

# 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 : 29/07/09


PHP5 possède un modèle objet 'non vide' : il est agrémenté de classes et d'interfaces internes, réunies dans ce qu'on appelle la SPL, ou Standard PHP Library. Nous allons voir en quoi elles peuvent s'avérer très utiles dans des développements orientés objets en PHP, de plus en plus nécessaires de nos jours, afin de maintenir une cohérence logique dans ses programmes.

I - Introduction.....	3
II - Dans le coeur de la SPL.....	5
II-A - Iterator.....	6
II-B - IteratorIterator.....	8
II-C - Countable.....	9
II-D - RecursiveIterator.....	10
II-E - LimitIterator.....	12
II-F - SeekableIterator.....	13
II-G - FilterIterator.....	15
II-H - RegexIterator.....	17
II-I - RecursiveDirectoryIterator.....	19
II-J - AppendIterator.....	20
II-K - ParentIterator.....	20
II-L - CachingIterator.....	21
II-M - SimpleXMLIterator.....	21
II-N - ArrayAccess.....	23
II-O - ArrayObject.....	24
II-P - SplObjectStorage.....	25
II-Q - Fonctions de la SPL.....	27
III - Conclusion.....	28

## 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).

 *Intégrées, plus ou moins : la SPL est en réalité une extension PHP, mais qui est très souvent (pour ne pas dire tout le temps) compilée dans PHP, quelle qu'en soit la distribution (Win, packages Linux...), ce qui donne l'impression de la nativité de ses fonctionnalités. La SPL rend beaucoup de services, et de plus en plus d'extensions PHP en sont dépendantes. Ainsi, à partir de PHP5.3, il ne sera plus possible de désactiver l'intégration de la SPL via la commande de compilation. Elle sera réellement "nativement" intégrée.*

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.

La documentation principale de la SPL sur le site officiel de PHP est incomplète, elle est en cours de rédaction et actuellement beaucoup de classes manquent. Heureusement, le développeur de la SPL a pris soin d'en créer une documentation à part, au format Doxygen, que vous trouverez ici : <http://www.php.net/~helly/php/ext/spl/>

Celle-ci est plus complète, même trop : certaines classes présentes dans cette doc n'existent pas encore dans PHP ! Pensez à vérifier cela au moyen de l'API de Reflection par exemple. Vous trouverez toutes sortes de diagrammes de classe sur cette mini doc, 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 :

### liste des classes de la SPL

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

Il m'en retourne 43 (à la date de cet article). 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 :

 *La SPL est en développement permanent. Mettre à jour sa version de PHP est donc aussi gage d'un meilleur support SPL. Si certains frameworks nécessitent une version de PHP précise comme configuration minimale, c'est en partie à cause du manque de certains objets dans certaines versions de PHP. PHP5.0 ne comptait que 13 classes dans la SPL !*

### liste des interfaces de la SPL

```
<?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 natives à PHP.

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 des éléments contenus dans un autre objet rendu itératif, 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 : il devient alors itératif.

On peut assimiler les itérateurs à des curseurs, dans le cas des bases de données.

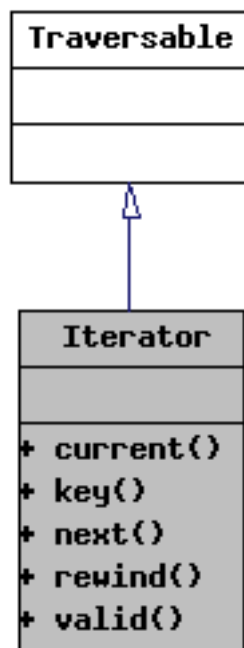
En réalité, PHP utilise lui-même les itérateurs, à chaque fois que vous faites un *foreach()*, ou un *count()*.

La syntaxe PHP du *foreach* est :

```
<?php
foreach ($foo as $key=>$val)
{
    echo "$key vaut $val";
}
```

En interne, il se passe (grossièrement) ceci :

```
<?php
$fooIt = $foo->getIterator();
$fooIt->rewind();
while ($fooIt->valid()) {
    echo "$fooIt->key() vaut $fooIt->current()";
    $fooIt->next();
}
```



Pour un type PHP array, nous avons à disposition des fonctions d'itération : *next()*, *prev()*, *key()*, *current()*... qui en interne vont utiliser les processus d'itération. *foreach* fonctionne sur les objets depuis PHP5, car ils implémentent tous, en interne, l'interface **Traversable**, ce qui signifie qu'ils sont parcourables au moyen d'une structure *foreach*.

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
    
```

Ceci est le comportement par défaut de PHP, l'itérateur récupère les noms des propriétés publiques en clé, et leur valeur en valeur.

## II-A - Iterator

Il est possible de modifier ce comportement par défaut, en implémentant l'interface **Iterator**; rappelez-vous de la syntaxe avec le while situé plus haut :

```

<?php
class MonArray implements Iterator
{
    private $_tab = array();

    private $_pas;

    public function __construct(array $array, $pas = 1)
    {
        $this->_tab = $array;
        $this->_pas = abs((int)$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, ceci permet un chainage des méthodes à l'utilisation)

```
<?php
$tablo = range (0,10);
$mon_array = new MonArray($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 avec un pas que je donne au constructeur de mon objet, dans mon exemple.

