Programmation Orientée Objet (suite) en PHP

PWS

Pierre Blarre - Florian Rodriguez

Namespace

- Dans un projet, il est possible d'avoir des classes qui portent le même nom. (librairie, framework, etc.) Pour éviter les conflits, on utilise les **espaces de noms**.
- Les namespaces permettent de définir un nom de «package», une sorte de dossier virtuel qui contient des classes.
- Les namespaces sont déclarés avec le mot-clé `namespace`.

Namespace

- Par exemple:

```php
<?php
namespace MonNamespace\MonSousNamespace;
class MaClasse
{
}

$objet = new MonNamespace\MonSousNamespace\MaClasse;
```

- On peut aussi utiliser le mot-clé `use` pour importer un namespace.

```php
<?php
use MonNamespace\MonSousNamespace\MaClasse;
$objet = new MaClasse;
```

- **namespace**, permet de dire "je travaille dans ce dossier"
- **use**, permet d'importer une class d'un autre "dossier"

- Si on souhaites charger une classe de PHP après avoir déclaré un namespace, il faudra alors préfixer la classe par un antislash `\`.

```php
<?php
namespace MonNamespace;
$objet = new \PDO;
```

Un objet, un identifiant

Quand on fait:

```php
$objet = new MaClasse;
```

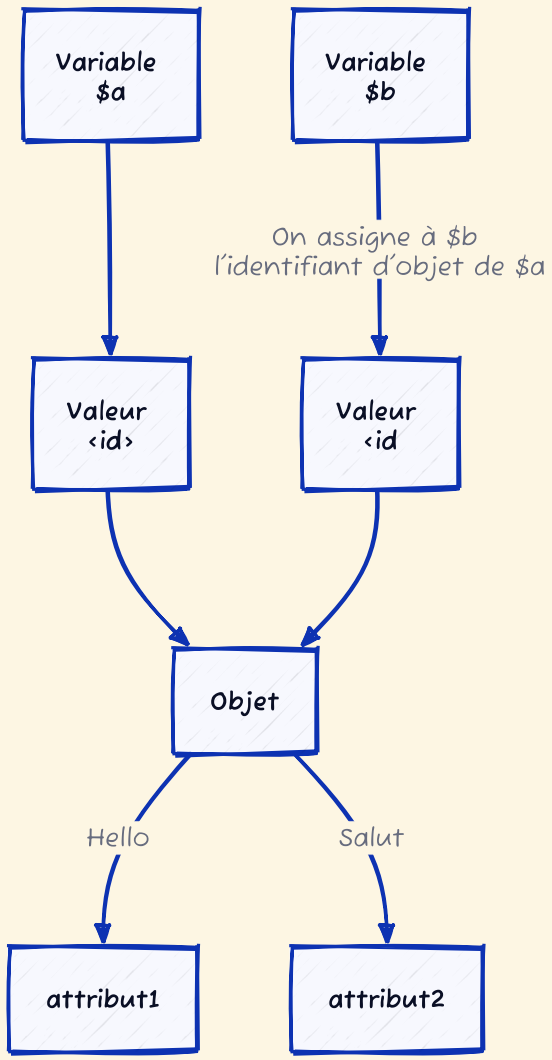
la variable `$objet` ne contient pas l'objet à proprement parler.

Quand on instancie une classe, la variable stockant l'objet ne stocke en fait pas l'objet lui-même, mais un **identifiant** qui représente cet objet.

```php
<?php
class MaClasse
{
    public $attribut1;
    public $attribut2;
}

$a = new MaClasse;

$b = $a; // On assigne à $b l'identifiant de $a, donc $a

$a->attribut1 = 'Hello';
echo $b->attribut1; // Affiche Hello.

$b->attribut2 = 'Salut';
echo $a->attribut2; // Affiche Salut.
```

- $a contient l'identifiant représentant l'objet créé.
- On assigne à $b la valeur de $a.

$a et $b font donc référence à la même instance.



