TP 2 – Introduction à la POO en PHP

Surcharge, classe abstraite, interface, méthodes magiques

Reprenons le contexte du TP 1.

Imaginons qu'on possède un site sur lequel les visiteurs peuvent s'enregistrer pour avoir accès à un espace personnel par exemple. Quand un visiteur s'enregistre pour la première fois, il devient un utilisateur du site.

A chaque fois qu'un visiteur s'inscrit et devient utilisateur, on va donc devoir créer des variables « nom d'utilisateur », « mot de passe », etc. et définir leurs valeurs et donner les permissions à l'utilisateur d'utiliser les fonctions connexion, déconnexion, etc.

Pour cela, on va créer un formulaire d'inscription sur notre site qui va demander un nom d'utilisateur et un mot de passe, etc. On va également définir les actions (fonctions) propres à nos utilisateurs : connexion, déconnexion, possibilité de commenter, etc.

Surcharge

Ajoutons immédiatement une constante à notre classe mère Utilisateur : on crée une constante ABONNEMENT qui stocke la valeur « 15 ». On peut imaginer que cette constante représente le prix d'un abonnement mensuel pour accéder au contenu de notre site.

```php
<?php
    class Utilisateur{
        protected $user_name;
        protected $user_region;
        protected $prix_abo;
        protected $user_pass;

        /*Attention: si vous utilisez une version PHP < PHP 7.1, ce code ne
         *fonctionnera pas*/
        public const ABONNEMENT = 15;

        public function __construct($n, $p, $r){
            $this->user_name = $n;
            $this->user_pass = $p;
            $this->user_region = $r;
        }

        public function __destruct(){
            //Du code à exécuter
        }

        public function getNom(){
            echo $this->user_name;
        }

        public function setPrixAbo(){
            /*On peut imaginer qu'on calcule un prix d'abonnement différent
             *selon les profils des utilisateurs*/
            if($this->user_region === 'Sud'){
                return $this->prix_abo = self::ABONNEMENT / 2;
            }else{
                return $this->prix_abo = self::ABONNEMENT;
            }
        }

        public function getPrixAbo(){
            echo $this->prix_abo;
        }
    }
?>
```

Figure 1: classe Utilisateur

- L'idée ici va être de définir un tarif d'abonnement différent en fonction des différents profils des utilisateurs et en se basant sur notre constante de classe ABONNEMENT.
- On va vouloir définir un tarif préférentiel pour les utilisateurs qui viennent du Sud (car c'est mon site et je fais ce que je veux !). On va ici rajouter deux propriétés dans notre classe : **$user_region** qui va contenir la région de l'utilisateur et **$prix_abo** qui va contenir le prix de l'abonnement après calcul.
- On crée ensuite les méthodes **setPrixAbo()** et **getPrixAbo()**
- La façon d'accéder à une constante va légèrement varier selon qu'on essaie d'y accéder depuis l'intérieur de la classe qui la définit (ou d'une classe étendue), on utilise le mot clé **self** ou depuis l'extérieur de la classe (**parent**).

On va maintenant se rendre dans notre classe étendue Admin :

- On surcharge ici la constante ABONNEMENT de la classe parente en lui attribuant une nouvelle valeur.
- Pour les objets de la classe Admin, on définit le prix de l'abonnement de base à 5. Dans notre classe, on modifie le constructeur pour y intégrer un paramètre « région » et on supprime également la méthode getNom() créée précédemment.
- Enfin, on surcharge la méthode setPrixAbo() : si l'administrateur indique venir du Sud, le prix de son abonnement va être égal à la valeur de la constante ABONNEMENT définie dans la classe courante (c'est-à-dire dans la classe Admin). Ici, self et parent nous servent à indiquer à quelle définition de la constante ABONNEMENT on souhaite se référer.

```php
<?php
    class Admin extends Utilisateur{
        protected $ban;
        public const ABONNEMENT = 5;

        public function __construct($n, $p, $r){
            $this->user_name = strtoupper($n);
            $this->user_pass = $p;
            $this->user_region = $r;
        }

        public function setBan($b){
            $this->ban[] .= $b;
        }
        public function getBan(){
            echo 'Utilisateurs bannis par '.$this->user_name. ' : ';
            foreach($this->ban as $valeur){
                echo $valeur .', ';
            }
        }

        public function setPrixAbo(){
            if($this->user_region === 'Sud'){
                return $this->prix_abo = self::ABONNEMENT;
            }else{
                return $this->prix_abo = parent::ABONNEMENT / 2;
            }
        }
    }
?>
```

