

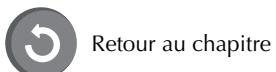
# L'héritage : gérons un nouveau type de contenu (POO avancée)

# Sommaire

<b>I. Mise en œuvre de l'héritage</b> .....	4
<b>II. Nouvelle architecture</b> .....	6
A. La classe générique Post .....	7
B. Un objet Post pour les méthodes <i>addPost</i> et <i>updatePost</i> .....	9
C. La classe <i>Breve</i> .....	10
D. Une solution : la surcharge .....	10
E. Alternatives non recommandées .....	11
F. La classe Filet .....	11
<b>III. Une solution alternative et l'opérateur de résolution de portée</b> .....	12
A. Une dernière modification sur la classe <i>postManager</i> .....	13
B. Aide au choix de conception .....	14
<b>IV. Les constantes de classe</b> .....	15
<b>V. Contrôle lors de la création des objets</b> .....	17

Crédits des illustrations:  
© DR.

## Les repères de lecture



Retour au chapitre



Définition



Objectif(s)



Espace Élèves



Vidéo /  
Audio



Point important /  
À retenir



Remarque(s)



Pour aller  
plus loin



Normes et lois



Quiz

Nous avons travaillé avec les brèves mais dans le monde de la presse d'autres types de contenus sont utilisés comme par exemple les filets ou les synthèses. Un filet est un contenu semblable à une brève mais qui s'en différencie par le fait qu'il possède un titre.

Considérons le besoin d'ajouter à notre application la gestion de filets. La première action à entreprendre serait d'ajouter une colonne *title* à notre table, puis de renommer celle-ci afin que sémantiquement elle soit définie de manière générique comme pouvant stocker soit des brèves, soit des filets (et à terme d'autres types de contenu).

Étant donné le caractère de publication des contenus nous pouvons renommer la table *breve* en *post*. Nous utilisons pour ce faire et à titre d'entraînement la console MySQL en validant la requête suivante :

```
RENAME TABLE breve TO post;
```

Fig. 1

Page du manuel MySQL : <http://dev.mysql.com/doc/refman/5.7/en/ rename-table.html>

Puis nous allons ajouter une colonne *title* dont les caractéristiques seront les suivantes :

- **type** : **VARCHAR** (255) pour insérer des titres éventuellement longs;
- **NULL** : oui de façon à ce que nous puissions enregistrer aussi bien des filets (avec titre) que des brèves (sans titre). Le fait que la valeur title soit **NULL** ou non sera d'ailleurs un moyen pratique de distinguer une brève d'un filet.

La requête MySQL d'ajout d'une colonne dans une table existante repose sur la commande ALTER TABLE suivie de ADD COLUMN.

Page du manuel MySQL : <https://dev.mysql.com/doc/refman/5.7/en/alter-table.html>

```
ALTER TABLE post ADD COLUMN title VARCHAR(255) NULL;
```

Fig.2

Et nous obtenons ainsi la nouvelle structure suivante :

```
mysql> DESCRIBE post;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
text	text	NO		NULL	
dt_creation	date	NO		NULL	
title	varchar(255)	YES		NULL	

Fig.3

Les informations stockées sont correctement restituées y compris les titres dont la valeur est NULL.