Clone

- En réalité, il n'y a qu'un seul objet et qu'un seul identifiant
- Deux variables contenant le même identifiant d'objet.
- Lorsqu'un objet est passé en paramètre à une fonction ou renvoyé par une autre, on ne passe pas une copie de l'objet mais une copie de son *identifiant*
- `$objet1 = $objet2` ne permet pas de "copier un objet". Pour copier toutes les valeurs d'un objet, il faut utiliser le *clonage* d'objet :

```php
<?php
$copie = clone $origine; // On copie le contenu de l'objet $origine dans l'objet $copie.
```

- Les deux objets contiennent des identifiants différents

Clone

- Lorsque l’on clone un objet, la méthode magique `__clone` du nouvel objet sera appelée (si elle est définie). (Au même titre que `__construct` lorsque l’on appelle `new`)
- On ne peut pas appeler cette méthode directement.
- C'est la méthode `__clone` du nouvel objet créé qui est appelée, pas la méthode `__clone` de l'objet à cloner.
- On peut utiliser cette méthode pour modifier certains attributs pour le nouvel objet, ou alors incrémenter un compteur d'instances par exemple.

Clone - exemple

```php
<?php
class MaClasse
{
  private static $instances = 0; //variable statique, valeur commune à tous les objets de cette classe

  public function __construct()
  {
    self::$instances++;
  }

  public function __clone()
  {
    self::$instances++;
  }

  public static function getInstances()
  {
    return self::$instances;
  }
}

$a = new MaClasse;
$b = clone $a;

echo 'Nombre d\'instances de MaClasse : ', MaClasse::getInstances();
```

```
Nombre d'instances de MaClasse : 2
```

Comparaison des objets

```php
<?php
if ($objet1 == $objet2)
{
    echo '$objet1 et $objet2 sont identiques !';
}
else
{
    echo '$objet1 et $objet2 sont différents !';
}
```

- Cette partie n'explique pas comment comparer des objets mais la démarche que PHP exécute pour les comparer.
- Pour que la condition renvoie `true`, il faut que :
  - `$objet1` et `$objet2` aient les mêmes attributs et les mêmes valeurs.
  - `$objet1` et `$objet2` soient des instances de la même classe.

Comparaison des objets - ==

Exemple :

```php
<?php
class A
{
    public $attribut1;
    public $attribut2;
}
class B
{
    public $attribut1;
    public $attribut2;
}

$a = new A;
$a->attribut1 = 'Hello';
$a->attribut2 = 'Salut';

$b = new B;
$b->attribut1 = 'Hello';
$b->attribut2 = 'Salut';

$c = new A;
$c->attribut1 = 'Hello';
$c->attribut2 = 'Salut';

if ($a == $b) ? echo '$a == $b' : echo '$a != $b';
echo ' <br/>';
```

Quel est le résultat de ce code ?

- **A** `$a == $b <br/> $a == $c`
- **B** `$a != $b <br/> $a == $c`
- **C** `$a != $b <br/> $a != $c`

A | B | C

Responses: 0

Comparaison des objets - ===

```php
<?php
class A
{
    public $attribut1;
    public $attribut2;
}

$a = new A;
$a->attribut1 = 'Hello';
$a->attribut2 = 'Salut';

$b = new A;
$b->attribut1 = 'Hello';
$b->attribut2 = 'Salut';

$c = $a;

if ($a === $b) ? echo '$a === $b' : echo '$a !== $b';

echo '<br />';

if ($a === $c) ? echo '$a === $c' : echo '$a !== $c';
```

Quel est le résultat de ce code ?

- **A** `$a === $b <br/> $a === $c`
- **B** `$a !== $b <br/> $a !== $c`
- **C** `$a !== $b <br/> $a === $c`

A B C

Responses: 0

Parcours des objets

- Parcourir un objet consiste à lire tous les attributs **visibles** de l'objet.
- On peut utiliser la même boucle que pour parcourir un tableau: **foreach**
- Syntaxe identique :
  - **foreach ($objet as $valeur)** : `$valeur` sera la valeur de l'attribut actuellement lu.
  - **foreach ($objet as $attribut => $valeur)** : `$attribut` aura pour valeur le nom de l'attribut actuellement lu et `$valeur` sera sa valeur.

