|  |
| --- |
| **Association pour la RÉadaptation et la Formation Professionnelle** |
| **Conception & Développement Informatique** |
|  |
| PHP – Programmation Orientée Objet |
| ***PHP, Concept Objet, UML*** |

|  |
| --- |
| **Mickaël DEVOLDÈRE** |
| MD v1.0.1 |
| **17/04/2018** |
| <http://www.arfp.asso.fr> |
| La reproduction partielle ou intégrale du présent document sur un support, quel qu’il soit, est formellement interdite sans l’accord écrit et préalable du Centre de Réadaptation de Mulhouse. |

PHP – Programmation Orientée Objet

PHP, Concept Objet, UML

|  |  |
| --- | --- |
| Technologies et langages | |
|  | HTML |
|  | CSS |
|  | Javascript |
|  | PHP |

|  |  |
| --- | --- |
| Légende des icônes | |
|  | Information complémentaire |
|  | Point d’attention particulier |
|  | Intervention du formateur possible |
|  | Lien vers une ressource externe |

Sommaire

[Pourquoi faire de l'objet ? 3](#_Toc511739127)

[Classes et instanciation 3](#_Toc511739128)

[Méthodes et attributs statiques 4](#_Toc511739129)

[Héritage 4](#_Toc511739130)

[Classe mère 5](#_Toc511739131)

[Classes abstraites 5](#_Toc511739132)

[Méthodes et classes finales 6](#_Toc511739133)

[Interfaces 6](#_Toc511739134)

[Exceptions 7](#_Toc511739135)

[Try/Catch 7](#_Toc511739136)

[Exception personalisée 8](#_Toc511739137)

[Remarques 9](#_Toc511739138)

[Problèmes fréquents 9](#_Toc511739139)

[Références 9](#_Toc511739140)

[Attention aux références 10](#_Toc511739141)

[Clonage 11](#_Toc511739142)

[Clonage personnalisé 11](#_Toc511739143)

[Substitution 12](#_Toc511739144)

[Type hinting 12](#_Toc511739145)

[Espaces de nom 13](#_Toc511739146)

[Multiples classes de même nom 14](#_Toc511739147)

[Pour aller plus loin 15](#_Toc511739148)

[Test d'instance 15](#_Toc511739149)

[Sérialisation 16](#_Toc511739150)

[Les méthodes magiques 17](#_Toc511739151)

[L'autoloader 17](#_Toc511739152)

[Historique du document 18](#_Toc511739153)

[Crédits 18](#_Toc511739154)

# Pourquoi faire de l'objet ?

L'objet est un paradigme de programmation très répandu et qui a fait ses preuves dans de nombreux projets. Son utilisation n'apporte pas de fonctionnalités au langage, c'est à dire que tout ce que l'on peut faire en utilisant la pogramation orientée objet peut être fait sans, cependant l'objet apporte beaucoup de choses en simplicité de compréhension, maintenance, factorisation et découpage de code, travail collaboratif ou encore en conception.

Toutes ces qualité font de l'objet un mécanisme indispensable à maîtriser pour tout développeur PHP.

Presque toutes les bibliothèques et frameworks que vous serez amenés à utiliser se basent sur le paradigme objet.

# Classes et instanciation

En PHP, voici à quoi ressemble une classe:

<?php

class User

{

protected $name = null;

public function \_\_construct($name)

{

$this->name = $name;

}

public function sayHello()

{

echo 'Hello, I am '.$this->name."!\n";

}

}

***Remarquez que:***

Les attributs peuvent être initialisés directement dans leur définition

Les modifieurs private , protected et public sont présents, comme dans beaucoup d'autre

langages

Le constructeur se définit à l'aide de la fonction magique \_\_construct()

Les attributs et méthodes de classes sont accessibles par l'opérateur -> , le point étant réservé pour la

concaténation de chaines.

Un objet de cette classe s'instanciera alors de la manière suivante:

<?php

$user = new User('Bob');

$user->sayHello();

## Méthodes et attributs statiques

En PHP, il est possible de rendre des méthodes et des attributs statiques à l'aide du modifieur « static » :

<?php

class Example

{

public static $counter = 0;

public $number;

public function \_\_construct()

{

$this->number = ++self::$counter;

}

}

$a = new Example;

echo $a->number."\n"; // 1

$b = new Example;

echo $b->number."\n"; // 2

Les attributs et méthodes statiques ne sont pas spécifiques à une instance mais globaux.