```
mysql> SELECT * FROM post;
```

id	text	dt_creation	title
1	Texte pour le contenu	2013-10-14	NULL
2	Nouveau contenu de la brève	2013-10-14	NULL

Fig.4

Suite à cette modification, il convient ensuite de modifier en conséquence nos deux classes.

Dans un premier temps, il semble pertinent de renommer les classes *Breve* et *breveManager* en *Post* et *postManager* puis de remplacer d'une façon similaire les noms des méthodes et le nom de la table qui sont utilisés.

Il ne resterait plus qu'à ajouter un *getter* et un *setter* dans la classe *Post* pour gérer un nouvel attribut *title* puis à ajouter dans les requêtes les éléments permettant d'intégrer le titre des filets.

Cependant, il convient de s'arrêter un instant et de réfléchir sur l'action à entreprendre concrètement.

## I. Mise en œuvre de l'héritage

Nous travaillons dans un contexte POO sur des objets *breves* et souhaitons ensuite travailler sur des objets *filets*. Ces objets sont certes similaires mais ce sont des objets différents, notamment parce que l'un possède un titre et l'autre non. À l'instar d'une moto et d'un vélo qui sont deux objets ayant chacun deux roues et sont guidés par une personne mais l'un possède un moteur l'autre non.

D'un point de vue conception informatique, une brève et un filet sont deux types d'objets distincts. Et comme nous l'avons vu en POO, une classe n'a qu'un seul rôle qui consiste à pouvoir créer un type d'objet donné.

Au vu de ces explications, il est donc judicieux de créer une classe *Filet* qui servirait à créer des objets *filets* et une classe *Breve* qui servirait à créer des objets *breves*.

Ces deux types d'objets ont des propriétés communes : un texte et une date de création. Ces propriétés sont au final semblables à tout contenu publié (que nous avons appelé : *post*).

Il se trouve qu'un des concepts clé de la POO est justement l'héritage qui permet à un objet de bénéficier des attributs et des méthodes d'un autre objet. Une classe héritant d'une autre classe est appelée la classe dérivée ou la classe fille, tandis que l'autre classe est appelée la classe générique ou la classe mère.



Les **filets** sont donc un format de *post*, et les **brèves** sont également un format de post et en ce sens brèves et filet héritent des *propriétés des posts*.

Notre nouvelle structure contiendra donc une classe mère (*Post*) et des classes filles (*Breve* et *Filet*) qui vont hériter des attributs et méthodes de la classe mère.

La syntaxe est la suivante, basée sur l'utilisation de la commande **extends**, la classe fille étant littéralement une extension de la classe mère.

```
class Post {  
    // attributs et méthodes  
}  
class Filet extends Post {  
}  
class Breve extends Post {  
}
```

Fig.5

Avant de reprendre notre projet, testons nos trois classes avec un exemple sur un nouveau fichier:

```
class Post  
{  
    public function display($a){  
        return $a;  
    }  
}  
class Breve extends Post {  
}  
class Filet extends Post {  
}  
$breve = new Breve();  
echo $breve->display('foo'); // affiche foo
```

Fig.6

Comme nous le voyons, bien que la classe *Breve* ne possède pas a priori la méthode *display*, le fait d'hériter (d'étendre) de la classe *Post* permet d'utiliser cette méthode.

Autre exemple avec cette fois un attribut dans la classe mère:

```
class Post  
{  
    protected $value;  
    public function __construct($data){  
        $this->value = $data;  
    }  
}  
class Breve extends Post {  
}  
class Filet extends Post {  
}  
$breve = new Breve('hello');  
var_dump($breve);
```

Fig.7

Le `var_dump` montre bien la valeur de l'attribut `value` qui n'est pourtant pas déclaré dans l'objet `Breve` et donc la valeur a été affectée par le constructeur de la classe mère lors de la création de l'objet `Breve`.

```
object(Breve) [1]
protected 'value' => string 'hello' (length=5)
```

Fig.8

À ce propos, les attributs protégés (`protected`) ont un comportement similaire aux attributs privés (`private`), c'est-à-dire qu'ils ne peuvent pas être utilisés en dehors de la classe, sauf dans les classes dérivées. Ainsi il est fréquent et standard que les attributs des classes soient `protected`. Dans ce cas, les attributs ne sont pas précédés par un `underscore`.

Nous découvrons ici la puissance de l'héritage qui peut être effectué à l'infini. Évoquons tout de suite l'impossibilité de mise en œuvre d'un héritage multiple du type: `class A extends B extends C extends D`, etc. Ceci est en effet interdit en PHP mais peut toutefois être contourné avec `class A extends B; class C extends A; etc.`

Nous reprenons l'étude de notre projet initial en organisant sa nouvelle architecture nous permettant de gérer les objets `brèves` ou `filets` (et à terme d'ajouter de nouveaux objets).

## II. Nouvelle architecture

Nous définissons donc notre nouvelle architecture comme suit:

- une classe mère: `Post`;
- une classe fille: `Breve`;
- une classe fille: `Filet`;
- une classe dédiée à la gestion de la relation avec la base de données: `postManager`.

Sans surprise la classe `Post` contient les attributs et les méthodes vues pour la classe `Breve` auxquels nous ajoutons naturellement l'attribut `title`, ainsi que les `getters` et `setters` adéquats.

Quant à la classe `postManager`, ses méthodes devront permettre de gérer le titre.

## A. La classe générique Post

Nous écrivons ici le code complet de la classe *Post* dont vont hériter les classes *Breve* et *Filet*.

Les ajouts relatifs à l'attribut *title* sont en rouge.

