Programmation orientée objet en PHP*

Partie 1

[R3.01 : Développement web]

Nous allons suivre un nouveau projet fil rouge nommé « MatchMaker » tout au long de ce TP (et des 2 prochains), qui crée des rencontres compétitives basées sur le niveau des joueurs.

1 Préparez votre environnement de travail

Gestion du code avec Git et GitHub

Initialisation d'un dépôt local Git

Vous allez travailler avec Git. Pour ce faire, vous devez créer un dépôt local, c'est à dire un dossier dans lequel toutes vos modifications seront enregistrées.

1. Créez un dossier nommé `poo-php` dans le répertoire `~\UwAmp\www`.
2. Placez-vous dans ce dossier `poo-php` nouvellement créé et ouvrez une invite de commande Git Bash ou Windows PowerShell.
3. Lancer la commande `git init`. Elle initialise ce dossier `poo-php` comme un dépôt.

Un dossier caché `.git` a été créé. Il contient tous les éléments non visibles de Git : la configuration, les logs, les branches...

4. Extraire le contenu du fichier `poo-php-p1.zip` directement à la racine de votre dossier `poo-php`.
5. Passez la commande `git add -all` pour indexer le tout nouveau contenu du dépôt.
6. Utilisez la commande `git commit -m "POO en PHP : 1er commit"` pour créer une première version.

L'argument `-m` permet de définir un message particulier rattaché au commit effectué. Si vous n'utilisez pas cet argument, la commande `git commit` vous demandera de saisir le message de commit.

*`https://openclassrooms.com`

Le screencast `git-init.asf` reprend pas à pas tout ça.

Création d'un dépôt distant GitHub

Pour mettre votre projet sur GitHub, vous devez d'abord créer un repository (nommez-le aussi poo-php) dans lequel il pourra être installé.

Puis vous allez devoir relier votre dépôt local au dépôt distant que vous venez de créer sur GitHub.

Pour cela, copiez le lien https (de la forme `https://github.com/alexbrabant/poo-php.git`) qui se trouve en haut sur fond bleu dans l'onglet Code et collez-le dans la commande `git remote add origin https://github.com/alexbrabant/poo-php.git`.

Ensuite tapez la commande `git branch -M main`.

Vous avez relié le dépôt local au dépôt distant. Vous pouvez donc envoyer des commits du repository vers le dépôt distant GitHub en utilisant la commande `git push -u origin main`

Le screencast `git-remote.asf` reprend pas à pas tout ça.

Enfin, n'oubliez pas de m'inviter en tant que collaborateur (alexbrabant) ;-)

2 Classes natives de PHP

Le langage PHP propose déjà plusieurs classes par défaut, comme `DateTime`. Pour utiliser cette classe, vous devez l'appeler par son nom, précédé du mot clé `new` :

```php
<?php

$date = new DateTime;
```

Le nom de la classe, `DateTime`, est écrit en *PascalCase*, qui équivaut au *UpperCamelCase*. C'est une bonne pratique qui fait partie des PSR (PHP Standards Recommendations), en particulier PSR-1 et PSR-12. Les propriétés et les méthodes doivent être écrites au format camelCase.

Les objets se comparent de manière un peu différente des autres types de variables. Si vous créez 2 objets issus de la même classe, que vous leur assignez les mêmes valeurs et les comparez simplement (==), vous vérifiez que les 2 éléments possèdent les mêmes propriétés et les mêmes **valeurs**. Si vous testez strictement (===), alors vous vérifiez que vous êtes en train de manipuler la même **instance**.

Pour rappel, les objets possèdent des propriétés (variables) et des méthodes (fonctions). Pour y accéder, il faut utiliser le symbole `->` à la suite de la variable, comme dans l'exemple suivant :

```php
<?php

$date = new DateTime;
echo $date->format('d/m/Y');
```

Pour savoir comment fonctionnent ces classes natives de PHP, il faut aller lire la documentation qui se trouve sur PHP.net (la version anglaise apporte plus de précision et d ?exemples que la version française).

