

# POO PHP5 : Standard Php Library (SPL)

par Julien Pauli ([Tutoriels, article et conférences PHP et développement web](#)) ([Blog](#))

Date de publication : 14/01/2008

Dernière mise à jour :

PHP5 possède un modèle objet 'non vide' : il est agrémenté de classes et d'interfaces internes, réunis dans ce qu'on appelle la SPL, ou Standard PHP Library.

Nous allons voir en quoi ils peuvent s'avérer très utiles.

I - Introduction

II - Dans le coeur de la SPL

II-A - Iterator

II-B - Countable

II-C - RecursiveIterator

II-D - LimitIterator

II-E - FilterIterator

II-F - RecursiveDirectoryIterator

II-G - ArrayAccess

II-H - ArrayObject

III - Conclusion

## I - Introduction

La plus grande amélioration que PHP5 a apporté à sa sortie, a été un modèle objet complet, très semblable à celui de Java ou C#.

Mais les développeurs de PHP ont saisi cette occasion pour intégrer dans son coeur tout un ensemble de classes et d'interfaces (nativement écrites en C, donc).

Outre une approche procédurale conservée, il est désormais possible avec PHP5 de développer des pages et des processus entièrement orientés objets. La SPL, ou Standard PHP Library, est un ensemble de classes disponibles, prêtes à être utilisées ou implémentées.

Nous allons faire un petit tour de quelques puissantes fonctionnalités offertes par la SPL, et nous allons voir comment certaines fonctions natives du langage PHP peuvent être impactées.

Comme les images seraient trop grandes pour être affichées ici, **je vous propose quelques diagrammes directement sur le site officiel**, vous pouvez aussi **regarder par là**

Le développeur principal de la SPL est Marcus Boerger, il est actif sur les Mailing Lists de PHP et participe aussi au développement du modèle objet de PHP5.

Bien qu'intégrée nativement à PHP, la SPL possède son petit site à part, que vous trouverez ici : <http://www.php.net/~helly/php/ext/spl/>

La documentation principale de PHP est souvent peu précise à son sujet, mieux vaut donc se fier à cette doc.

Vous trouverez toutes sortes de diagrammes de classe sur ce site, ainsi que l'API. Personnellement j'utilise Zend Studio mais sa base d'autocomplétion n'est pas parfaite, car la SPL, comme PHP, bouge beaucoup. Elle s'étoffe, et s'améliore de version en version.

Qu'importe, nous n'allons pas la détailler complètement, car si je compte le nombre de classes :

```
<?php
var_dump(spl_classes());
```

Il m'en retourne 43. Attention, il prend en compte les classes d'Exception, la SPL possédant ses propres Exceptions dont certaines rappelleront un peu Java : OutOfRangeException par exemple.

De plus, cette fonction ne liste pas les interfaces, or elles demeurent importantes, listons-les :

```
<?php
var_dump(get_declared_interfaces());
```

Une manière plus simple : le très connu phpinfo() renvoie aussi des informations sur la SPL actuellement compilée :

## SPL

SPL support	enabled
<b>Interfaces</b>	Countable, OuterIterator, RecursiveIterator, SeekableIterator, SplObserver, SplSubject
<b>Classes</b>	AppendIterator, ArrayIterator, ArrayObject, BadFunctionCallException, BadMethodCallException, CachingIterator, DirectoryIterator, DomainException, EmptyIterator, FilterIterator, InfiniteIterator, InvalidArgumentException, IteratorIterator, LengthException, LimitIterator, LogicException, NoRewindIterator, OutOfBoundsException, OutOfRangeException, OverflowException, ParentIterator, RangeException, RecursiveArrayIterator, RecursiveCachingIterator, RecursiveDirectoryIterator, RecursiveFilterIterator, RecursiveIteratorIterator, RecursiveRegexIterator, RegexIterator, RuntimeException, SimpleXMLIterator, SplFileInfo, SplFileObject, SplObjectStorage, SplTempFileObject, UnderflowException, UnexpectedValueException

## II - Dans le coeur de la SPL

La SPL comporte ainsi des classes, et des interfaces (surtout des interfaces).