```
class Post {
    protected $id;
    protected $text;
    protected $dt_creation;
    protected $title;

    // Constructeur
    public function __construct(array $data) {
        $this->setId($data['id']);
        $this->setText($data['text']);
        $this->setDt_creation($data['dt_creation']);
        $this->setTitle($data['title']);
    }
    //setter
    public function setId($id) {
        if((is_int($id)) AND ($id > 0)) {
            $this->id = $id;
        }
    }
    public function setTitle($title) {
        $this->title = $title;
    }
    public function setText($text) {
        if (is_string($text)) {
            $this->text = $text;
        }
    }
    public function setDt_creation($dt_creation) {
        list($y, $m, $d) = explode("-", $dt_creation);
        if(checkdate($m, $d, $y)){
            $this->dt_creation = $dt_creation;
        }
    }
    // getter
    public function getId() {
        return $this->id;
    }
    public function getTitle() {
        return $this->title;
    }
    public function getText() {
        return $this->text;
    }
    public function getDt_creation() {
        return $this->dt_creation;
    }
}
```

Fig.9

Nous écrivons également le code complet de la classe *postManager*:

```
class postManager
{
    private $db; // Instance de PDO
    public function __construct($db) {
        $this->setDb($db);
    }
    public function setDb(PDO $dbh) {
        $this->_db = $dbh;
    }
    public function addPost (Post $post) {
        $sql = 'INSERT INTO post (text, dt_creation, title)
                VALUES (:text, :dt_creation, :title)';
        $stmt=$this->_db->prepare($sql);
        $stmt->bindParam(':text', htmlspecialchars
        ($post->getText()));
        $stmt->bindParam(':dt_creation', $post->getDt_creation());
        $stmt->bindParam(':title', $post->getTitle());
        $stmt->execute();
    }
    public function getPost ($id = ''){ // vide par défaut
        if(empty($id)) {
            $sql = 'SELECT id, text, dt_creation, title FROM post';
            $stmt=$this->_db->prepare($sql);
        }
        elseif (is_numeric($id)) {
            $sql = 'SELECT id, text, dt_creation, title FROM post
                    WHERE id =:id';
            $stmt=$this->_db->prepare($sql);
            $stmt->bindParam(':id', $id);
        }
        $stmt->execute();
        while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
            $result[] = $row;
        }
        return $result;
    }
    public function updatePost (Post $post)
    {
        $sql = 'UPDATE post SET text = :text, dt_creation = :dt_creation ,
                  title = :title WHERE id = :id';
        $stmt=$this->_db->prepare($sql);
        $stmt->bindParam(':id', $post->getId());
        $stmt->bindParam(':text', $post->getText());
        $stmt->bindParam(':dt_creation', $post->getDt_creation());
        $stmt->bindParam(':title', $post->getTitle());
        $stmt->execute();
    }
    public function deletePost ($id)
    {
        $sql = 'DELETE FROM post WHERE id =:id';
        $stmt=$this->_db->prepare($sql);
        $stmt->bindParam(':id', $id);
        $stmt->execute();
        $count = $stmt->rowCount();
        return $count;
    }
}
```

Fig. 10

## B. Un objet Post pour les méthodes *addPost* et *updatePost*

Nous remarquons, en effet, que nous imposons un objet *Post* en argument pour ces deux méthodes (cf. surlignées en jaune). Or la classe mère *Post* sert essentiellement à construire des objets *brève* et *filet* via leur classe respective *Breve* et *Filet*. Nous n'aurons donc pas au sens strict des objets post à créer ou à mettre à jour.

La réponse à la question est liée aux propriétés de l'héritage. À partir du moment où un objet est créé à partir d'une classe dérivée, alors cet objet sera considéré comme conforme aux attentes.

Testons de suite dans un fichier à part les propriétés de l'héritage pour un objet imposé comme étant une valeur attendue par une méthode :

```
class A
{
    public function foo (A $a) // ici un objet A est imposé
    {
        return 'test';
    }
}
class B extends A {
}
class C {
}
$a = new A();
$b = new B(); // l'objet B instancie une classe dérivée de A
$c = new C();
echo $a->foo($a); // fonctionne correctement
echo $a->foo($b); // fonctionne correctement
echo $b->foo($b); // fonctionne correctement
```

Fig.11

En revanche, si nous passons un objet autre que A ou bien produit par une classe non dérivée de A...

