Pour ce faire, vous devez impérativement utiliser une méthode nommée **`remplirListeCoureurs()`** dans la classe **`Equipe`**. C'est cette méthode qui s'occupe de la lecture du bon fichier et de la création de tous les objets **`Coureurs`** nécessaires.
- Coder l'affichage souhaité, une fois les données chargées dans les objets.

Extrait de l'affichage à respecter

JUMBO-VISMA		
1	1	JONAS VINGEGAARD
	2	TIESJ BENOOT
	3	WILCO KELDERMAN
	4	SEPP KUSS
	5	CHRISTOPHE LAPORTE
	6	WOUT VAN AERT
	7	DYLAN VAN BAARLE
	8	NATHAN VAN HOOYDONCK
UAE TEAM EMIRATES		
2	11	TADEJ POGAČAR
	12	MIKKEL BJERG
	13	FELIX GROSSSCHARTNER
	...	...

Exercice 6

Adaptez vos classes pour gérer l'abandon d'un coureur et le forfait d'une équipe.

Vous devez prévoir un attribut **`abandon`** avec une valeur par défaut dans la classe **`Coureur`** ainsi qu'une méthode **`abandonner()`** permettant de matérialiser l'abandon d'un coureur à partir de son numéro de coureur. Seule la fonction **`abandonner()`** doit pouvoir modifier cet attribut.

Vous devez prévoir une méthode **`forfait()`** permettant de matérialiser le forfait de l'équipe en provoquant l'abandon de tous ses coureurs.

Pour tester, vous provoquerez l'abandon d'un coureur et le forfait d'une équipe, dans votre code, après la création des objets et avant l'affichage de la table.

Adaptez votre table HTML en colorant le nom d'un coureur en rouge s'il a abandonné.

Exercice 7

Au lieu de stocker les équipes dans un tableau global à votre script, placez ce tableau comme variable de classe de la classe Equipes. Adaptez votre code en conséquence.

Exercice 8

Ajoutez une méthode de classe à la classe **`Equipe`**, nommée **`trouveEquipe($nom)`**, qui retourne un objet **`Equipe`** à partir de son nom, ou **`FALSE`** si pas trouvé.

Testez la recherche avec deux équipes : **`COFIDIS`** et **`BILOUTE`**, et faites un **`var_dump()`** de chaque résultat.

Exercice 9

Ajoutez une méthode de classe à la classe **`Equipe`**, nommée **`trouveDossard($num)`**, qui retourne un tableau de deux objets : une **`Equipe`** et un **`Coureur`** à partir de son numéro de dossard, ou **`FALSE`** si pas trouvé.

Testez la recherche avec deux dossards : **`25`** et **`3212`**, et faites un **`var_dump()`** de chaque résultat.

Adaptez ensuite votre code pour créer l'abandon d'un joueur par son numéro de dossard et le forfait d'une équipe par son nom.