

Utilisation des interfaces en PHP

par Alain Sahli ([Tutoriels PHP de Alain Sahli](#)) ([Blog](#))





Date de publication : 19.07.2007

Dernière mise à jour :


Cet article traite de l'utilisation des interfaces en PHP 5. Il n'est pas destiné à des débutants et de bonnes bases en POO sont requises pour la compréhension du code fourni tout au long de l'article.

- I - Introduction
- II - Les interfaces
 - II-A - Création
 - II-B - Implémentation
 - II-B-1 - Implémentation d'une interface
 - II-B-2 - Implémentation de plusieurs interfaces
 - II-B-3 - Utilisation du typage objet
 - II-B-4 - Analogie avec les classes abstraites
 - II-C - Exemple d'utilisation
 - II-C-1 - L'interface TextEditor
 - II-C-2 - Les classes de documents
 - II-C-2-a - La classe HTMLText
 - II-C-2-b - La classe PDFText
 - II-C-3 - Résultat final
- III - Les interfaces existantes
- IV - Conclusion

I - Introduction

Contrairement à  **PHP4**, PHP5 permet de gérer les  **interfaces**. Elles permettent de créer un modèle que les  **classes** qui l' **implémentent** doivent respecter. On peut comparer une interface à une fiche électrique, elle définit un standard de "connexion" sans savoir quels types d'appareils vont venir s'y connecter. Les interfaces sont très pratiques lorsque plusieurs personnes développent un même projet, ou lorsqu'on développe un site modulaire.

II - Les interfaces

Une interface regroupe toutes les  **méthodes** qu'une classe doit implémenter. Ce procédé permet d'appeler les méthodes d'une classe sans se soucier de l'existence de leur existence. Les méthodes déclarées dans une interface ne peuvent être de type `private` étant donné que l'on y définit les méthodes que l'on pourra appeler lors de l'instanciation. Déclarer des méthodes `private` reviendrait à imposer un style de traitement dans la classe alors que ce n'est pas le but. Les interfaces permettent également de définir des constantes, il suffit de les y déclarer à l'aide du mot clé **"const"**

Une des caractéristiques principales des interfaces est d'obliger les classes qui les implémentent à créer toutes leurs méthodes. Si ce n'est pas le cas, une erreur sera générée et c'est là toute la puissance d'une interface car elle nous assure que la classe contient bien les méthodes qu'elle doit implémenter.

II-A - Création

Pour créer une interface on procède de la même manière que lorsqu'on crée une classe. Toutefois à la place d'utiliser le mot clé `"class"`, on utilise le mot clé **"interface"**.


Création d'une interface

```
<?php
interface IMonInterface
{
    public function titi($name);
    public function toto($firstName);
}

interface IMonInterfaceStatic
{
    static function staticTiti($name);
    static function staticToto($firstName);
}
?>
```

II-B - Implémentation

L'implémentation se fait de la même manière que l'héritage, sauf que l'on utilise le mot clé **"implements"**. Une classe peut implémenter plusieurs interfaces.

 *Si une classe implémente une interface mais n'en redéfinit pas toutes les méthodes, une erreur sera générée !*

II-B-1 - Implémentation d'une interface

Comme vous pouvez le voir ci-dessous, j'ai simplement déclaré une classe `"MaClasse"` qui implémente l'interface `"IMonInterface"`. Ensuite il a fallu créer toutes les méthodes présentes dans l'interface `"IMonInterface"` ;

Implémentation d'une interface

```
<?php
require_once 'IMonInterface.php';
class MaClasse implements IMonInterface
{
    public function titi($name)
    {
        echo $name;
    }
}
```

Implémentation d'une interface

```

    }

    public function toto($firstName)
    {
        echo $firstName;
    }
}
?>
    
```

II-B-2 - Implémentation de plusieurs interfaces

Lorsqu'une classe implémente plusieurs interfaces, il faut les séparer par une ',' (virgule). La classe ci-dessous implémente deux interfaces ; il a donc fallu créer toutes les méthodes contenues dans l'interface "monInterface" ainsi que toutes les méthodes présentes dans l'interface "monInterfaceStatic".