```
echo $b->foo($c);
```

Fig.12

... alors nous avons l'erreur suivante qui est explicite :

```
Catchable fatal error: Argument 1 passed to A::foo() must be an
instance of A, instance of C given
```

Fig.13

Ainsi nous pouvons « généraliser » les objets imposés en indiquant qu'il est obligatoire que ce soit un objet *Post* car cela fonctionnera parfaitement pour les objets *brèves* et *filets* qui instantient les classes dérivées de la classe *Post*.

## C. La classe *Breve*

Écrivons maintenant la classe *Breve* en indiquant son héritage avec *extends*. Limitons-nous à la syntaxe suivante et testons le fonctionnement en créant un objet *brève*.

```
class Breve extends Post{  
}  
$breve_data = array('id' => 1, 'text' => 'Un texte', 'dt_creation'  
=> '2013-11-15' );  
$breve = new Breve($breve_data);
```

Fig. 14

Une erreur de type *Notice* est alors affichée:

```
Notice: Undefined index: title in chemin/du/fichier/Post.php
```

Fig. 15

En effet, dans son contexte d'héritage, la classe *Breve* hérite du setter *setTitle* de la classe *Post*. Or puisqu'aucun index 'title' n'existe dans le tableau *\$breve\_data* que nous utilisons, la valeur n'est donc pas définie pour ce *setter*.

## D. Une solution : la surcharge

La surcharge consiste à réécrire la méthode d'une classe mère dans une classe fille, généralement en la modifiant.

Nous pouvons donc effectuer la surcharge sur la méthode *setTitle* en lui faisant affecter **NULL** à l'attribut *\$title*.

Il faudra donc également spécifier un constructeur pour la classe *Breve* (ce qui est également une surcharge), ce constructeur appelant la méthode *setTitle* surchargée.

```
class Breve extends Post{  
    public function __construct(array $data) {  
        $this->setId($data['id']);  
        $this->setText($data['text']);  
        $this->setDt_creation($data['dt_creation']);  
        $this->setTitle();  
    }  
    public function setTitle($title) {  
        $this->title = null;  
    }  
}
```

Fig. 16

Ce qui nous donne comme résultat:

- Le tableau de données initial

```
array  
  'id' => int 1  
  'text' => string 'Un autre texte' (length=14)  
  'dt_creation' => string '2013-11-15' (length=10)
```

Fig. 17

- L'objet créé

```
object(Breve) [2]
protected 'title' => null
protected 'id' => int 1
protected 'text' => string 'Un autre texte' (length=14)
protected 'dt_creation' => string '2013-11-15' (length=10)
```

Fig.18

## E. Alternatives non recommandées

- Modifier le tableau de données initial

Nous aurions obtenu un objet identique sans effectuer les surcharges dans la classe *Breve* mais en ajoutant un index *title* dans le tableau de données initial :

```
$breve_data = array('id' => 1, 'text' => 'Un autre texte', 'dt_
creation' => '2013-11-15', 'title' => null );
```

Fig.19

Cependant, nous aurions perdu en rigueur et en robustesse. En effet, en instaurant que le modèle d'une brève (c'est-à-dire la classe) a pour propriété un titre de valeur *null*, nous sommes certains que ce sera toujours le cas, notamment en cas d'erreurs éventuelles lors de la création de ce type de contenu.

- Définir une valeur **NULL** par défaut à l'attribut *title*

```
protected $title = NULL ;
```

Fig.20

Il peut être tentant de définir directement par défaut la valeur d'un attribut pour obtenir le résultat attendu. Si ce résultat est effectivement correct, ce faisant nous n'utilisons pas le setter adéquat et dégradons ainsi l'homogénéité du code et donc nous perdons en lisibilité et en maintenabilité. Il est donc fortement déconseillé d'affecter une valeur directement à un attribut d'une classe (hormis pour des éventuels besoins de tests qui devront être temporaires).

## F. La classe Filet

Écrivons maintenant la classe *Filet* puis créons un objet *filet*.

```
class Filet extends Post{
}
$filet_data = array('id' => 1, 'text' => 'Un autre texte', 'dt_
creation' => '2013-11-15', 'title' => 'Titre du filet' );
$filet = new Filet($filet_data);
```

Fig.21

L'objet filet a bien été créé.