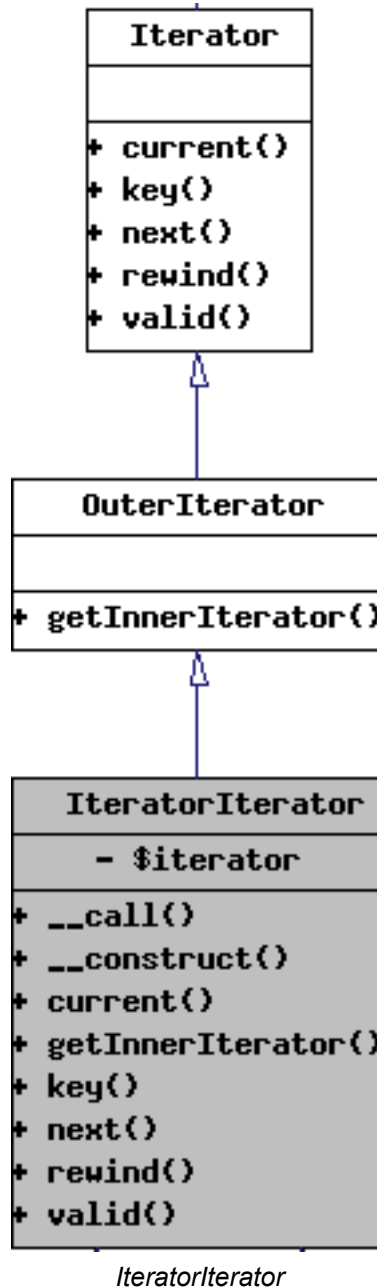
 *Très important : Utiliser l'interface `Iterator` permet de décaler la logique de sélection et de navigation dans les données. L'interface `Iterator` permet au développeur de prendre la main sur le comportement de la structure `foreach`, lorsqu'appliquée à la classe. Le développeur doit donc guider `foreach` au moyen de 5 méthodes.*

*Grâce à `Iterator`, dans le code d'une boucle `foreach` je n'ai plus aucune logique de sélection des données, le `foreach` me retourne **directement** toutes les données auxquelles je m'attends.*

*Vous devez donc faire attention au code dans ces 5 méthodes, par exemple si `valid()` retourne 'true' tout le temps, `foreach` est alors dans une boucle infinie.*

Pour résumer : avec **Iterator**, c'est au développeur de guider tout le comportement d'un futur `foreach` appliqué sur son objet. L'objet en question gagne alors en puissance puisqu'il est capable lui-même de décrire la manière dont il veut qu'on le traverse.

## II-B - IteratorIterator



**IteratorIterator** est un itérateur générique simple permettant d'itérer sur un objet implémentant **Traversable**, et donc à fortiori **Iterator**, qui en hérite. `foreach` réagit à tout ce qui est **Traversable**, et certains objets internes à PHP comme **PDOStatement** implémentent cette interface. On peut donc itérer dessus avec un simple `foreach`, mais si on doit utiliser un itérateur de plus haut niveau (comme ceux dans cet article), il sera nécessaire de transformer la structure traversable, en une structure itérative. **IteratorIterator** existe dans ce but.

La différence entre **Traversable** et **Iterator** est difficile à saisir, elle se situe en interne dans le ZendEngine.

**i** Il n'est pas possible d'un point de vue utilisateur, d'implémenter l'interface **Traversable** directement, vous devez utiliser au minimum **Iterator**

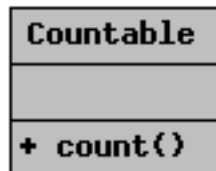
```

<?php
$pdo = new PDO('mysql:host=localhost;dbname=test','logintest','passtest');
$stmt = $pdo->query("SELECT * FROM test");
$it = new IteratorIterator($stmt);
    
```

```
foreach ($it as $val) {
    echo $val;
}
```

Ce code affiche tous les résultats de la requête SQL concernée, mais l'objet **IteratorIterator** n'est pas nécessaire ici. Voyez la section concernant **CachingIterator** pour un exemple plus concret.