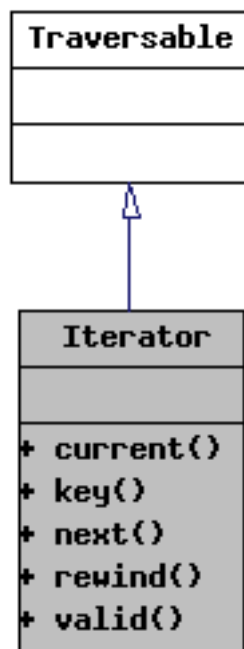
Le plus grand atout qu'offre la SPL est l'itérateur, et il y en a de toutes sortes. Rappelons d'abord ce qu'est un itérateur : C'est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent il s'agit d'un conteneur (tableau, objet, jeu de résultats, arbre, liste ...). Le conteneur doit alors fournir des méthodes à l'itérateur (en implémentant une interface) afin de lui permettre de le parcourir. On peut assimiler les itérateurs à des curseurs, dans le cas des bases de données.

En réalité, PHP utilise lui-même les itérateurs, à chaque fois que vous faites un `foreach()`, ou un `count()`.

La syntaxe PHP interne du `foreach` de PHP peut ressembler à ceci :

```

<?php
$iterator->rewind();
while ($iterator->valid())
{
    echo $iterator->current();
    $iterator->next();
}
    
```



Pour un type PHP array, nous nous avons à disposition des fonctions d'itération : `next()`, `prev()`, `key()`, `current()`, etc...

`foreach` fonctionne sur les objets depuis PHP5, car ils implémentent tous, en interne, l'interface `Traversable`.

Si j'utilise un `foreach` sur un objet, je vais avoir la liste de ses attributs publics, et de leurs valeurs, respectivement placés en clé, et valeur :

```

<?php
class bar
    
```

```
{
    private $var = 2 ;
    public $foo = 8;
}

$bar = new bar;
foreach ($bar AS $k=>$v)
{
    echo $k . " a la valeur " . $v;
}

// affiche foo a la valeur 8
```

## II-A - Iterator

Maintenant je peux changer ce comportement, en implémentant l'interface Iterator; rappelez-vous de la syntaxe avec le while situé plus haut :

```
<?php
class mon_array implements Iterator
{
    private $tab = array();
    private $pas;

    public function __construct(array $array,$pas = 1)
    {
        $this->tab = $array;
        $this->pas = (int)$pas;
    }

    public function valid()
    {
        return array_key_exists(key($this->tab), $this->tab);
    }

    public function next()
    {
        for ($i=1; $i<=$this->pas;$i++)
        {
            next($this->tab);
        }
        return $this;
    }

    public function rewind()
    {
        reset($this->tab);
        return $this;
    }

    public function key()
    {
        return key($this->tab);
    }

    public function current()
    {
        return current($this->tab);
    }
}
```

J'ai créé une classe "par dessus" un tableau, et je vais pouvoir spécifier le pas d'avancée de l'itération, dans le constructeur. Notez qu'il arrive que je retourne \$this, sur des méthodes ne demandant pas obligatoirement un return.

Si je le désire, je pourrai ainsi chaîner mes méthodes

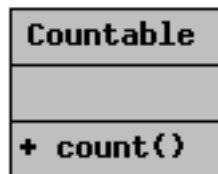
```
<?php
$tablo = range (0,10);
$mon_array = new mon_array($tablo,2);

foreach($mon_array AS $v)
{
    echo $v;
}
```

Ce script m'affiche une suite de chiffres pairs 0,2,4,6,8,10.

J'ai donc modifié le comportement de foreach, en lui demandant d'itérer les éléments par un pas que je donne au constructeur de mon objet, dans mon exemple.