Dans l'exemple ci-dessus,l'attribut $counter n'est pas répété dans **$a** et dans **$b** mais n'est présent qu'une seule fois, ce qui explique que les valeurs sont différentes à l’affichage.

## Héritage

L'héritage s'écrit avec avec le mot clé « extends » :

<?php

class A

{

public $a = 12;

}

class B extends A // La classe B hérite de A

{

public $b = 34;

}

$b = new B;

echo $b->a, "\n"; // 12

echo $b->b, "\n"; // 34

## Classe mère

L'accès aux méthodes et aux attributs de la classe mère peut se faire à l'aide du mot clé parent :

<?php

class Rectangle

{

protected $width;

protected $height;

public function \_\_construct($width, $height)

{

$this->width = $width;

$this->height = $height;

}

}

class Square extends Rectangle

{

public function \_\_construct($width)

{

parent::\_\_construct($width, $width);

}

}

## Classes abstraites

PHP vous permet de déclarer des classes ou des méthodes comme abstraites à l'aide du mot clé abstract . Si au

moins une méthode d'une classe est abstraite, ou que la classe est marquée elle même comme abstraite, elle ne

pourra pas être instanciée:

<?php

abstract class Message

{

abstract public function getName();

abstract public function getBody();

public function display()

{

echo 'From: '.$this->getName()."\n";

echo 'Contents: '.$this->getBody()."\n";

}

}

$m = new Message(); // Erreur

## Méthodes et classes finales

Il est possible d'utiliser le mot clé « final » sur une classe ou une méthode, afin d'en empêcher l'héritage:

<?php

class A

{

public final function f()

{

return 42;

}

}

class B extends A

{

public function f()

{

return 30; // Erreur

}

}

# Interfaces

En PHP, les interfaces se déclarent comme une classe à l'aide du mot clé « interface » , elles ne contiennent que

des prototypes de méthodes. Une classe peut implémenter une interface avec la notation :

« implements nomInterface »:

<?php

interface CanSpeak

{

public function speak();

}

class Human implements CanSpeak

{

public function speak()

{

echo "I am Human!\n";

}

}

$human = new Human();

$human->speak();

# Exceptions

Comme la plupart des langages orienté objet, PHP propose un mécanisme d'exceptions permettant d'affiner la

gestion d'erreur. Par défaut, les exceptions remonteront jusqu'à être disposée sous forme d'erreur:

<?php

throw new Exception('Erreur, ça a planté !');

// Donnera lieu à :

*PHP Fatal error: Uncaught exception 'Exception' with message 'Erreur, ça a planté !' in index.php:2*

*Stack trace:*

*#0 {main}*

*thrown in index.php on line 2*

## Try/Catch

Il est possible de capturer les exceptions grâce aux mots clés try et catch :

<?php

try

{

throw new Exception('Bad');

}

catch (Exception $e)

{

echo 'Erreur: ' .$e->getMessage() . "\n";

}

## Exception personalisée

PHP vous offre également la possibilité de surcharger les classes d'exception, dont Exception est la "racine"

pour créer vos propres types d'exceptions:

<?php

class MyException extends Exception

{

}

try

{

throw new MyException();

}

catch (MyException $my)

{

echo "MyException\n";

}

catch (Exception $e)

{

echo "Exception\n";

}

Comme vous pouvez le constater, les exceptions peuvent être capturées avec un certain ordre de priorité.