Implémentation de plusieurs interfaces

```


<?php
require_once 'IMonInterface.php';
require_once 'IMonInterfaceStatic.php';


class MaClassMultiInterfaces implements IMonInterface, IMonInterfaceStatic
{
    public function titi($name)
    {
        echo $name;
    }

    public function toto($firstName)
    {
        echo $firstName;
    }

    static function staticTiti($name)
    {
        echo $name;
    }

    static function staticToto($firstName)
    {
        echo $firstName;
    }
}
?>
    
```

 Lors du débogage, il peut être utile de tester si une interface existe ! La fonction `interface_exists()` nous permet de le vérifier. En production cette fonction ne devrait pas être utilisée.

 Lors de l'implémentation de plusieurs interfaces, il faut faire attention de ne pas créer les mêmes noms de méthodes dans plusieurs interfaces !

II-B-3 - Utilisation du typage objet

Il est désormais possible de spécifier dans une méthode quel "type" d'objet il faut passer en paramètre. J'entends par "type" le nom d'une interface que la classe doit implémenter. Voici un exemple:

Typage objet

```

<?php
    
```

Typage objet

```

interface MyInterface
{
    public function sayHello();
}

class MyClass implements MyInterface
{
    public function sayHello()
    {
        echo 'Hello World!';
    }
}

class ErrorClass
{
    public function toto()
    {
        echo 'toto';
    }
}

class Main
{
    public function __construct(myInterface $obj)
    {
        $obj->sayHello();
    }
}


$myClass = new MyClass();
$main = new Main($myClass); // Pas d'erreur

$errorClass = new ErrorClass();
$main = new Main($errorClass); // Erreur car ErrorClass n'implémente pas l'interface MyInterface
?>
    
```



On peut tester si un objet implémente une interface grâce à l'opérateur instanceof de la manière suivante : `object instanceof interface_name`

II-B-4 - Analogie avec les classes abstraites

Une  **classe abstraite** est une classe qui ne peut pas être instanciée. Elle est généralement utilisée pour être héritée. On peut donc utiliser une classe abstraite comme une interface en créant simplement des méthodes vides. Il y a cependant quelques inconvénients à utiliser des classes abstraites plutôt qu'une interface. Par exemple une classe ne peut hériter que d'une seule classe, ce qui n'est pas le cas avec les interfaces.


Cependant les classes abstraites sont parfois utilisées communément avec les interfaces. Lorsque il y a beaucoup de méthodes déclarées dans une interface on peut créer une classe abstraite qui implémente les méthodes de cette interface. Ainsi il suffit d'implémenter l'interface puis hériter de la classe abstraite associée. Ensuite on surcharge uniquement les méthodes qui nous sont utiles.

II-C - Exemple d'utilisation

Nous sommes maintenant capables de créer et d'implémenter des interfaces. Réalisons un exemple concret.

Nous allons créer des classes qui permettent de créer des documents dans différents formats. Plus précisément une classe qui va créer des documents au format HTML et une autre qui va créer des documents au format PDF. Afin que toutes les classes aient le même fonctionnement nous allons créer l'interface TextEditor. Cette interface

contiendra deux méthodes, addTitle et addParagraph. Ainsi, l'utilisation d'une classe de format de document sera toujours la même.

 Cette exemple a été réalisé avec la version 5.2.0 de PHP. De plus, la librairie FPDF est nécessaire au bon fonctionnement de cet exemple.

II-C-1 - L'interface TextEditor

L'interface TextEditor ne contient que deux méthodes. Une méthode addParagraph qui va permettre d'ajouter un paragraphe dans le document et une méthode addTitle qui va permettre d'ajouter un titre en spécifiant à quel niveau on doit le placer.

Interface TextEditor

```
<?php
interface TextEditor
{
    public function addParagraph($text);
    public function addTitle($text, $level);
}
?>
```

II-C-2 - Les classes de documents

Nous allons donc créer deux classes de format de documents, HTMLText et PDFText.

II-C-2-a - La classe HTMLText

Classe HTMLText

```
<?php
class HTMLText implements TextEditor
{
    private $html;

    public function __construct($title)
    {
        $this->html = '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>'.$title.'</title>
</head>
<body>
<h1>'.$title.'</h1>';
    }

    public function addParagraph($text)
    {
        $this->html .= ' <p>'.$text.'</p>';
    }

    public function addTitle($text, $level)
    {
        $this->html .= ' <h'.$level.'>'.$text.'</h'.$level.'>';
    }

    public function __destruct()
    {
        $this->html .= ' </body> </html>';
    }
}
```