## II-B - Countable

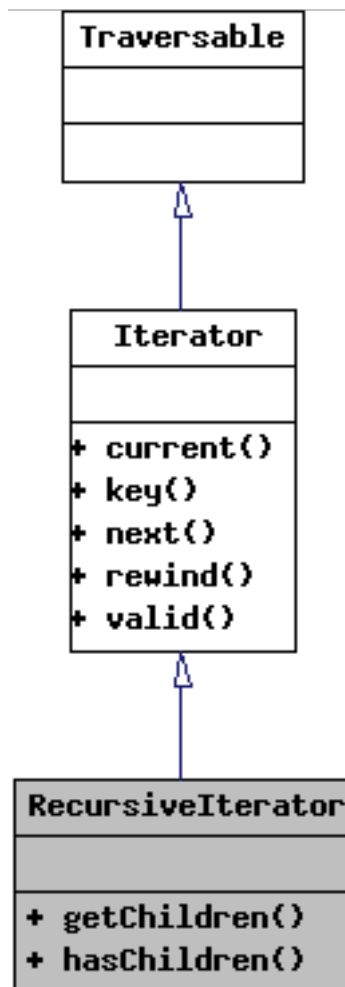


Vous pouvez de même implémenter countable, seule la méthode **count()** devra être définie, et elle va guider le comportement de la fonction *count()* de PHP :

```
<?php
class mon_array implements Iterator, Countable
{
    // la classe reste la même qu'au dessus, on rajoute cependant :
    public function count()
    {
        return count($this->tab);
    }
}
```

Je retourne naturellement le nombre de valeurs dans mon tableau, mais par exemple pour un conteneur représentant un jeu de résultats de base de données, le calcul aurait pu être différent.

## II-C - RecursiveIterator



Grâce à l'interface RecursiveIterator, je vais implémenter des méthodes qui vont permettre à l'itérateur d'itérer sur des éléments nichés les uns dans les autres. Le cas bateau représente un tableau, dans un tableau, dans un tableau ...

Il est très facile d'itérer toutes les valeurs d'un coup, il suffit d'implémenter l'interface RecursiveIterator, et d'utiliser l'objet d'itération RecursiveIteratorIterator adapté avec :

```

<?php
class mon_array implements RecursiveIterator
{
    private $tab = array();
    private $pas;

    public function __construct(array $array, $pas = 1)
    {
        $this->tab = $array;
        $this->pas = (int)$pas;
    }

    public function valid()
    {
        return array_key_exists(key($this->tab), $this->tab);
    }

    public function next()
    {

```

```
for ($i=1; $i<=$this->pas;$i++)
{
    next($this->tab);
}
return $this;
}

public function rewind()
{
    reset($this->tab);
    return $this;
}

public function key()
{
    return key($this->tab);
}

public function current()
{
    return current($this->tab);
}

public function hasChildren()
{
    return is_array(current($this->tab));
}

public function getChildren()
{
    return new self(current($this->tab));
}
```

L'exemple :

```
<?php
$tablo = range (0,10);
$tablo[] = array('u',array('w'));
$tablo[] = array('x',array('z'));
$a = new RecursiveIteratorIterator(new mon_array($tablo,1));

foreach($a AS $k=>$v)
{
    echo $v.' ' ;
}
```

Affiche : 0 1 2 3 4 5 6 7 8 9 10 u w x z

RecursiveIteratorIterator est un itérateur, et il attend obligatoirement un paramètre implémentant l'interface RecursiveIterator. C'est le cas de mon\_array. Il définit 2 nouvelles méthodes (par rapport à Iterator) qui vont être appelées par RecursiveIteratorIterator.

La première, **hasChildren()**, permet de savoir si le contenu en cours d'itération peut lui-même être itéré de la même manière que son père actuel. Savoir si le père possède un enfant.

On retourne donc un booléen qui dans notre cas regarde si l'élément couramment itéré est un autre tableau.

Si true est retourné, alors RecursiveIteratorIterator appelle immédiatement **getChildren()**, qui doit lui retourner un objet implémentant RecursiveIterator à son tour, et il commence l'itération dessus.