```
object(Filet) [3]
protected 'id' => int 1
protected 'text' => string 'Un autre texte' (length=14)
protected 'dt_creation' => string '2013-11-15' (length=10)
protected 'title' => string 'titre du filet' (length=14)
```

Fig.22

### III. Une solution alternative et l'opérateur de résolution de portée

Il est important de bien considérer que la rigueur et la logique sous-jacentes à une mise en œuvre correcte de la POO ne brident pas la souplesse de la conception et que, à l'instar d'autres développements en PHP, plusieurs manières de faire peuvent tout à fait être correctes.

Dans le cas présent, tout en préservant l'esprit de la POO, au lieu d'effectuer le travail sur la classe *Breve*, nous aurions très bien pu arriver à des résultats identiques en effectuant des surcharges dans la classe *Filet* et non dans la classe *Breve*.

Pour ce faire, il aurait fallu :

- ne pas ajouter l'attribut *title* (et les méthodes associées) dans la classe mère *Post*;
- et intégrer cet attribut (et les méthodes associées) dans la classe *Filet*.

C'est un travail que nous allons effectuer en copiant notre répertoire de travail actuel et en indiquant qu'il s'agit d'une deuxième version. Après avoir supprimé les membres (attributs et méthodes) de la classe *Post* qui concernent le titre, nous écrivons la nouvelle classe *Filet*.

Celle-ci possédera donc l'attribut *title* et les méthodes *getTitle* et *setTitle*. Son constructeur devra donc à la fois appeler :

- le setter *setTitle*;
- ainsi que les *setters* liés à l'*id*, au texte et à la date de création.

Nous remarquons que ces derniers éléments constituent déjà le constructeur de la classe mère *Post*. Il est donc pertinent de réutiliser directement ce constructeur dans la classe *Filet*.

Pour y parvenir, nous utilisons la syntaxe suivante :

```
parent::__construct($data);
```

Fig.23

Le mot-clé *parent* fait référence à la classe mère et l'**opérateur de résolution de portée** : (symbole « double deux-points ») permet d'appeler le membre (attribut, constante ou méthode) d'une classe ; le constructeur n'étant finalement comme nous l'avons vu qu'une méthode particulière.

Le reste de la classe étant composé de l'attribut *title*, du *getter* et du *setter*:

```
class Filet extends Post{
    protected $title;
    /* Le constructeur appelle à la fois le constructeur de la
    classe mère ainsi que la méthode setTitle.*/
    public function __construct(array $data) {
        // appel du constructeur de la classe parent
        parent::__construct($data);
        // appel dans le constructeur du setter
        $this->setTitle($data['title']);
    }
    // getter
    public function getTitle() {
        return $this->title;
    }
    // setter
    public function setTitle($title) {
        if (is_string($title)) {
            $this->title = $title;
        }
    }
}
```

Fig.24

La classe *Breve* étant réduite à la simple syntaxe suivante :

```
class Breve extends Post {
}
```

Fig.25

## A. Une dernière modification sur la classe *postManager*

Il apparaît cependant qu'il est nécessaire de vérifier dans la classe *postManager* si l'objet qui est passé dans l'argument de la méthode *addPost* contient bien tous les *getters* et notamment *getTitle*. En effet, seule la classe *Filet* possède une telle méthode et donc, dans le cas de l'insertion d'une *brève* dans la base de données, l'erreur suivante va se produire :

```
Fatal error: Call to undefined method Breve::getTitle() in chemin/
vers/fichier postManager.php on line 19
```

Fig.26

La méthode *getTitle* n'existe en effet pas pour un objet *brève*. La solution consiste à vérifier l'existence de cette méthode pour en déduire la requête MySQL à effectuer. Pour ce faire, nous utilisons la fonction PHP *method\_exists* qui vérifie si la méthode existe pour l'objet fourni.

Page du Manuel : <http://php.net/manual/fr/function.method-exists.php>

Voici donc le résultat de notre intervention sur la méthode *addPost*:

```
public function addPost (Post $post)
{
    if (!method_exists($post, 'getTitle')) // c'est une brève
    {
        $sql = 'INSERT INTO post (text, dt_creation)
                VALUES (:text, :dt_creation)';
        $stmt=$this->_db->prepare($sql);
    }
    else
    {
        $sql = 'INSERT INTO post (title, text, dt_creation)
                VALUES (:title, :text, :dt_creation)';
        $stmt=$this->_db->prepare($sql);
        $stmt->bindParam(':title', htmlspecialchars($post-
>getTitle()));
    }
    $stmt->bindParam(':text', htmlspecialchars($post-
>getText()));
    $stmt->bindParam(':dt_creation', $post->getDt_creation());
    $stmt->execute();
}
```

Fig.27

Le développement de la seconde version alternative est terminé et il est bien sûr nécessaire de le tester:

- en créant une brève dans la base;
- en créant un filet dans la base.

Résultat:

```
mysql> select * from post;
```

id	text	dt_creation	title
1	Texte de breve	2013-11-15	NULL
2	Texte de filet	2013-11-15	Titre du filet

Fig.28

## B. Aide au choix de conception

Comment choisir la meilleure solution entre ces deux possibilités que sont la surcharge dans la classe *Breve* ou dans la classe *Filet*?

La réponse dépend clairement du contexte.

Si le projet a besoin d'autres types de contenus de *publication (post)* qui possèdent plus d'éléments comme le nom d'un auteur, le nom d'un fichier image ou un lien d'un flux vidéo et que ces types de publications ont tous par défaut besoin d'un titre, alors il sera plus logique d'effectuer la surcharge dans la classe *Breve*, qui sera la seule à ne pas avoir de titre.

**Il en ressort qu'il est tout à fait indispensable de bien avoir une vision d'ensemble sur les besoins du projet afin de déterminer en amont la conception la plus adaptée.**

Ajoutons toutefois qu'en cas d'erreur de conception (suite à une analyse imparfaite du développeur ou à une communication défectueuse de la part du client qui évoque son projet), l'architecture organisée d'un développement en POO permet de corriger le tir plus simplement et de s'y retrouver plus facilement du fait de la logique suivie.

Par ailleurs les développements effectués ici sont par essence réutilisables, notamment le principe des *getters* et *setters* ainsi que les opérations CRUD de la classe *postManager*. Les développements web sont en effet souvent construits selon le même principe et la bonne compréhension de ce système de gestion de l'information donne des clefs efficaces pour aborder sérieusement un projet professionnel.

Pour la suite de notre étude, nous considérons qu'il peut être envisagé d'intégrer à terme d'autres types de contenus qui tous posséderont un titre et donc nous garderons la première version de notre développement, à savoir la surcharge de la méthode *setTitle* dans la classe *Breve*.

## IV. Les constantes de classe

En POO comme dans le code procédural, il est possible d'utiliser des constantes, c'est-à-dire de créer dans une classe des valeurs constantes non modifiables.

La syntaxe repose simplement sur le mot-clé *const* suivi du nom de la constante et de l'affectation de sa valeur.

```
class A {  
    const FOO = 'bar';  
}
```

Fig.29

- **En dehors de la classe**, l'accès à cette valeur constante non modifiable s'effectue avec l'opérateur de résolution de portée :: (symbole double deux-points).

Elle peut être utilisée sans **instanciation de la classe** ou suite à une instanciation de la classe.

### Sans **instanciation**

```
echo A::FOO;
```

Fig.30

### Après **instanciation**

```
$a = new A();  
echo $a::FOO;
```

Fig.31

- À l'intérieur de la classe, nous devons utiliser le mot-clé **self** suivi de l'opérateur de résolution de portée pour accéder à la constante de classe.

```
class A {
    const FOO = 'bar';
    public function show() {
        return self::FOO;
    }
}
$a = new A();
echo $a->show();
```

Fig.32

Attention ! Il ne faut pas utiliser la métavariable **\$this** avec les constantes.

```
public function show() {
    return $this->FOO; // retourne une erreur
}
```

Fig.33

Nous obtiendrons l'erreur suivante :