## II-C - Countable



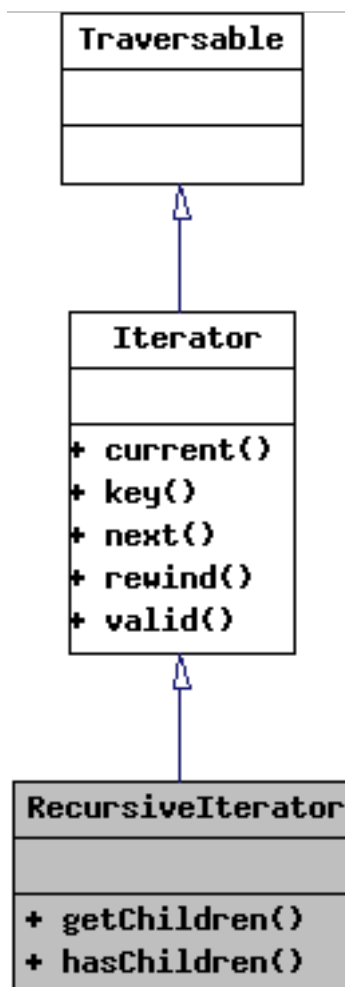
Vous pouvez de même implémenter **Countable**, seule la méthode **count()** devra être définie, et elle va guider le comportement de la fonction **count()** de PHP :

```
<?php
class MonArray 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.

 **A retenir :** **Countable** modifie le comportement par défaut de la fonction **count()** du langage PHP, de la même manière que **Iterator** modifie le comportement de la structure **foreach()** de PHP

## II-D - RecursiveIterator



Grâce à l'interface **RecursiveIterator**, je vais implémenter des méthodes qui vont permettre à l'itérateur d'itérer sur des éléments, de même nature, 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'itérateur **RecursiveIteratorIterator** adapté avec :

```

<?php
class MonArray implements RecursiveIterator
{
    private $_tab = array();
    private $_pas;

    public function __construct(array $array, $pas = 1)
    {
        $this->_tab = $array;
        $this->_pas = abs((int)$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'));
$a = new RecursiveIteratorIterator(new MonArray($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 **MonArray**. 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.

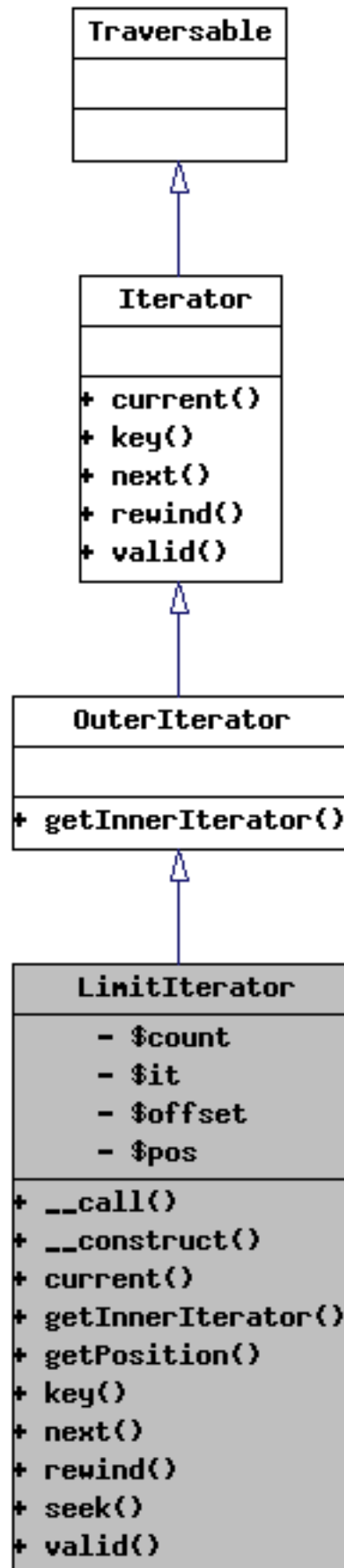
Si true est retourné, alors **RecursiveIteratorIterator** appelle immédiatement **getChildren()**, qui doit lui retourner un objet de la même classe que l'objet couramment itéré (un enfant) à son tour, et il commence l'itération dessus.

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 itératif composite, c'est à dire pouvant être parcouru et pouvant contenir des instances de lui-même, est susceptible d'implémenter **RecursiveIterator**.

On pense immédiatement au cas du dossier, comportant des dossiers, des fichiers, les dossiers étant des sortes de fichiers, etc.

## II-E - LimitIterator



Cet itérateur est tout aussi simple, il prend en paramètre un objet itératif (implémentant **Iterator**), puis une valeur de départ et un nombre de valeurs à parcourir.

Il limite alors l'objet itératif à certains résultats, tout bêtement, mais cela peut s'avérer plus que pratique :