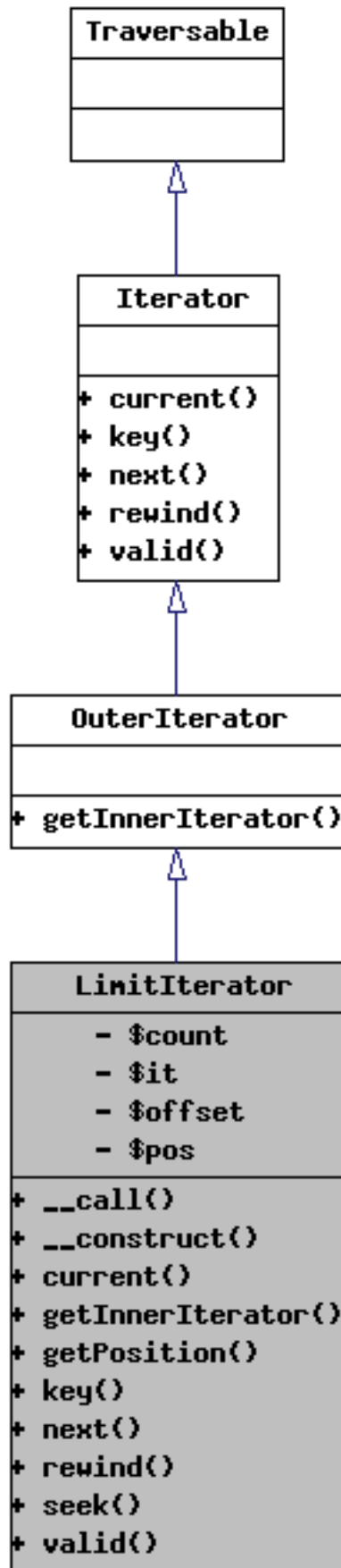
Lorsqu'il a fini avec un enfant, il reprend le cours d'itération de son père, et ainsi de suite, quelle que soit la profondeur.

Nul besoin d'écrire des foreach dans des foreach..., la récursivité est assurée en interne par l'interface.

Vous pouvez créer de la même manière un menu de type 'fil conducteur' , contenant des éléments de menus, ou des menus, etc. De manière plus générale, tout élément composite, c'est à dire pouvant contenir des instances de lui-même, est susceptible d'implémenter Recursivelterator.

On pense immédiatement au cas du dossier, comportant des dossiers, etc. Nous allons traiter ce cas là.

## II-D - Limitlterator



Cette classe est tout aussi simple, elle prend en paramètre un objet itérable (implémentant Iterator), puis une valeur de départ et une de fin.

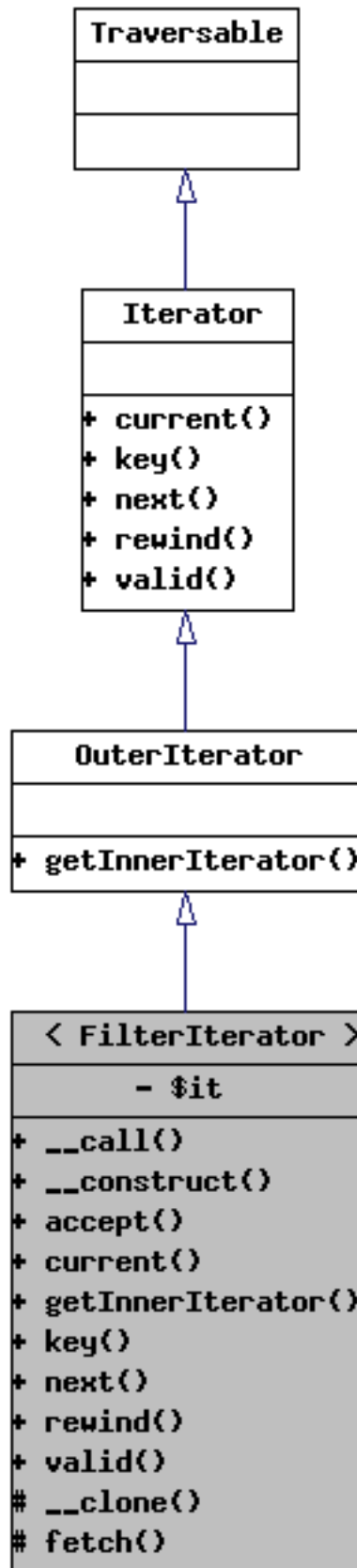
Elle limite alors l'itérateur à certains résultats, tout bêtement, mais cela peut s'avérer plus que pratique :

```
<?php
$itere = new RecursiveIteratorIterator(new mon_array($tablo,1));

$limiteur = new LimitIterator($itere,5,4);
foreach($limiteur AS $v)
{
    echo $v;
}
```

Je vais afficher 8 9 10 u. Je pars du 5ème élément, et j'en affiche 4.