3 Créez vos propres classes

Une classe se déclare de la manière suivante :

```php
<?php

class NomDeLaClasse
{
}
```

Pour instancier une classe, rien ne change.

```php
<?php

class Pont
{
}

$pont = new Pont;
```

Ajoutons une propriété `$longueur` :

```php
<?php

declare(strict_types=1);

class Pont
{
   public float $longueur = 0;
}

$pont = new Pont;
$pont->longueur = 263.0;

var_dump($pont);
```

En ligne 3 se trouve une instruction demandant à PHP d'être exigeant avec le typage. Le mot clé `declare` permet d'indiquer à PHP un comportement spécifique pour le fichier dans lequel nous nous trouvons. Si vous êtes curieux, d'autres instructions existent dans la documentation PHP.

Ajoutons une méthode `getSurface()` :

```php
<?php

declare(strict_types=1);

class Pont
{
   public float $longueur;
   public float $largeur;

   public function getSurface(): float
   {
       return $this->longueur * $this->largeur;
   }
}
```

Appelons maintenant cette méthode pour obtenir la surface :

```php
<?php
// ...

$pont = new Pont;
$pont->longueur = 286.0;
$pont->largeur = 15.0;

$surface = $pont->getSurface();

var_dump($surface);
```

Travail à faire

Vous trouverez dans le fichier `index.php` décompressé dans votre dépôt git local un squelette de code simple, écrit de manière fonctionnelle qui pourrait être le système de classement. Vous modifierez ce code en créant :

- une classe `Player` ayant uniquement une propriété `public int $level` ;
- une classe `Encounter` (rencontre en français) sans propriété mais avec deux méthodes publiques qui sont les deux fonctions existantes dans le code fourni.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Créez vos propres classes"
> git push -u origin main
```

4 Encapsulez les données de vos objets

On part du principe que vous souhaitez toujours avoir un maximum de contrôle sur ce que contiennent les propriétés. Avec le typage strict, vous êtes garanti d'avoir un flottant en longueur et en largeur. Mais rien ne vous empêche d'y insérer des valeurs négatives, par exemple. Pour se protéger, il faut privatiser `$longueur` et `$largeur` et ajouter un setter (ou mutateur) en français et un getter (ou accesseur en français) :

```php
<?php

declare(strict_types=1);

class Pont
{
   private float $longueur;
   private float $largeur;

   public function setLongueur(float $longueur): void
   {
     if ($longueur < 0) {
        trigger_error('La longueur est trop courte. (min 1)', E_USER_ERROR);
     }
     $this->longueur = $longueur;
    }

    public function getLongueur(): float
    {
      return $this->longueur;
    }

    public function setLargeur(float $largeur): void
    {
      if ($largeur < 0) {
        trigger_error('La largeur est trop courte. (min 1)', E_USER_ERROR);
      }
      $this->largeur = $largeur;
    }
```

```
    public function getLargeur(): float
    {
      return $this->largeur;
    }

    public function getSurface(): float
    {
      return $this->longueur * $this->largeur;
    }
}
```

L'intérêt d'utiliser les getters et les setters est qu'il est toujours possible de récupérer la longueur, de la modifier mais en s'assurant d'effectuer certains contrôles avant (définis dans le setter).

5 Utilisez les propriétés et méthodes statiques

Rappels

Les méthodes et propriétés **statiques** se rapportent à une classe (donc communes à tous nos objets).

Une méthode sans instance, doit être déclarée statique et faire attention à manipuler uniquement des propriétés statiques.

Lorsqu'une propriété est déclarée statique, la valeur qu'elle contient sera partagée pour toutes les instances.

Exemple

Déclarons une méthode statique `validerTaille()` avec le mot clé `static` :

```
<?php

declare(strict_types=1);

class Pont
{
   public static function validerTaille(float $taille): bool
   {
       if ($taille < 50.0) {
           trigger_error(
               'La taille est trop courte. (min 50m)',
               E_USER_ERROR
           );
       }

       return true;
```

```
    }

    public function setLongueur(float $longueur): void
    {
        self::validerTaille($longueur);

