

# POO avancée : le système de brèves

# Sommaire

<b>I. Structure du contenu .....</b>	<b>3</b>
<b>II. Le besoin : création et affichage d'une brève.....</b>	<b>4</b>
A. Démarrer en définissant les attributs de la classe .....	4
B. Des méthodes pour accéder et modifier les attributs .....	5
C. Retour sur le constructeur.....	6
D. Une autre classe spécifique pour la relation avec la base de données.....	7

Crédits des illustrations:  
© DR.

Nous avons déjà abordé la POO et avons appréhendé le principe de la mise en œuvre des classes et des objets. Dans ce module, nous allons découvrir certaines notions utiles pour tout développement et appliquer la méthodologie inhérente à tout projet élaboré en POO.

Il sera donc utile de se référer avant de débuter, aux premières approches de la POO que nous avons détaillés précédemment.

Nous nous servirons par la même occasion de nos nouvelles connaissances relatives aux requêtes préparées et à PDO et serons à même de bien saisir les avantages de la POO.

Nous allons nous servir d'un système de brèves pour explorer plus avant la POO. Dans le langage journalistique, une brève a une signification précise : il s'agit d'un contenu court, daté mais non titré. C'est donc un type de contenu très simple à mettre en œuvre.

## I. Structure du contenu

La structure de ce contenu est :

- le corps du contenu ;
- une date.

Nous aurons donc une base de données que nous appellerons dbpost qui contiendra une table *brève* dont la structure ici obtenue avec la commande MySQL DESCRIBE sera :

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

Fig. 1

Une colonne *id* est la clef primaire, une colonne *text* contient le contenu de la brève et une colonne *dt\_creation* stocke la date de création de la brève au format **DATE** de MySQL (YYYYMM-DD, par exemple 2013-11-17).

La colonne pourrait également s'appeler *date* car ce n'est pas un mot réservé par MySQL mais nous préférerons éviter les ambiguïtés éventuelles avec les autres fonctions et mots-clés que nous utiliserons par la suite.

À noter que toutes ces colonnes seront **NOT NULL**, c'est-à-dire qu'elles devront obligatoirement contenir une valeur.

## II. Le besoin : création et affichage d'une brève

Très classiquement, nous avons besoin de créer des brèves et de les afficher. Pour ce faire, nous allons utiliser la POO en choisissant de respecter le fonctionnement prévu des objets et nous créons donc une classe *Breve* afin de pouvoir l'instancier et de créer autant d'objets *breve* que souhaité.

```
<?php  
class Breve  
{  
}  
?>
```

Fig.2



La première étape consiste à analyser la structure du contenu et donc la structure de la base.

### A. Démarrer en définissant les attributs de la classe

La règle est que toutes les colonnes constitueront les attributs de la classe, ce qui est logique puisque les attributs ou propriétés sont le moyen de décrire l'objet.

#### *Exemple*

Un vélo de couleur rouge est défini par sa couleur et donc la couleur est un de ses attributs.

Quelles sont les colonnes de la table ? La réponse est simple : id, text et dt\_creation.

Chacun de ces éléments est donc un attribut de la classe *Breve* et nous déclarons donc comme suit les attributs respectant le nommage des intitulés de colonnes de la table :

```
class Breve  
{  
    // attributs  
    $id;  
    $text;  
    $dt_creation;  
}
```

Fig.3

Nous avions vu précédemment, que nous devions mentionner le type d'attribut avant de le déclarer. Nous allons poser le fait que tous les attributs que nous utiliserons désormais seront du *type private* (privés), sauf dans le cas particulier que nous aborderons plus tard.

Pour mémoire **les attributs de type private ne peuvent pas être utilisés en dehors de la classe**. La règle de notation veut que les attributs privés soient préfixés par un *underscore*.

Puisque nous ne pouvons pas les utiliser directement, cela implique que nous devons déclarer des méthodes afin de leur affecter une valeur et de récupérer cette valeur.

En effet, le type *private* empêche de faire ceci :

```
$breve = new Breve();
echo $breve->$_id; // erreur PHP
```

Fig.4

## B. Des méthodes pour accéder et modifier les attributs



Pour mémoire, rappelons que la métavariable `$this` que nous avons déjà utilisée permet de récupérer n'importe quel attribut ou méthode de la classe et que dans le cas des attributs, le dollar est omis.

Afin d'accéder aux attributs et pouvoir modifier leur valeur nous créons des méthodes dédiées pour chacun des attributs.