## II-E - FilterIterator



Même principe que LimitIterator, FilterIterator ne va retourner un résultat que si celui-ci est accepté, via une règle, que vous devez définir grâce à la méthode **accept()** FilterIterator est une classe abstraite qui définit une méthode abstraite **accept()**. C'est un proxy qui agrège un itérateur, et va lui faire suivre les demandes.

```
<?php
class mon_array_filtre extends FilterIterator
{
    public function __construct(mon_array $mon_array)
    {
        parent::__construct(new RecursiveIteratorIterator($mon_array));
    }

    public function accept()
    {
        $it = $this->getInnerIterator();
        return strlen($it->current()) == 1 ? true: false;
    }
}

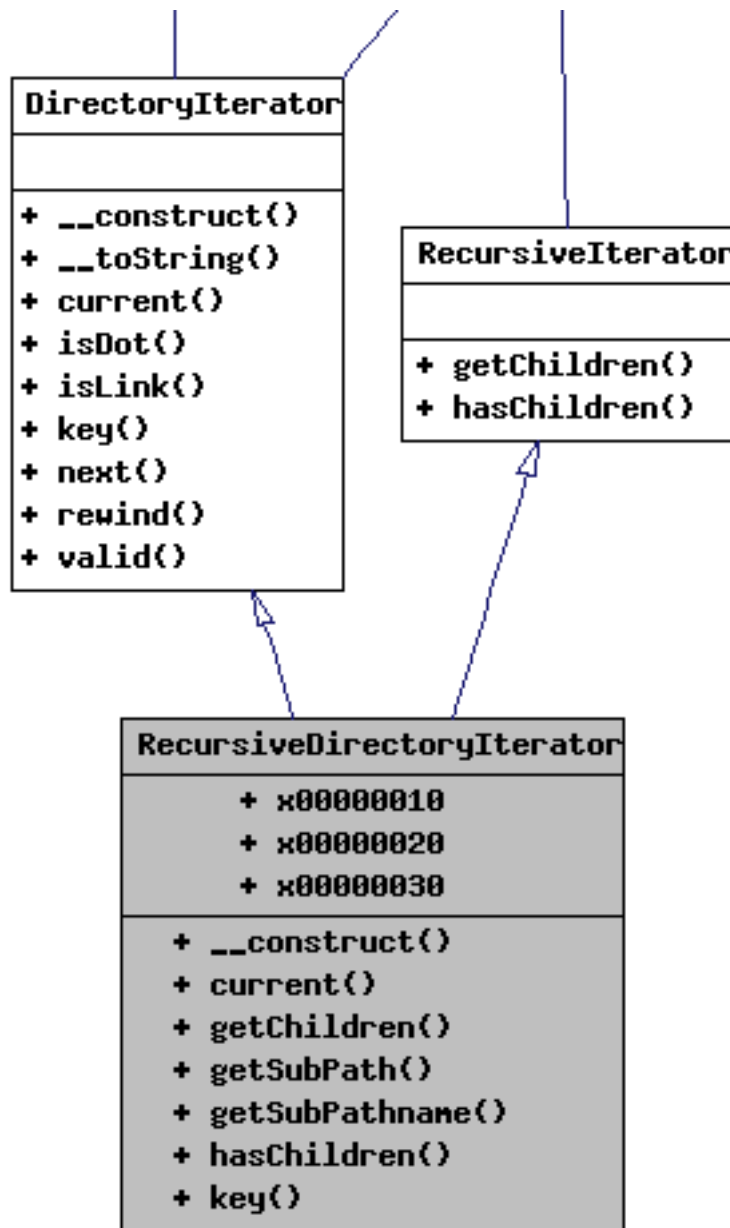
$a = new mon_array_filtre(new mon_array(array('erf', array('r'), 'x', 'y', 'zz')));

foreach($a AS $v)
{
    echo $v;
}
```

Ce petit code va afficher 'rxy'. Je lui ai demandé de n'afficher que les éléments de mon\_array d'une seule lettre.