# Remarques

**Il n'y a pas d'héritage multiple en PHP.**

PHP ne supporte pas la surcharge, méthodes ayant le même nom mais des prototypes différents, vous pouvez cependant utiliser des paramètres optionnels et non typés, voici un exemple illustrant un argument optionel ayant une valeur par défaut:

<?php

class A

{

public function f($x = 42)

{

echo "x = $x\n";

}

}

$a = new A;

$a->f(); // x = 42

$a->f(67); // x = 67

# Problèmes fréquents

## Références

Lorsque l'on passe un objet en argument d'une fonction, on ne passe pas une copie de cette objet mais une référence vers l'objet (à ne pas confondre avec une référence vers la variable qui décrit l'objet). Ainsi, toute

modification se fera directement sur l'objet:

<?php

class A

{

public $attr = 1;

}

function func($a)

{

$a->attr = 2;

}

$a = new A();

func($a);

echo $a->attr; // 2

## Attention aux références

Attention à ne pas confondre référence vers un objet et référence entre les variables, regardons l'exemple suivant:

<?php

class A

{

public $attr = 1;

}

$a = new A;

$b = $a;

$b->attr = 2;

echo $a->attr; // 2

$b = null;

echo gettype($a); // object

$c = &$a;

$c = null;

echo gettype($a); // null

Dans ce cas, la ligne **$b = $a** fait en sorte que la variable **$b** référence le même objet que **$a** . Ainsi la

modification de l'attribut sur **$b->attr** est aussi visible sur **$a->attr** . En revanche, la variable **$b** est

bien différente de **$a** , c'est pourquoi l'affecter à null ne change nullement la valeur de **$a** ; En revanche,

l'utilisation de l'opérateur de référence & pour créer la variable **$c** fait en sorte que **$c** soit un alias de **$a** , il

référencera alors non pas seulement le même objet mais aussi la même variable.

# Clonage

Si vous souhaitez créer une copie d'un objet, vous pouvez utiliser le mécanisme de clonage de cet objet. PHP vous

propose pour cela d'utiliser le mot clé clone .

<?php

class A

{

public $attr = 1;

}

$a = new A;

$a->attr = 5;

$b = clone $a;

$b->attr = 6;

echo $a->attr."\n"; // 5

echo $b->attr."\n"; // 6

## Clonage personnalisé

Son comportement peut cependant être non trivial et soulève souvent des questions: Faut t-il cloner également les objets référencés ? Est-ce que toute les propriétés doivent être clonées ? Pour répondre à ces questions, il vous est possible d'écrire votre propre méthode de clonage, avec le nom "magique" \_\_clone() :

<?php

class Identified

{

public static $instances = 0;

public $instance;

public function \_\_construct()

{

$this->instance = ++self::$instances;

}

public function \_\_clone()

{

$this->instance = ++self::$instances;

}

}

$a = new Identified();

$b = clone $a;

echo $a->instance."\n"; // 1

echo $b->instance."\n"; // 2

# Substitution

PHP étant interprété, les types ne sont connus qu'au moment de l'execution. Ainsi, lorsque vous écrivez une

méthode, les paramètres ne sont pas typés. Cela peut s'avérer pratique pour la substitution, mais aussi provoquer

des problèmes très innatendus:

<?php

class A

{

public $attr = 1;

}

function f($a)

{

echo $a->attr;

}

$a = new A; // $a est une instance de la classe A

f($a); // 1

$a = [12]; // $a est maintenant un array

f($a); // Erreur (La fonction f($a) utilise $a comme un objet alors que $a est un array.)

## Type hinting

Depuis PHP 5.3, un mécanisme permet d'éviter ce genre d'erreur fréquente (passage d'argument du mauvais type), il s'agit du *type hinting* (ou indication de type):

<?php

function f(A $a)

{

echo $a->attr."\n";

}

// Si l'argument passé en paramètre n'est pas du type A, une erreur claire sera levée dès l'appel à la méthode

Le type indiqué dans les paramètres de la fonction peut être le type de la classe mère ou d'une interface qui doit être implémentée par l'objet passé. Il est fortement recommandé de mettre une indication de type le plus souvent possible dans vos prototype de fonctions et de méthodes afin d'éviter les erreurs obscures qui peuvent survenir lors du passage d'un objet du mauvais type.

# Espaces de nom

Souvent, la création de classes et d'interface engendre un problème de nommage, car il peut devenir difficile d'éviter les problèmes de collisions de noms (deux classes ayant le même nom). Depuis PHP 5.3, il est possible d'utiliser des espaces de nom (ou namespace ) pour éviter ce problème.