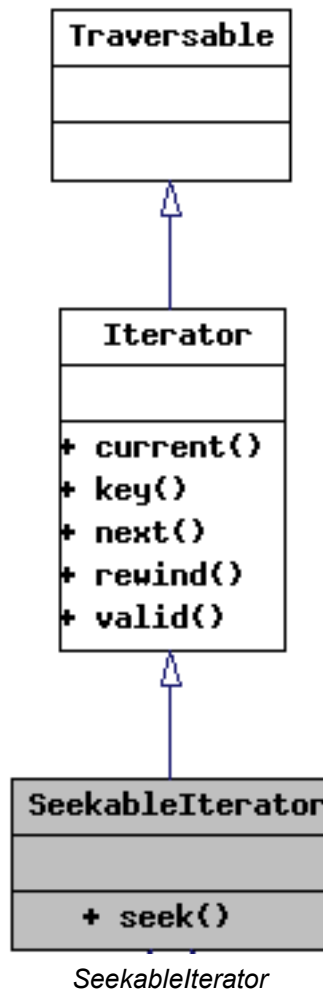
```
<?php
$itere = new RecursiveIteratorIterator(new MonArray($tablo,1));

$limiteur = new LimitIterator($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.

Il est à ce titre impressionnant de remarquer combien les gens essaient de réinventer la roue, en mettant des "if(...)" ou "return" ou je ne sais quoi dans leur boucle foreach, la rendant rapidement illisible.

## II-F - SeekableIterator



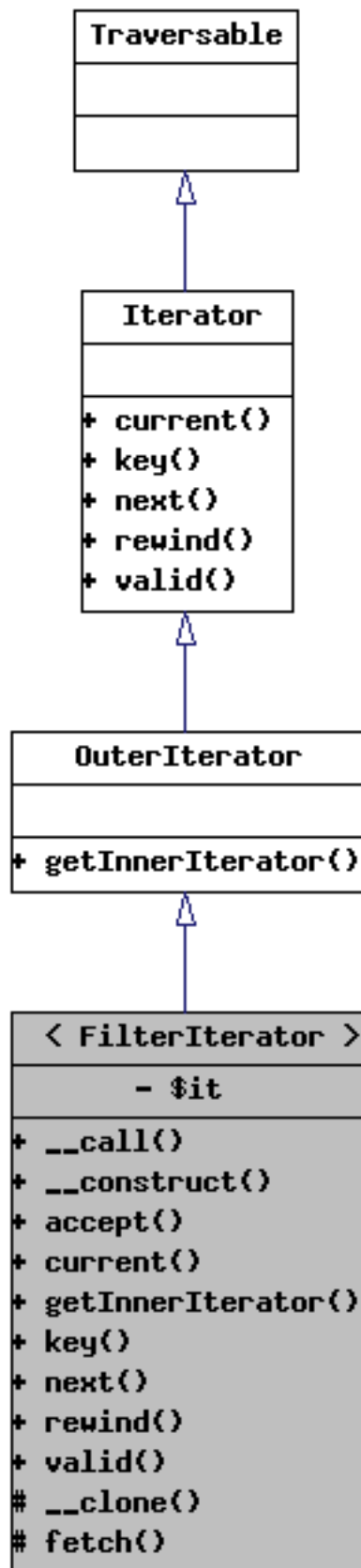
SeekableIterator est un itérateur, mais dans lequel on peut se déplacer pour demander l'accès à l'enregistrement se trouvant à la position X de l'objet itéré.

```
<?php
class MonArray implements SeekableIterator
{
    // même code qu'auparavant
}
```