FilterIterator attend en paramètre un objet implémentant Iterator, c'est le cas de mon\_array que nous avons déjà vu plus haut.

## II-F - RecursiveDirectoryIterator



Revenons à la récursivité : RecursiveDirectoryIterator est une classe (et non une interface), qui permet la récursivité sur le parcours d'un répertoire.

```

<?php
$directory = "path/to/dir";
$iter = new RecursiveIteratorIterator(
    new RecursiveDirectoryIterator($directory, RecursiveDirectoryIterator::KEY_AS_FILENAME),
    RecursiveIteratorIterator::SELF_FIRST);

foreach ($iter as $entry)
{
    if ($entry->isDir())
    {
        $token = "<b>%s</b>";
    }else{
        $token = "%s";
    }
}

```

```

echo str_repeat(" ", 3*$iter->getDepth());
printf($token."<br>", $entry);
}
    
```

Cet exemple crée un arbre au format HTML basique, de tout le dossier voulu, récursivement. Les dossiers le composant sont mis en gras, et la méthode **getDepth()** de l'itérateur permet de renvoyer sous forme d'entier, la profondeur dans laquelle il se trouve actuellement.

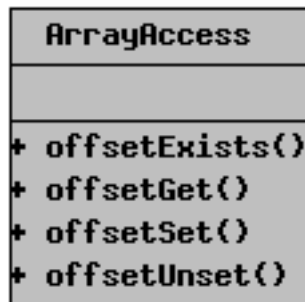
En un mot : merveilleux.

Si vous ne passez pas l'option SELF\_FIRST, alors les dossiers listés, ne seront pas retournés, ce sont les « têtes de noeud » et l'itérateur ne les renvoie pas par défaut. Une fois de plus, la documentation nous en apprend beaucoup sur toutes ces petites options.

Aussi, \$entry est un objet SPLFileInfo et tout un tas de méthodes lui sont attachées

Elle possède aussi un **\_\_toString()**, qui renvoie **getFilename()**, qui elle, renvoie par défaut le chemin complet du fichier, nous avons modifié ce comportement pour obtenir juste le nom du fichier, grâce à une constante passée au constructeur : KEY\_AS\_FILENAME.