Figure 2: classe Admin

On teste dans la page principale :

Dans le code l'écriture $u::ABONNEMENT est donc strictement équivalente à UTILISATEUR::ABONNEMENT.

```php
<body>
  <h1>Titre principal</h1>
  <?php
    require 'classes/utilisateur.class.php';
    require 'classes/admin.class.php';

    $pierre = new Admin('Pierre', 'abcdef', 'Sud');
    $mathilde = new Admin('Math', 123456, 'Nord');
    $florian = new Utilisateur('Flo', 'flotri', 'Est');

    $pierre->setPrixAbo();
    $mathilde->setPrixAbo();
    $florian->setPrixAbo();

    $u = 'Utilisateur';
    echo 'Valeur de ABONNEMENT dans Utilisateur : ' .$u::ABONNEMENT. '<br>';
    echo 'Valeur de ABONNEMENT dans Admin : ' .Admin::ABONNEMENT. '<br>';

    echo 'Prix de l\'abonnement pour ';
    $pierre->getNom();
    echo ' : ';
    $pierre->getPrixAbo();
    echo '<br>Prix de l\'abonnement pour ';
    $mathilde->getNom();
    echo ' : ';
    $mathilde->getPrixAbo();
    echo '<br>Prix de l\'abonnement pour ';
    $florian->getNom();
    echo ' : ';
    $florian->getPrixAbo();
  ?>
  <p>Un paragraphe</p>
</body>
```

Figure 3: Page principale- Test Surcharge

Classes abstraites

Normalement, si notre code est bien construit, on devrait voir une hiérarchie claire entre ce que représentent nos classes mères et nos classes enfants. Dans le cas présent, j'aimerais que ma classe mère définisse des choses pour TOUS les types d'utilisateurs et que les classes étendues s'occupent chacune de définir des spécificités pour UN type d'utilisateur en particulier. Encore une fois, ici, on touche à des notions qui sont plus de design de conception que des notions de code en soi mais lorsqu'on code la façon dont on crée et organise le code est au moins aussi importante que le code en soi. Il faut donc toujours essayer d'avoir la structure globale la plus claire et la plus pertinente possible.

Nous allons donc partir du principe que nous avons deux grands types d'utilisateurs : les utilisateurs classiques et les administrateurs. On va donc transformer notre classe Utilisateur afin qu'elle ne définisse que les choses communes à tous les utilisateurs et allons définir les spécificités de chaque type utilisateur dans des classes étendues Admin et Abonne.

- On commence déjà par modifier notre classe parent **Utilisateur** en la définissant comme abstraite. On supprime le constructeur qui va être défini dans les classes étendues et on déclare également la méthode **setPrixAbo()** comme abstraite afin de forcer ainsi les classes étendues à l'implémenter.
- Maintenant qu'on a défini notre classe Utilisateur comme abstraite, il va falloir l'étendre et également implémenter les méthodes abstraites. On va commencer par aller dans notre classe étendue **Admin** et supprimer la constante ABONNEMENT puisque nous allons désormais utiliser celle de la classe abstraite. On va donc également modifier le code de notre méthode **setPrixAbo().**

```php
<?php
    abstract class Utilisateur{
        protected $user_name;
        protected $user_region;
        protected $prix_abo;
        protected $user_pass;
        public const ABONNEMENT = 15;

        public function __destruct(){
            //Du code à exécuter
        }

        abstract public function setPrixAbo();

        public function getNom(){
            echo $this->user_name;
        }

        public function getPrixAbo(){
            echo $this->prix_abo;
        }
    }
?>
```