```
public function seek($index)
{
    if (array_key_exists($index,$this->_tab)) {
        $this->_key = $index;
        return $this->_tab[$index];
    }
}
```

On voit dans ce code que l'on peut déplacer le curseur de l'itérateur à la place que l'on souhaite.

## II-G - 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 MonArrayFiltre extends FilterIterator
{
    public function __construct(MonArray $mon_array)
    {
        parent::__construct(new RecursiveIteratorIterator($mon_array));
    }

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

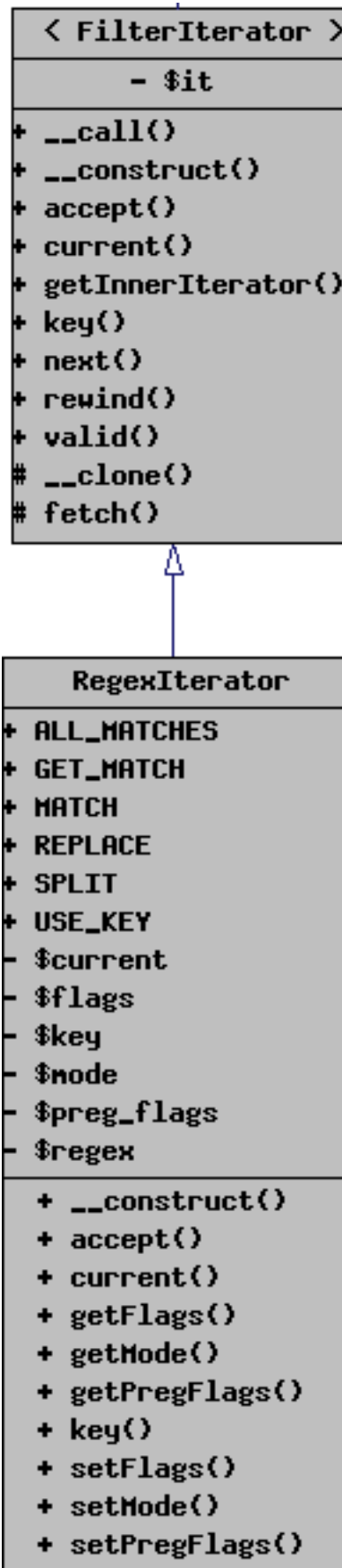
$a = new MonArrayFiltre(new MonArray(array('erf', array('r'), 'x', 'y', 'zz')));

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

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

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

## II-H - RegexIterator



*RegexIterator*

RegexIterator hérite de FilterIterator, et permet d'utiliser la méthode **accept()** avec une expression régulière.

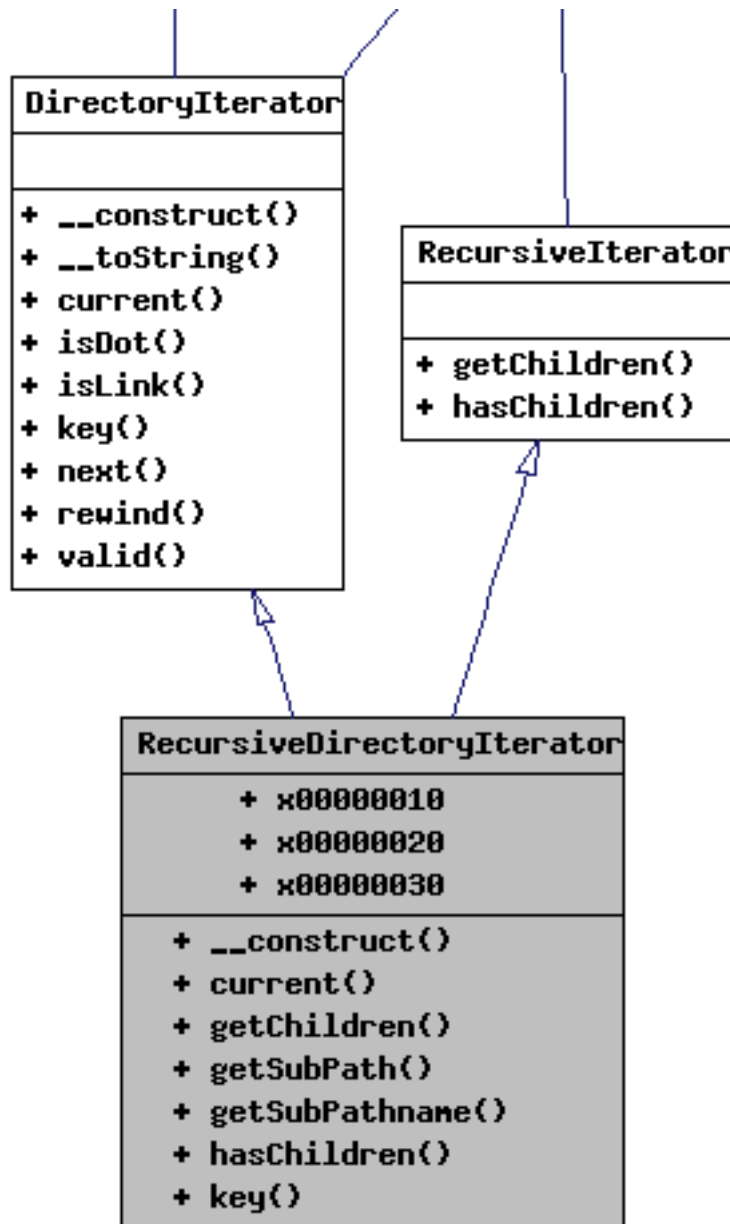
```
<?php
class DefinedFunction extends RegexIterator
{
    public function __construct($regex, $mode = 0, $flags = 0, $preg_flags = 0)
    {
        $phpFunctions = get_defined_functions();

        parent::__construct(new ArrayIterator(array_shift($phpFunctions)), $regex, $mode, $flags, $preg_flags);
    }
}

foreach (new DefinedFunction('/str/') as $key=>$val) {
    echo $val;
}
```

Cet exemple affiche toutes les fonctions PHP qui comprennent la chaîne 'str' dedans. **RegexIterator** permet de définir certains paramètres pour piloter l'expression régulière. \$mode par exemple, peut contenir RegexIterator::GET\_MATCH, qui va alors retourner le tableau \$matches utilisé par *preg\_match()* en dessous, ou encore RegexIterator::SPLIT, qui va ordonner d'utiliser *preg\_split()* en dessous. \$flags peut contenir RegexIterator::USE\_KEY, à ce moment là l'expression régulière est utilisée sur les clés des données d'entrées, et non plus sur les valeurs. \$preg\_flags peut avoir n'importe quelle valeur des constantes globales PHP PREG\_\*

## II-I - RecursiveDirectoryIterator



Revenons à la récursivité : **RecursiveDirectoryIterator** est un itérateur, 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{
        $token = "%s";
    }
    echo str_repeat("&nbsp;", 3*$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 dont **\_\_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-J - AppendIterator

**AppendIterator** sert à fusionner plusieurs objets itératifs en un seul.

AppendIterator
- \$iterators
+ __call() + __construct() + append() + current() + getInnerIterator() + key() + next() + rewind() + valid()

*AppendIterator*

```
<?php
$arr1 = new ArrayIterator(array(1,2,3));
$arr2 = new ArrayIterator(array(4,5,6));