        $this->longueur = $longueur;
    }
}

var_dump(Pont::validerTaille(150.0));
var_dump(Pont::validerTaille(20.0));

$towerBridge = new Pont;
$towerBridge->setLongueur(20.0);
```

Vous remarquez deux choses dans le code ci-dessus :

- Pour faire référence à un élément de la classe, il faut utiliser `::` à la place de la flèche `->`.
- Vous devez utiliser le mot clé `self` à la place de `this` pour cibler une méthode statique de classe, lorsque vous l'appelez depuis une instance de cette même classe.

Remarque

Parfois, vous pouvez avoir besoin d'une propriété statique immuable, c'est-à-dire qui ne changera jamais.
Nous pouvons alors passer cette propriété en constante avec le mot-clé `const` :

Exemple

```
<?php

declare(strict_types=1);

class Pont
{
    private const UNITE = 'm^2';
    private float $longueur;
    private float $largeur;

    public function getSurface(): string
    {
        return ($this->longueur * $this->largeur) . self::UNITE;
    }
   // setLongueur et setLargeur ne changent pas
}
```

```php
$towerBridge = new Pont;
$towerBridge->setLongueur(286.0);
$towerBridge->setLargeur(15.0);

echo $towerBridge->getSurface();
```

On constate dans le code ci-dessus que :

- Une constante de classe s'écrit avec le mot-clé `const` est une valeur lui est donnée directement.
- Les constantes ne sont pas préfixées par `$` et par convention sont écrites en `UPPER_SNAKE_CASE` (tout en majuscule, avec les mots séparés par des underscores)
- Une constante est statique, on utilise le mot-clé `self` pour l'utiliser dans `getSurface`.

Travail à faire

Transformez les méthodes de la classe `Encounter` pour utiliser le mot clé `static` et intégrez les viariables globales immuables existantes en constantes de la classe `Encounter`.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Utilisez les propriétés et méthodes statiques"
> git push -u origin main
```

6 Découvrez les méthodes particulières

L'usage du mot clé `class` vous donne accès, dès le départ, à un jeu de méthodes prédéfinies. Elles sont reconnaissables parce qu'elles sont préfixées par deux **underscores**. Il en existe plusieurs, que vous pouvez regarder dans la documentation PHP. Ici, nous nous intéressons tout d'abord à deux d'entre elles : texttt`__`construct (le constructeur) et `__destruct` (le destructeur).

Exemple

```php
<?php

declare(strict_types=1);

class Pont
{
    private float $longueur;
    private float $largeur;
```

```php
    public function __construct(float $longueur, float $largeur)
    {
        $this->longueur = $longueur;
        $this->largeur = $largeur;
    }
}

$towerBridge = new Pont(286.0, 62.0);
```

Depuis PHP8, vous pouvez aussi rencontrer le constructeur sous sa forme courte. En précisant la visibilité directement au niveau des arguments du constructeur, vous pouvez **ne pas déclarer les propriétés et leur assignation**. Ce sera fait automatiquement par PHP :

```php
<?php

declare(strict_types=1);

class Pont
{
    public function __construct(private float $longueur, private float $largeur)
    {
    }
}

$towerBridge = new Pont(286.0, 62.0);
var_dump($towerBridge);
```

La méthode `__destruct` est appelée automatiquement lorsque l'objet est supprimé de la mémoire, ce qui est fait à chaque fois que le script se termine. Il existe deux autres moyens de le déclencher manuellement : en supprimant l'objet avec `unset` ou en remplaçant le contenu de la variable qui y fait référence. Pour faire simple, dès que PHP détecte que plus rien ne fait référence à un objet en mémoire, il le détruit, et donc déclenche `__destruct`.

Pour ne pas rendre ce cours trop long, je n'aborderai pas les autres méthodes avec vous. Je vous invite donc à aller les découvrir sur la documentation PHP.

Travail à faire

Implémentez un constructeur, un accesseur et un mutateur dans la classe `Player`.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Découvrez les méthodes particulières"
```

```
> git push -u origin main
```