Par exemple, si le fichier alice/image.php contient:

<?php

namespace Alice;

class Image

{

// Code de la classe

}

On pourra l'utiliser comme cela:

<?php

include('alice/image.php');

use Alice\Image;

$image = new Image;

Ainsi, la classe de Alice ne "pollue" pas l'espace de nom global mais est disponible sous Alice\Image , si quelqu'un d'autre souhaite écrire un classe de gestio n d'images, il pourra le faire en utilisant un autre espace de nom.

# Multiples classes de même nom

Si Bob écrit à son tour une classe Image et la place sous l'espace de noms Bob\Image , il sera possible d'utiliser

les deux soit à l'aide de la déclaration entière du nom des classes.

<?php

$a = new Alice\Image;

$b = new Bob\Image;

Il est également possible d'importer une classe à l'aide du mot clé use , par défaut, le nom de la classe (ici, Image) sera un raccourci vers son emplacement complet (ici, Alice\Image ):

<?php

use Alice\Image;

$a = new Image;

$b = new Bob\Image;

Enfin, le mot clé as permet de donner un nom de substitution (ou alias) à la classe dans le fichier courant:

<?php

use Bob\Image as BobImage;

use Alice\Image as AliceImage;

$a = new AliceImage;

$b = new BobImage;

# Pour aller plus loin

## Test d'instance

Il est possible de tester qu'un objet est bien l'instance d'une classe en PHP à l'aide du mot clé instanceof :

<?php

interface P {};

class A {};

class B extends A {};

class Q implements P {};

$a = new A;

$b = new B;

$q = new Q;

var\_dump($a instanceof A); // true

var\_dump($b instanceof A); // true

var\_dump($a instanceof B); // false

var\_dump($q instanceof A); // false

var\_dump($q instanceof P); // true

Notez que si l'objet testé est l'instance d'une classe fille de la classe passée, **instanceof** retournera vrai, comme

par exemple pour l'expression **$b instanceof A** ci-dessus.

Ce système fonctionne également pour tester si un objet implémente une interface, comme avec

**$q instanceof P** ci-dessus.

## Sérialisation

Contrairement aux types "basiques" (nombres, chaînes, tableaux...), les objets peuvent s'avérer complexes à

représenter sous forme de chaîne de caractère pour être sauvegardé dans un fichier, un cookie ou encore une

variable de session par exemple.

Pour cela, vous pouvez utiliser la sérialisation. Les fonctions PHP serialize() et unserialize() permettent de représenter un objet sous forme de chaîne de caractères et, inversement, d'obtenir un objet à partir d'une chaîne sérialisée:

<?php

class A

{

public $attr = 0;

}

if (file\_exists('a.txt'))

{

$a = unserialize(file\_get\_contents('a.txt'));

}

else

{

$a = new A;

}

$a->attr++;

echo $a->attr."\n";

file\_put\_contents('a.txt', serialize($a)); // écrit dans un fichier

## Les méthodes magiques

Il existe en PHP des méthodes magiques. Ces dernières peuvent par exemple permettre de surcharger l'accès à un

Attribut ou une méthode même s'il/elle n'existe pas:

|  |  |
| --- | --- |
| Nom | Utilité |
| \_\_get($name) | Apellée lors de l'accès en lecture à un attribut non-existant. |
| \_\_set($name, $value) | Apellée lors de l'accès en écriture à un attribut non-existant. |
| \_\_call($method, $args) | Appelée lors d'un appel à une méthode non existante. |

## L'autoloader

L'autoloading est un mécanisme apparu dans PHP 5.3 qui permet d'exécuter du code au moment ou une classe est demandée et qu'elle n'est pas chargée dans le but de la charger dynamiquement.

Prenez quelques minutes pour lire la documentation officielle à ce sujet : [spl\_autoload\_register()](http://php.net/manual/fr/function.spl-autoload-register.php)

--- Fin du document ---

# Historique du document

|  |  |  |
| --- | --- | --- |
| Auteur | Date | Observations |
| Mickaël DEVOLDÈRE | 17/04/2018 | Création du document |
|  |  |  |
|  |  |  |

# Crédits

<http://php.net>

<http://gregwar.com/php/programmation-orientee-objet.html>