Classe HTMLText

```
    echo $this->html;
}
}
?>
```

II-C-2-b - La classe PDFText

Classe PDFText

```
<?php
class PDFText extends FPDF implements TextEditor
{
    public function __construct($title)
    {
        parent::__construct();
        $this->AddPage();
        $this->SetFont('Arial', 'B', 40);
        $this->Cell(40, 10, $title); $this->Ln();
    }

    public function addParagraph($text)
    {
        $this->SetFont('Arial', 'B', 16);
        $this->Cell(40, 10, $text);
        $this->Ln();
    }

    public function addTitle($text, $level)
    {
        $this->AddPage();
        $this->SetFont('Arial', 'B', bcdiv(70, $level));
        $this->Cell(40, 10, $text);
        $this->Ln();
    }

    public function __destruct()
    {
        $this->Output();
    }
}
?>
```

II-C-3 - Résultat final

Maintenant que tous ces fichiers sont créés, il nous faut encore créer l'index.php qui va utiliser ces deux types de document.

Fichier index.php

```
<?php
define('FPDF_FONTPATH', 'chemin/vers/fpdf/font/');
require 'fpdf/fpdf.php';

$doc = new PDFText('Document de test');
// $doc = new HTMLText('Document de test');

$doc->addTitle('Section 1', 2);
$doc->addParagraph('azertyui');
$doc->addParagraph('azertyui');
$doc->addParagraph('azertyui');

$doc->addTitle('Section 2', 2);
$doc->addParagraph('azertyui');
```

Fichier index.php

```
$doc->addParagraph( 'azertyui' );  
?>
```

Le code ci-dessus va donc créer un document au format PDF. Pour avoir le même documents au format HTML, il suffit de mettre en commentaire la ligne `$doc = new PDFTxt('Document de test')` et décommenter la ligne `$doc = new HTMLText('Document de test')`. On voit donc bien l'utilité d'une interface dans ce cas ! Il nous suffit de changer l'objet qui définit le type de document et ensuite comme les méthodes sont les mêmes pour toutes les classes de format de document, il n'y a rien à changer dans le code. De plus, si l'on veut créer une nouvelle classe pour un autre type de document, il nous suffit d'implémenter cette interface.

III - Les interfaces existantes

Peut-être l'aurez-vous déjà remarqué, lorsqu'on utilise la fonction `get_declared_interfaces()` PHP retourne le tableau suivant:

```
Array
(
    [0] => Traversable
    [1] => IteratorAggregate
    [2] => Iterator
    [3] => ArrayAccess
    [4] => reflector
    [5] => RecursiveIterator
    [6] => SeekableIterator
)
```

Ce sont des interfaces prédéfinies par PHP. Étant donné qu'un exemple est mille fois plus parlant que des lignes de texte je vais vous donner un exemple concret. Imaginez que vous aimeriez avoir un tableau qui ne supporte pas les clés de type 'string'. Vous savez que c'est impossible car PHP est très souple au niveau des tableaux et il accepte les clés de type 'string'. Pour remédier à ce problème, nous allons créer notre propre classe de gestion de tableau qui ne va accepter que les clés de type 'int'.

Gestion de tableau

```
<?php
class MyArray implements ArrayAccess
{
    protected $array = array();

    // Méthode d'ajout d'une valeur dans le tableau
    function offsetSet($offset, $value)
    {
        if(!is_numeric($offset))
        {
            throw new Exception('Clé invalide!');
        }
        $this->array[$offset] = $value;
    }

    // Retourne une valeur contenue dans le tableau
    public function offsetGet($offset)
    {
        return $this->array[$offset];
    }

    // Supprime une valeur du tableau
    public function offsetUnset($offset)
    {
        unset($this->array[$offset]);
    }

    // Test si une valeur existe dans le tableau
    public function offsetExists($offset)
    {
        return isset($this->array[$offset]);
    }
}

$myArray = new MyArray();
$myArray['a'] = 10; // va générer une exception 'Clé invalide'
$myArray[3] = 12;
?>
```

Les quatre méthodes implémentées dans la classe MyArray sont obligatoires ! Ce n'est pas moi qui ait choisi de les implémenter, c'est l'interface `ArrayAccess` qui m'oblige à les implémenter pour garantir le bon fonctionnement de mon tableau. Chaque interface a ses propres méthodes, je vous laisse les découvrir ici : **SPL**.

IV - Conclusion

Pour conclure je dirais que les interfaces vous permettent de rendre une application orientée objet flexible et évolutive :

- en dissimulant son fonctionnement interne;
- en offrant aux développeurs une base commune pour l'implémentation de modules complexes.