## II-G - ArrayAcces



ArrayAcces modifie carrément la manière dont se comportent les [ et ] pour accéder aux tableaux. Un objet peut alors être utilisé presque comme un tableau :

```

<?php
class Calcul implements ArrayAccess
{
    private $_num;
    private $_signe;

    public function __construct($num, $signe)
    {
        $this->_num = (int)$num;
        switch ($signe)
        {
            case "+": case "-": case "*": case "/": case "%":
                break;
            default:
                throw new Exception("Operation non autorisée");
        }
        $this->_signe = $signe;
    }
    public function offsetSet($offset, $value) { }

    public function offsetGet($offset)
    
```

```
{
    return eval("return \$offset \$this->_signe \$this->_num;");
}

public function offsetUnset($offset) { }

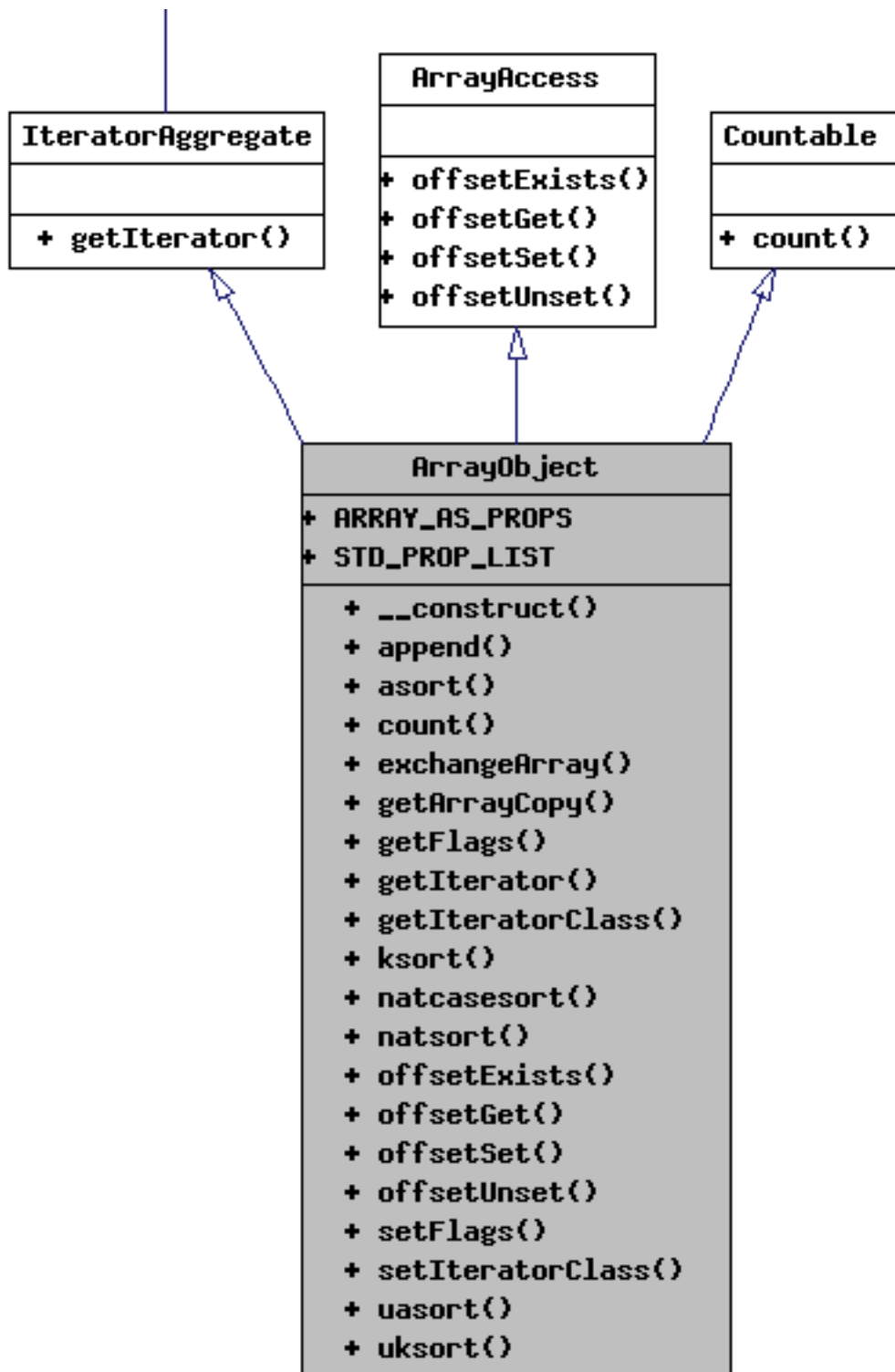
public function offsetExists($offset)
{
    return true;
}
}
$c = new Calcul(10, "+");

echo $c[0]; // 10
echo $c[4]; // 14
echo $c[-10]; // 0
$c[8] = 5;
echo $c[8]; // 18 quand même
```

Si vous connaissez SimpleXML, vous noterez que vous pouvez accéder, via l'objet SimpleXMLElement, aux propriétés des tags xml , avec une syntaxe de tableau. Ca n'est pas magique ! , c'est que tout simplement SimpleXMLElement implémente ArrayAccess (en interne).

Ce comportement (comme tous les autres d'ailleurs) sont très utiles dans l'élaboration de frameworks.

## II-H - ArrayObject



`ArrayObject` est sans doute l'objet le plus connu et le plus utile de la SPL. Pour faire simple : utilisez un tableau comme un objet.

Ou alors utilisez cet objet comme un tableau, complet (par défaut).

Vous pouvez aussi jouer avec son itérateur (interne), et le changer à votre guise.

```
<?php
$lar = new ArrayObject(array('a','b','c'));
$lar['key'] = 'd';

echo $lar->count(); // 4
$oldArray = $lar->exchangeArray(array(1,2,3));
foreach($lar as $value){
echo $value;
}
// affiche alors 123
```

En passant une constante spéciale au constructeur, on peut utiliser alors l'accès objet sur cet ArrayObject, en plus de l'accès tableau classique :

```
<?php
$lar = new ArrayObject(array('a','b','c'),ArrayObject::ARRAY_AS_PROPS);
$lar->key = 'd';

echo $lar->count(); // 4
```

C'est pratique car quelques fois on se retrouve dans un cas où l'analyseur syntaxique nous bloque, pensez à ceci :

```
$o->aMethod()['somekey'];
```

Si aMethod() retourne un array, on ne peut directement accéder à une propriété, sans passer par une valeur intermédiaire. Si on utilise l'ArrayObject correctement, alors on a

```
$o->aMethod()->somekey;
```

Ce qui est syntaxiquement valide.