Figure 4: classe abstraite Utilisateur

```php
<?php
    class Admin extends Utilisateur{
        protected static $ban;

        public function __construct($n, $p, $r){
            $this->user_name = strtoupper($n);
            $this->user_pass = $p;
            $this->user_region = $r;
        }

        public function setBan(...$b){
            foreach($b as $banned){
                self::$ban[] .= $banned;
            }
        }
        public function getBan(){
            echo 'Utilisateurs bannis : ';
            foreach(self::$ban as $valeur){
                echo $valeur .', ';
            }
        }

        public function setPrixAbo(){
            if($this->user_region === 'Sud'){
                return $this->prix_abo = parent::ABONNEMENT / 6;
            }else{
                return $this->prix_abo = parent::ABONNEMENT / 3;
            }
        }
    }
?>
```

Figure 5: classe étendue Admin

On va ensuite créer un nouveau fichier de classe qu'on va appeler **abonne.class.php**. Dans ce fichier, nous allons définir un constructeur pour nos abonnés qui représentent nos utilisateurs de base et allons à nouveau implémenter la méthode **setPrixAbo().**

```php
<?php
    class Abonne extends Utilisateur{
        public function __construct($n, $p, $r){
            $this->user_name = $n;
            $this->user_pass = $p;
            $this->user_region = $r;
        }

        public function setPrixAbo(){
            if($this->user_region === 'Sud'){
                return $this->prix_abo = parent::ABONNEMENT / 2;
            }else{
                return $this->prix_abo = parent::ABONNEMENT;
            }
        }
    }
?>
```

Figure 6: classe étendue Abonné

On peut maintenant retourner sur notre script principal et créer des objets à partir de nos classes étendues et voir comment ils se comportent :

```php
<body>
    <h1>Titre principal</h1>
    <?php
        require 'classes/utilisateur.class.php';
        require 'classes/admin.class.php';
        require 'classes/abonne.class.php';

        $pierre = new Admin('Pierre', 'abcdef', 'Sud');
        $mathilde = new Admin('Math', 123456, 'Nord');
        $florian = new Abonne('Flo', 'flotri', 'Est');

        $pierre->setPrixAbo();
        $mathilde->setPrixAbo();
        $florian->setPrixAbo();

        echo 'Prix de l\'abonnement pour ';
        $pierre->getNom();
        echo ' : ';
        $pierre->getPrixAbo();
        echo '<br>Prix de l\'abonnement pour ';
        $mathilde->getNom();
        echo ' : ';
        $mathilde->getPrixAbo();
        echo '<br>Prix de l\'abonnement pour ';
        $florian->getNom();
        echo ' : ';
        $florian->getPrixAbo();
    ?>
    <p>Un paragraphe</p>
</body>
```

Figure 7: Page principale- Test classe abstraite

Interfaces

On va pouvoir définir une interface de la même manière qu'une classe mais en utilisant cette fois-ci le mot clef interface à la place de class. Nous nommerons généralement nos fichiers d'interface en utilisant « **interface** » à la place de « classe ». Par exemple, si on crée une interface nommée **Utilisateur**, on enregistrera le fichier d'interface sous le nom **utiilsateur.interface.php** par convention.