$ai = new AppendIterator();
$ai->append($arr1);
$ai->append($arr2);

foreach ($ai as $value) {
    echo $value; // 123456
}
```

Ici aussi, nul besoin de faire suivre 2 instructions foreach l'une derrière l'autre : la logique de selection des données est déplacée dans l'itérateur.

## II-K - ParentIterator

**ParentIterator** est capable de ne retourner que les éléments ayant un enfants, depuis un **RecursiveIterator** quelconque.

Prenons un exemple avec un **RecursiveArrayIterator** :

```
<?php
```

```

$sarr = array(
    10,
    array(1, 2, 3),
    11,
    array(4, 5, 6),
    12 );
$pi = new ParentIterator(new RecursiveArrayIterator($sarr));

foreach ($pi as $key=>$val) {
    printf("%s - %s", $key, print_r($val, true));
}
/*
1 - Array
(
    [0] => 1
    [1] => 2
    [2] => 3
)
3 - Array
(
    [0] => 4
    [1] => 5
    [2] => 6
)
*/
    
```

## II-L - CachingIterator

Qui n'a jamais souhaité répondre à cette question simple : Je suis dans un itérateur à une position X, "reste-t-il un élément après ma position, ou suis-je le dernier ?" ?

**CachingIterator** sert à répondre à cette question, il prend en paramètre un itérateur, et va le parcourir tout de suite, d'un coup.

En bouclant après sur le **CachingIterator**, on pourra alors savoir s'il reste des résultats, grâce à la méthode **hasNext()**, car celui-ci aura déjà fait le tour de tous les éléments de la structure.

```

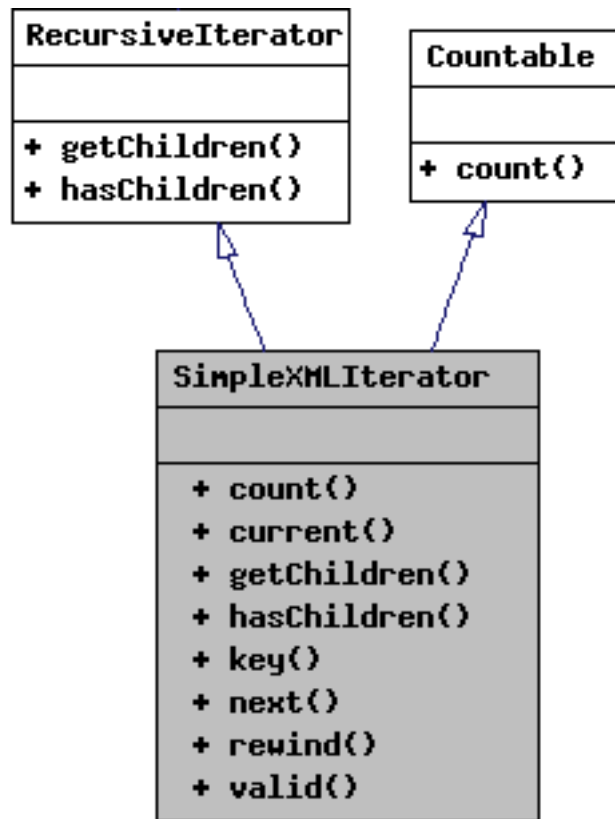
<?php
$pdo = new PDO('mysql:host=localhost;dbname=test', 'logintest', 'passtest');
$stmt = new CachingIterator(new IteratorIterator($pdo->query("SELECT * FROM test")));
$stmt->rewind();
$stmt->next();
$stmt->next();

if ($stmt->hasNext()) {
    echo "il reste des résultats";
}
    
```

Ici, un objet PDOStatement est utilisé (un jeu de résultats de base de données). Le **CachingIterator** permet de savoir s'il reste des résultats après avoir avancé 2 fois le pointeur interne.

## II-M - SimpleXMLIterator

Cet objet est lié à simpleXML, et est automatiquement instancié lors de la création d'un noeud de **SimpleXMLElements**. Cependant, on peut l'instancier à part, en lui passant une chaîne représentant du XML valide. Associé à un **RecursiveIteratorIterator**, on peut alors afficher le contenu de tous les noeuds facilement :



*SimpleXMLIterator*

```

<?php
$xml = <<<EOF
<doc>
<a>hello</a>
<a>
<b>world</b>
<c />
<d>
<e>coucou</e>
</d>
</a>
<c>developpez</c>
</doc>
EOF;

$xml = new RecursiveIteratorIterator(new SimpleXMLIterator($xml));
foreach ($xml as $val) {
    printf("%s - ", $val); // hello world coucou developpez
}
    
```

Evidemment on peut le lier avec d'autres itérateurs.

## II-N - ArrayAccess

ArrayAccess
+ offsetExists()
+ offsetGet()
+ offsetSet()
+ offsetUnset()

L'interface **ArrayAccess** modifie carrément la manière dont se comportent les "[" et "]" pour accéder aux tableaux. Un objet peut alors être utilisé quasiment comme un tableau :

```
<?php
class FunctionProxy implements ArrayAccess
{
    const SEPARATOR = '_';
    private $_func;

    public function offsetGet($offset)
    {
        $this->_func = $offset;
        return $this;
    }

    public function offsetSet($offset,$value)
    {
        throw new Exception('Affectation interdite');
    }

    public function offsetExists($offset)
    {
        return ($this->_func == $offset);
    }