```
Notice: Undefined property: A::$FOO
```

Fig.34

Ceci est dû au fait que **\$this** et **self** ont chacun un rôle spécifique.

Puisque les constantes de classe sont par définition liées à la classe, l'usage de **self** pour y accéder est tout à fait logique.

Il existe d'autres usages de l'opérateur de résolution de portée et de **self**, notamment pour accéder à des **attributs ou des méthodes dits statiques, qui n'ont pas pour objectif d'être utilisés dans le cadre d'une instanciation** et pour lesquels est associé le mot-clé **static**.

Nous utiliserons plus loin les éléments statiques dans le cadre de la gestion des erreurs lors de la création d'objets, mais il peut être intéressant d'en prendre déjà connaissance sachant que la syntaxe ne posera pas de problèmes particuliers.

- Page du manuel PHP sur l'opérateur de résolution de portée :

<http://www.php.net/manual/fr/language.oop5.paamayim-nekudotayim.php>

- Page du manuel sur les attributs et méthodes statiques :

<http://www.php.net/manual/fr/language.oop5.static.php>

Concernant notre projet, il est intéressant de se pencher sur l'usage des constantes de classe pour optimiser la classe *Breve*.

Pour mémoire, nous avons surchargé la méthode `setTitle` en lui faisant assigner à l'attribut `$title` la valeur `null` comme ceci :

```
$this->title = NULL;
```

Fig.35

Or, dans le sens d'une plus grande structuration, il est pertinent d'envisager que la valeur que nous affectons soit issue d'une constante de classe dont la valeur n'est pas modifiable. Le code final sera donc :

```
class Breve extends Post{
    const TITLE_BREVE = NULL; // définition de la constante
    public function __construct(array $data) {
        $this->setId($data['id']);
        $this->setText($data['text']);
        $this->setDt_creation($data['dt_creation']);
        $this->setTitle();
    }
    public function setTitle() {
        // utilisation de la valeur contenue dans la constante
        $this->title = self::TITLE_BREVE;
    }
}
```

Fig.36

La valeur souhaitée est donc disponible dans une constante de classe et la méthode appelle avec `self` cette constante pour en affecter la valeur à l'attribut `title`.

Ceci évite de se retrouver avec des valeurs éparpillées au sein du code dans les différentes méthodes et participe ainsi de la rigueur du développement et de sa réutilisabilité à terme.

En effet, les méthodes ne faisant qu'appeler des constantes de classe, il suffit pour une autre classe dérivée ou adaptée, par exemple dans le cadre d'un projet similaire, d'intervenir dans le code uniquement au niveau de ces constantes.

## V. Contrôle lors de la création des objets

Afin de ne pas créer d'objets incomplets ou dont les propriétés ne seraient pas celles attendues, il nous est nécessaire de contrôler celles-ci. Au stade actuel du développement, si nous élaborons un objet sur la base de valeurs non conformes à celles spécifiées dans les `setters`, nous obtiendrons un objet avec tout ou partie des attributs qui seront vides (`NULL`).

### Exemple

```
$filet_data = array('id' => "1", 'text' => true, 'dt_creation' => 0,
                    'title' => 16);
$filet = new Filet($filet_data);
```

Fig.37



À propos et pour mémoire si besoin, il peut être utile de se référer aux pages du manuel PHP consacrées aux types : <http://www.php.net/manual/fr/language.types.php>

- Le type attendu de l’attribut *id* est un entier et nous soumettons une chaîne de caractères.
- Le type attendu de l’attribut *text* est une chaîne de caractère et nous soumettons un booléen.
- Le type attendu de l’attribut *dt\_creation* est un format de date et nous soumettons un entier.
- Le type attendu de l’attribut *title* est une chaîne de caractère et nous soumettons un entier.

Conséquence, l’**objet** est créé mais, dans notre exemple, l’ensemble des propriétés possède la **valeur NULL**:

```
object(Filet) [3]
protected 'id' => null
protected 'text' => null
protected 'dt_creation' => null
protected 'title' => null
```

Fig.38

Une première solution pourrait consister à laisser l’objet être créé puis d’effectuer les vérifications sur les valeurs des propriétés dans le code.

Bien que fonctionnelle, cette piste est sans doute moins pertinente qu’une vérification intégrée au sein de la classe qui respecterait mieux la cohérence du principe suivant: un objet est construit si l’ensemble de ces propriétés est conforme aux attentes.

Pour effectuer un parallèle avec le monde réel, que dirait-on d’un vélo sans roues ou d’une chaise sans pied?

Nous allons donc intégrer les contrôles dans la classe sur la base du principe suivant:

- chaque setter pourra en cas d’erreur engendrer un message;
- ce message sera affecté à une variable *\$error*;
- le constructeur stoppe la création de l’objet si la variable *\$error* n’est pas vide.

**Il faut remarquer que la variable \$error ne concerne pas l’objet mais la classe.** En effet un objet *brève* ou *filet* ne possède pas d’attributs *\$error*. En revanche la classe en a besoin pour construire l’objet, il s’agit donc d’une donnée intrinsèque et spécifique à la classe.

Par ailleurs nous devons pouvoir accéder à cette variable sans devoir instancier la classe puisqu’une probabilité existe justement qu’elle ne le sera pas. Et c’est typiquement ce que permettent les attributs ou les méthodes statiques. Nous définirons donc la variable *\$error* comme étant statique.

Afin de respecter l’architecture de la classe, nous créerons un *setter* et un *getter* dédié à la mise à jour et à la récupération de *\$error*, et concernant les messages d’erreur, pour une meilleure lisibilité nous les définirons dans des constantes.

Enfin, nous utiliserons les exceptions que nous avons vues précédemment. Le constructeur lancera une exception si `$error` n'est pas vide et nous récupérons cette exception avec le couple `try/catch`. Le constructeur utilisera le mot-clé `self`; pour accéder à la variable statique.

Voici la partie de la classe `Post` qui intègre la gestion des erreurs ainsi que la méthode `setId`.

Vous êtes invités à finaliser la mise à jour de la classe pour les autres `setters`.

```
class Post {
    protected $id;
    protected $text;
    protected $dt_creation;
    protected $title;
    // déclaration de l'attribut statique $error
    protected static $error;
    // déclaration des messages d'erreur dans des constantes
    const MSG_ERROR_ID = 'ID doit être un entier.';
    const MSG_ERROR_TEXT = 'TEXT doit être une chaîne de caractères.';
    const MSG_ERROR_DATECREATION = 'DATE doit être format YYYY-MM-DD. ';
    const MSG_ERROR_TITLE = 'TITRE doit être une chaîne de caractères. ';
    const MSG_ERROR_END = 'L\'objet ne peut pas être créé.';
    /* le constructeur intègre désormais une vérification de $error
    Il lance une exception si $error n'est pas vide */
    public function __construct(array $data) {
        $this->setId($data['id']);
        $this->setText($data['text']);
        $this->setDt_creation($data['dt_creation']);
        $this->setTitle($data['title']);
        // utilisation de self:: pour accéder à error
        if(!empty(self::$error)) {
            throw new Exception(self::$error . self::MSG_ERROR_END);
        }
    }
    // gestion des erreurs
    public function setError($msg) {
        self::$error = $msg;
    }
    public function getError() {
        return self::$error;
    }
    // appel de setError si la valeur attendue n'est pas conforme
    public function setId($id) {
        if((is_int($id)) AND ($id > 0)) {
            $this->id = $id;
        }
        else {
            $this->setError(self::MSG_ERROR_ID);
        }
    }
    //etc.
```

Fig.39

Il ne faut naturellement pas oublier de mettre à jour les classes dérivées `Breve` et `Filet`.

- La classe *Filet* ne contenait aucune instruction et ne nécessite pas de mise à jour particulière, l'héritage mis en œuvre fait que cette classe hérite également de la nouvelle gestion des erreurs.
- La classe *Breve* nécessite que son constructeur contienne le lancement de l'exception. Notons que puisque la classe *Breve* est une classe dérivée, nous utilisons le mot-clé *parent* au lieu de *self*, car nous faisons bien référence à la variable déclarée dans sa classe mère.

```
public function __construct(array $data) {
    $this->setTitle();
    $this->setId($data['id']);
    $this->setText($data['text']);
    $this->setDt_creation($data['dt_creation']);
    if (!empty(parent::$error))
    {
        throw new Exception(parent::$error . parent::MSG_ERROR_END);
    }
}
```

Fig.40

Il ne reste plus qu'à récupérer l'exception dans la suite du code.

```
# 1. objet breve
try {
    $breve = new Breve($breve_data);
    var_dump($breve);
} catch (Exception $e) {
    echo $e->getMessage();
}
# 2. objet filet
try {
    $filet = new Filet($filet_data);
    var_dump($filet);
} catch (Exception $e) {
    echo $e->getMessage();
}
```

Fig.41

Et à effectuer des tests avec des valeurs correctes... ou des valeurs incorrectes (ici un *ID* incorrect)

Le ID doit être un entier. L'objet ne peut pas être créé.

Fig.42

Nous avons donc amélioré nos classes en intégrant un système de gestion des erreurs.