Il est possible de même, de changer l'itérateur de l'arrayObject, par défaut, il s'agit d'un ArrayIterator, qui hérite entre autres de SeekableIterator (héritant d'Iterator). On peut lui inculquer un RecursiveArrayIterator :

```
<?php
$lar = new ArrayObject(array('a','b','c',array('d')));
$lar['key'] = array('some value','another value');

$lar->setIteratorClass('RecursiveArrayIterator');
$itere = new RecursiveIteratorIterator($lar);
foreach($itere as $value){
echo $value;
}
// affiche : abcdsome valueanother value
```

Définitivement terminés les foreach dans des foreach ^^

### III - Conclusion

Non, PHP ne sera jamais un langage "orienté objet" (tel que java), et restera un langage 'fonctionnel' (basé sur des fonctions dans un contexte global); mais on perçoit nettement depuis PHP5, et les supérieurs (5.1 , 5.2), la volonté des développeurs et contributeurs du PHPGroup de fournir de vraies solutions aux programmeurs Objet, qui sont de plus en plus nombreux, et exigeants.

PHP5 est réellement capable en entreprise, et a atteint un niveau permettant de concevoir des systèmes complexes, et favorisant nettement le travail en équipe. Car en utilisant massivement la SPL, on arrive à un taux de réutilisation de code très élevé, et à une interface commune d'accès « aux données » , au sens le plus large possible; le tout de manière très intuitive et proche de Java, C++ ou C#.

Souvenez vous de la règle du développeur : au moins on en écrit, au plus le risque de bug est faible.

La SPL propose des solutions qui sont intégrées directement dans le moteur Zend Engine 2, et qui font réagir foreach (par exemple) ou même les accesseurs de tableau pourtant banals [ et ].

Elle est écrite en C (en grande partie) et donc compilée : absolument aucun problème de performance; Zend Engine 2 a d'ailleurs été créé dans l'optique de supporter le modèle objet actuel de PHP(5).

De plus, si vous voulez lire sa source, aucun problème, **prenez donc par là** pour une traduction PHP.

PHP est opensource, la consultation des sources est très utile pour la SPL, car certaines sources sont carrément écrites en PHP (le reste en C), il suffit de fouiller le dépôt cvs, branche **php-src/ext/spl/internal**, on voit alors le code(PHP) de IteratorIterator (entres autres).

On voit aussi beaucoup d'autres exemples que l'on peut utiliser ou modifier comme on le sent. Marcus répond aussi aux questions sur la mailing list Internal, de PHP. Remarquez aussi : un PDOStatement, ou un SimpleXMLElement implémentent l'interface « Traversable », l'interface au sommet de la SPL, significative pour le Zend Engine 2. En fait, le PHPGroup utilise la SPL dans certaines extensions. Il existe par exemple un SimpleXMLIterator...

Outre la SPL, `get_declared_classes()`; et `get_declared_interfaces()` vont vous renseigner sur les possibilités objet intégrées à PHP5.

Les objets DateTimezone et DateTime sont très pratiques et se prennent en main rapidement (PHP5.1 minimum)

Ne parlons pas de l'utilité des Exceptions#

D'autres objets sont plus courants, comme les PDO, les DOM, ou encore les SimpleXML.

Enfin, les objets de Reflection, aussi, sont plus que jamais utiles, dans les cas où la documentation venait à être maigre sur certains objets internes de PHP. Quant à l'auto-complétion de code, sous Zend Studio, elle est convenable pour la majorités des cas, quelques rares petites lacunes à noter #

Pour aller plus loin :

#### Utilisation des interfaces en PHP5

#### Introduction aux designs patterns (java)