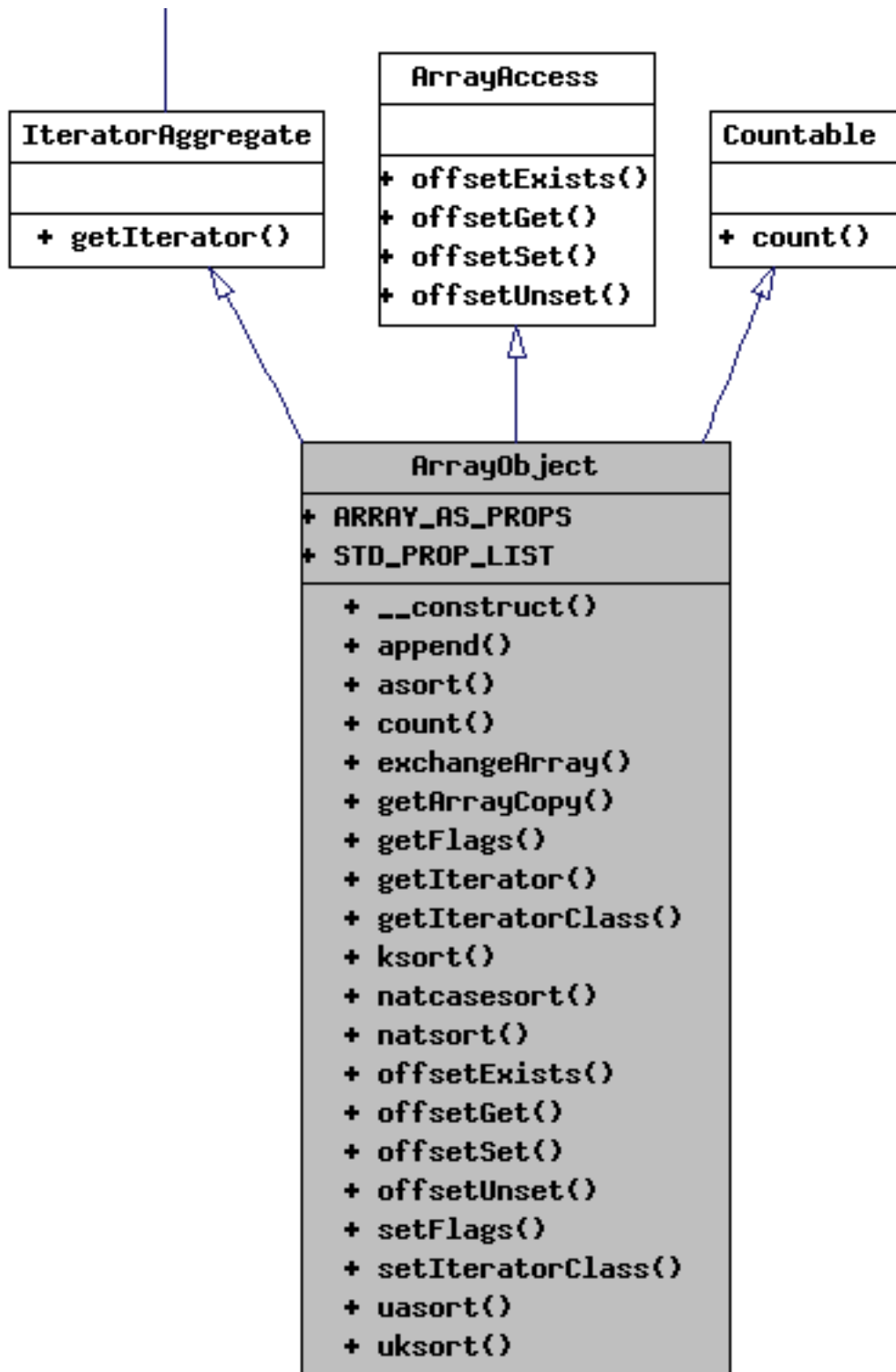
    public function offsetUnset($offset)
    {
        throw new Exception('Déréférencement interdit');
    }

    public function __call($func,$args)
    {
        $func = $this->_func.self::SEPARATOR.$func;
        if (!function_exists($func)) {
            throw new Exception("PHP function $func doesn't exist");
        }
        return @call_user_func_array($func,$args);
    }
}
$p = new functionProxy();
$p['array']->combine($array1, $array2); // proxy vers la fonction PHP array_combine()
$p['str']->replace(/* ... */); // proxy vers la fonction PHP str_replace()
```

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-O - 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
```

```

$ar = new ArrayObject(array('a', 'b', 'c'));
$ar['key'] = 'd';

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

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

```

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

echo $ar->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
$ar = new ArrayObject(array('a', 'b', 'c', array('d')));
$ar['key'] = array('some value', 'another value');