On va ensuite pouvoir réutiliser les définitions de notre interface dans des classes. Pour cela, on va implémenter notre interface. On va pouvoir faire cela de la même manière que lors de la création de classes étendues mais on va cette fois-ci utiliser le mot clef **implements** à la place de **extends.**

```php
<?php
    interface Utilisateur{
        public const ABONNEMENT = 15;
        public function getNom();
        public function setPrixAbo();
        public function getPrixAbo();
    }
?>
```

Figure 8: interface Utilisateur

```php
<?php
    class Abonne implements Utilisateur{
        protected $user_name;
        protected $user_region;
        protected $prix_abo;
        private $user_pass;

        public function __construct($n, $p, $r){
            $this->user_name = $n;
            $this->user_pass = $p;
            $this->user_region = $r;
        }

        public function getNom(){
            echo $this->user_name;
        }

        public function getPrixAbo(){
            echo $this->prix_abo;
        }

        public function setPrixAbo(){
            if($this->user_region === 'Sud'){
                return $this->prix_abo = Utilisateur::ABONNEMENT / 2;
            }else{
                return $this->prix_abo = Utilisateur::ABONNEMENT;
            }
        }
    }
?>
```

Figure 9: Abonné implémente Utilisateur

Ensuite, on utilise une interface Utilisateur plutôt que notre **classe abstraite Utilisateur** qu'on laisse de côté pour le moment. Notez bien ici que toutes les méthodes déclarées dans une interface doivent obligatoirement être implémentées dans une classe qui implémente une interface. Vous pouvez également observer que pour accéder à une constante d'interface, il va falloir préciser le nom de l'interface devant l'opérateur de résolution de portée.

```php
<?php
    class Admin implements Utilisateur{
        protected $user_name;
        protected $user_region;
        protected $prix_abo;
        private $user_pass;
        protected static $ban;

        public function __construct($n, $p, $r){
            $this->user_name = strtoupper($n);
            $this->user_pass = $p;
            $this->user_region = $r;
        }

        public function getNom(){
            echo $this->user_name;
        }

        public function getPrixAbo(){
            echo $this->prix_abo;
        }

        public function setBan(...$b){
            foreach($b as $banned){
                self::$ban[] .= $banned;
            }
        }
        public function getBan(){
            echo 'Utilisateurs bannis : ';
            foreach(self::$ban as $valeur){
                echo $valeur .', ';
            }
        }

        public function setPrixAbo(){
            if($this->user_region === 'Sud'){
                return $this->prix_abo = Utilisateur::ABONNEMENT / 6;
            }else{
                return $this->prix_abo = Utilisateur::ABONNEMENT / 3;
            }
        }
    }
?>
```

Figure 10: Admin implémente Utilisateur

```php
<body>
    <h1>Titre principal</h1>
    <?php
        require 'classes/utilisateur.interface.php';
        require 'classes/admin.class.php';
        require 'classes/abonne.class.php';

        $pierre = new Admin('Pierre', 'abcdef', 'Sud');
        $mathilde = new Admin('Math', 123456, 'Nord');
        $florian = new Abonne('Flo', 'flotri', 'Est');

        $pierre->setPrixAbo();
        $mathilde->setPrixAbo();
        $florian->setPrixAbo();

        echo 'Prix de l\'abonnement pour ';
        $pierre->getNom();
        echo ' : ';
        $pierre->getPrixAbo();
        echo '<br>Prix de l\'abonnement pour ';
        $mathilde->getNom();
        echo ' : ';
        $mathilde->getPrixAbo();
        echo '<br>Prix de l\'abonnement pour ';
        $florian->getNom();
        echo ' : ';
        $florian->getPrixAbo();
    ?>
    <p>Un paragraphe</p>
</body>
```

Figure 11: Page principale- Test Interface

A vous de continuer !

1. Interface ou classe abstraite : quel serait le choix le plus approprié pour cet exercice ?
2. Modifiez le formulaire que vous aviez précédemment créé et y rajouter les données manquantes.
3. Utiliser la méthode magique __toString() pour afficher un récapitulatif des informations d'un utilisateur.
4. Créez une liste d'utilisateurs, afin que lorsque l'on rajoute un nouvel utilisateur via le formulaire, celui-ci est rajouté à la liste
5. Enfin affichez la liste des utilisateurs et leurs informations.