Pour bien les identifier, et par convention, les méthodes d'accès seront préfixées par *get*, nous les appellerons les getters (également appelées *accesseurs – accessor* en anglais) et les méthodes de modification seront préfixées par *set*, nous les appellerons les setters (également appelées *mutateurs – mutator* en anglais). Cette syntaxe reprend le nom de l'attribut mais ne contient pas le underscore de la variable et l'écriture sera du type *camelCase* qui permet d'écrire tous les mots accolés et en les distinguant avec une majuscule, sauf le premier (comme*CeciParExemple*).

Ainsi le getter (*get*) de `$_id` sera `getId`, le setter (*set*) de `$_dt_creation` sera `setDt_creation`. Ajoutons que ces méthodes seront de type public (sauf si un besoin spécifique du développement devait imposer un autre type).

*Exemple pour `$_id`:* `$this->_id;`

Nous écrivons donc les méthodes suivantes :

```
class Breve
{
    private $_id;
    private $_text;
    private $_dt_creation;
    //setter
    public function setId($id) {
        // affecte à $_id la valeur $id passée en argument
        $this->_id = $id;
    }
    public function setText($text) {
        // affecte à $_text la valeur $text passée en argument
        $this->_text = $text;
    }
    public function setDt_creation($dt_creation) {
        // affecte à $_dt_creation la valeur passée en argument
        $this->_dt_creation = $dt_creation;
    }
    // getter
    public function getId() {
        // permet de récupérer la valeur de l'attribut $_id
        return $this->_id;
    }
    public function getText() {
        // permet de récupérer la valeur de l'attribut $_text
        return $this->_text;
    }
    public function getDt_creation() {
        // permet de récupérer la valeur de l'attribut $_dt_creation
        return $this->_dt_creation;
    }
}
```

Fig.5

Puisque les attributs et les méthodes *getter* et *setter* sont déclarés, nous pouvons effectuer un premier essai de fonctionnement avec des valeurs de test.

```
$breve = new Breve();
$breve->setId(1);
$breve->setText('Contenu de la brève');
$breve->setDt_creation('2013-11-14');
echo $breve->getId();
echo $breve->getText();
echo $breve->getDt_creation();
```

Fig.6

## C. Retour sur le constructeur

Nous avions vu que le constructeur nommé `__construct()` sert, comme son nom l'indique, à construire l'objet lors de l'instanciation. Ce qui revient à dire que l'objet se sert du constructeur de la classe pour être créé.

Il apparaît donc pertinent que le constructeur active les *setters* lors de la création d'un objet afin que celui-ci alimente les valeurs des attributs.

```
class Breve
{
    private $_id;
    private $_text;
    private $_dt_creation;

    public function __construct(array $data)
    {
        $this->setId($data['id']);
        $this->setText($data['text']);
        $this->setDt_creation($data['dt_creation']);
    }
    // ici les setter
    // ici les getter
}
```

Fig.7

C'est donc le constructeur lorsqu'il est appelé par le mot-clé `new` qui va récupérer les données nécessaires à l'instanciation de l'objet. Ici comme nous le constatons dans l'argument ces données sont sous forme de tableau (`$data`). D'ailleurs il peut être utile d'obliger à ce que les valeurs soient du type tableau de données et nous ajoutons alors `array` dans l'argument, si les données ne sont pas un tableau alors le constructeur n'utilisera pas ces données, ce qui est un bon moyen de contrôler les valeurs utilisées.

```
$values = array('id' => 1, 'text' => 'Contenu de la brève', 'dt_
creation' => '2013-11-14' );
$breve = new Breve($values);
echo $breve->getId();
echo $breve->getText();
echo $breve->getDt_creation();
```

Fig.8

Si \$values n'est pas un tableau de données, une erreur sera donc générée car nous avons interdit que la valeur passée en argument dans le constructeur soit autre chose qu'un tableau de données.

```
Catchable fatal error: Argument 1 passed to Breve::__construct()
must be an array
```

Fig.9

En résumé, lors de l'instanciation (c'est-à-dire la création de l'objet sur la base de la structure de la classe) le constructeur est appelé. Les valeurs utiles pour créer l'objet sont passées en argument au constructeur. Le constructeur active les *setters* pour ajouter les valeurs aux attributs de l'objet.

```
public function setId($id) {
    if((is_int($id)) AND ($id > 0)) {
        $this->_id = $id;
    }
}
public function setText($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;
    }
}
```

Fig.10



**Explode:** <http://php.net/manual/fr/function.explode.php>

**List:** <http://php.net/manual/fr/function.list.php>

**Checkdate:** <http://php.net/manual/fr/function.checkdate.php>

Les contrôles des *setters* *setId* et *setText* ne posent pas de difficulté particulière. Le contrôle de la date implique l'utilisation de la fonction native PHP *checkdate()* que nous pourrons mettre en œuvre après avoir utilisé la fonction *explode()* et l'élément de langage *list()* sur la valeur passée en argument.