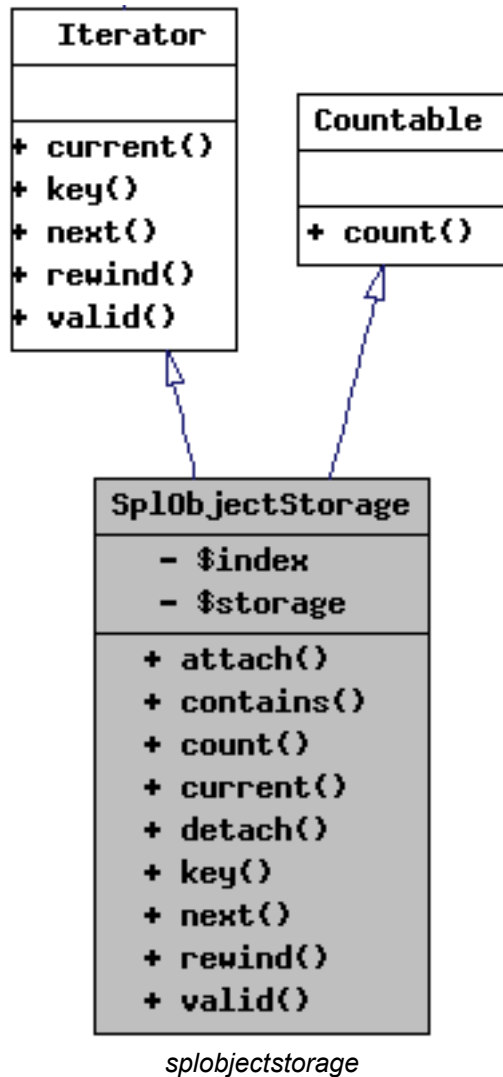
$ar->setIteratorClass('RecursiveArrayIterator');
$itere = new RecursiveIteratorIterator($ar);
foreach($itere as $value) {
    echo $value;
}
// affiche : abcddsome valueanother value
    
```

Définitivement terminés les foreach dans des foreach ^^

## II-P - SplObjectStorage

`SplObjectStorage` est un conteneur itératif d'objets. Son avantage est qu'il s'occupe de stocker et de retrouver un objet, sans que l'on ait besoin de s'occuper de cela.

Lorsqu'un objet contient des instances d'autres objets, `SplObjectStorage` est l'endroit idéal pour les stocker. Plus simple qu'un tableau ou un `ArrayObject`, ce support de stockage s'occupe notamment d'éviter de stocker 2 mêmes instances, et leur suppression est très simple :



```

<?php
class Panier extends SplObjectStorage
{
    private $_items;

    public function attach($i)
    {
        if (!$i instanceof Item) {
            throw new Exception('Item requis');
        }
        return parent::attach($i);
    }

    public function detach($i)
    {
        if (!$i instanceof Item) {
            throw new Exception('Item requis');
        }
        return parent::detach($i);
    }


    public function calculTotal()
    {
        foreach ($this as $item) {
            $somme[] = $item->price;
        }
        return array_sum($somme);
    }
}
    
```

La méthode ***attach()*** se charge de ne pas stocker deux mêmes instances (aucune exception n'est retournée). Quant à ***detach()***, elle se charge de retrouver l'objet qu'on lui passe en paramètre (s'il existe) pour le supprimer du conteneur. Cette classe est bien plus pratique qu'il n'y paraît, elle peut même servir de support pour un design pattern registre.

## II-Q - Fonctions de la SPL

L'extension SPL rajoute des classes et des interfaces à PHP, mais pas seulement : quelques fonctions existent aussi. La plus intéressante reste sans aucun doute *iterator\_to\_array()*, qui prend en paramètre un itérateur et le parcourt totalement pour sortir ses résultats sous forme de tableau. En gros : ceci évite un foreach inutile juste pour regarder les résultats :

```
$it = new ArrayIterator(array('foo'=>'bar', 'baz'=>'developpez', 'julien'=>'pauli'));  
var_dump(iterator_to_array($it));
```

Suivant le même exemple : *iterator\_count(\$iterator)* compte le nombre d'éléments dans la structure itérative. *spl\_object\_hash()* retourne un identifiant unique pour un objet. 2 mêmes instances mémoires auront les mêmes identifiants. Extrêmement pratique pour stocker un objet et générer une "clé de rangement". *spl\_autoload\_register()* permet de  **gérer une pile d'autoloads** (fonctions étant chargées de charger une classe de manière automatique)

### 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, 5.3), 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 svn, 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...

Nous n'avons pas tout vu sur la SPL, mais c'est un bon début pour commencer, donc usez et abusez de cette extension qui permet de créer des structures réellement agréables à manipuler et à maintenir.

PHP 5.3 introduit lui aussi de nouveaux objets SPL, souvenez-vous que rester à jour sur la version de PHP est là aussi gage de qualité de vos programmes, car plus d'objets disponibles, et moins buggés.

Pour aller plus loin :

**Utilisation des interfaces en PHP5**

**Introduction aux designs patterns (java)**  **PHP5 : Design Pattern observateur aidé de la Standard PHP Library (SPL)**  **POO PHP5 : Créer un agrégateur à base de réflexion et de SPL**