## D. Une autre classe spécifique pour la relation avec la base de données

### 1. Une classe, un rôle

Nous avons maintenant notre classe complète et pouvons instancier autant de brèves que nous souhaitons. Il nous faut à présent établir la relation avec la base de données pour y ajouter des brèves et les afficher dans une page.

Nous pourrions ajouter des méthodes contenant les requêtes MySQL dans la classe, comme par exemple une méthode *insertNews*. Mais ce faisant, nous perdrons un des intérêts de la POO qui veut **qu'une classe ait un seul rôle**. La classe *Breve* a pour rôle d'instancier un objet brève en contrôlant le type mais n'a pas pour rôle d'effectuer les requêtes vers la base de données.

## 2. La classe de gestion de la relation à la base de données

Pour ce faire, nous allons donc créer une nouvelle classe dont le rôle sera uniquement d'effectuer ces requêtes. Cette classe étant dédiée à la gestion des brèves en relation avec la base de données nous l'appellerons *breveManager*.

Quels seront ces attributs ? Du fait que la classe a pour rôle d'effectuer les requêtes MySQL, il lui est nécessaire de disposer des informations de connexion à la base (connexion écrite avec PDO).

Il apparaît donc pertinent qu'un attribut de cette classe contienne la connexion à la base et nous nommons cet attribut *\$\_db* qui sera de type *private*. Il semble qu'aucun autre attribut ne soit nécessaire à la classe *breveManager*.

Concernant les méthodes à créer, nous considérons simplement que chaque requête doit correspondre à une et une seule méthode de la classe et de fait nous aurons autant de méthodes que de requêtes.

Pour commencer, nous écrivons notre classe avec une seule méthode d'ajout de brèves dans la base de données et nous appelons cette méthode *addBreve*. Nous ajouterons les autres méthodes par la suite.

Le code de la classe *breveManager* que nous écrivons à la suite de la classe *Breve* dans le même fichier pour le moment :

```
class breveManager
{
    private $_db; // Instance de PDO
    public function __construct($db)
    {
        $this->setDb($db);
    }
    public function setDb(PDO $dbh) // Un objet PDO est attendu
    {
        $this->_db = $dbh;
    }
    // insertion
    public function addBreve (Breve $breve) // Un objet Breve
est attendu
    {
        $sql = 'INSERT INTO breve (text, dt_creation)
VALUES (:text, :dt_creation)';
        $stmt=$this->_db->prepare($sql);
        $stmt->bindParam(':text', htmlspecialchars($breve-
>getText()));
        $stmt->bindParam(':dt_creation', $breve->getDt_creation());
        $stmt->execute();
    }
}
```

Fig. 11



Fonction *htmlspecialchars*:  
<http://php.net/manual/fr/function.htmlspecialchars.php>

Nous remarquons trois points :

- nous vérifions dans les deux méthodes qu'un objet est passé en argument;
- la méthode *addBreve* utilise les *getters* de la classe *Breve* pour récupérer les informations de l'objet *Breve*. C'est donc bien ici le rôle de la classe *Breve* de créer un objet et d'effectuer les vérifications;
- nous utilisons cependant pour des raisons de sécurité la fonction native *htmlspecialchars* afin que les éventuelles saisies en HTML soient remplacées, comme par exemple le signe < qui devient &lt;.

Toujours dans le même fichier, nous procédons à la création des objets utiles destinés à ajouter une nouvelle brève dans la base :

```
// creation d'un objet breve
$breve_data = array('id' => 1, 'text' => 'Texte pour le contenu', 'dt_
creation' => '2013-10-14' );
$breve = new Breve($breve_data);

// affectation dans la variable $db de la connexion PDO
$db = new PDO('mysql:host=localhost;dbname=emweb_news', 'root', '');

// Instanciation de la classe breveManager, nous créons un objet manager
// La connexion PDO est passée en paramètre au constructeur.
$manager = new breveManager($db);

// appel de la méthode addBreve, nous passons un objet breve en argument.
$manager->addBreve($breve);
// le manager a ajouté la nouvelle brève, à vérifier dans la base de données.
```

Fig. 12

Il peut paraître lourd d'effectuer l'intégralité de ce processus pour enregistrer une nouvelle brève dans la base mais il est important de considérer premièrement que nous possédons désormais un code très structuré qui sera donc à la fois plus fiable et plus simple à maintenir et deuxièmement, que l'ensemble des opérations seront intégrées, et pas uniquement l'insertion de nouvelles brèves dans la base de données.