Parcours des objets - exemple

```php
<?php
class MaClasse
{
    public $attribut1 = 'Premier attribut public';
    public $attribut2 = 'Deuxième attribut public';

    protected $attributProtege1 = 'Premier attribut protégé';
    protected $attributProtege2 = 'Deuxième attribut protégé';

    private $attributPrive1 = 'Premier attribut privé';
    private $attributPrive2 = 'Deuxième attribut privé';

    function listeAttributs()
    {
        foreach ($this as $attribut => $valeur)
        {
            echo '<strong>', $attribut, '</strong> => ', $valeur, '<br />';
        }
    }
}


class Enfant extends MaClasse
{
    function listeAttributs() { // Redéclaration de la fonction
        foreach ($this as $attribut => $valeur)
```

Parcours des objets - exemple

```php
$classe = new MaClasse;
$enfant = new Enfant;

echo '---- Liste les attributs depuis l\'intérieur de la clas
$classe->listeAttributs();

echo '<br />---- Liste les attributs depuis l\'intérieur de l
$enfant->listeAttributs();

echo '<br />---- Liste les attributs depuis le script global

foreach ($classe as $attribut => $valeur)
{
    echo '<strong>', $attribut, '</strong> => ', $valeur, '<b
}
```

Récapitulatif

- Une variable ne contient jamais d'objet à proprement parler, mais leurs *identifiants*
- Pour dupliquer un objet, l'opérateur = n'a pas l'effet désiré : il faut **cloner** l'objet grâce à l'opérateur `clone`.
- Pour comparer deux objets, l'opérateur `==` vérifie que les deux objets sont issus de la même classe et que les valeurs de chaque attribut sont identiques, tandis que l'opérateur `===` vérifie que les deux identifiants d'objet sont les mêmes.
- Il est possible de parcourir un objet grâce la structure **foreach**

Les interfaces

- Permettent **d'imposer une structure** aux classes, c'est-à-dire d'obliger certaines classes à implémenter certaines méthodes.
- Le rôle d'une interface:
  - Décrire un comportement à notre objet.
  - Les interfaces ne doivent pas être confondues avec l'héritage.
  - Exemple : une voiture et un personnage n'ont aucune raison d'hériter d'une même classe. Par contre, une voiture et un personnage peuvent tous les deux se déplacer, donc une interface représentant ce point commun pourra être créée.
- Exemple

```php
<?php
interface Movable
{
    public function move($dest);
}
```

- Toutes les méthodes présentes dans une interface doivent être publiques.
- Une interface ne peut pas lister de méthodes abstraites ou finales.
- Une interface ne peut pas avoir le même nom qu'une classe et vice-versa.
- On implémente une interface grâce au mot-clé `implements`.

Les interfaces - exemple

```php
<?php
interface Movable
{
    public function move($dest);
}

class Personnage implements Movable
{

}
```

Dans ce cas, si on instancie Personnage :

```
PHP Fatal error:  Class Personnage contains 1 abstract method and must therefore be declared abstract or implement the
```

`Personnage` n'a pas implémenté la méthode présente dans l'interface `Movable`

Les interfaces - héritage et implémentations multiples

- Si héritage + interface, alors il faut d'abord spécifier la classe à hériter avec le mot-clé `extends`,
- puis les interfaces à implémenter avec le mot-clé `implements`
- Possible d'implémenter plusieurs interfaces par classe, à condition que celles-ci n'aient pas de méthodes portant le même nom.

```php
<?php
interface iA
{
    public function test1();
}

interface iB
{
    public function test2();
}

class A implements iA, iB
{
    // Pour ne générer aucune erreur, il va falloir écrire les méthodes de iA et de iB.
    public function test1()
    {
    }

    public function test2()
    {
    }
}
```

Les constantes d'interfaces

- Les constantes d'interfaces fonctionnent exactement comme les constantes de classes. Elles ne peuvent être écrasées par des classes qui implémentent l'interface. Exemple :

```php
<?php
interface iInterface
{
    const MA_CONSTANTE = 'Hello !';
}

echo iInterface::MA_CONSTANTE; // Affiche Hello !

class MaClasse implements iInterface
{

}

echo MaClasse::MA_CONSTANTE; // Affiche Hello !
```

Hériter ses interfaces

- Comme pour les classes, on peut hériter des interfaces grâce à l'opérateur `extends`. On ne peut pas réécrire ni une méthode ni une constante qui a déjà été listée dans l'interface parente :

```php
<?php
interface iA
{
    public function test1();
}

interface iB extends iA
{
    public function test1 ($param1, $param2); // Erreur fatale : impossible de réécrire cette méthode.
}

interface iC extends iA
{
    public function test2();
}

class MaClasse implements iC
{
    // Pour ne générer aucune erreur, on doit écrire les méthodes de iC et aussi de iA.
    public function test1()
    {
    }

    public function test2()
    {
    }
```

- Contrairement aux classes, les interfaces peuvent hériter de plusieurs interfaces à la fois:

```php
<?php
interface iA
{
    public function test1();
}

interface iB
{
    public function test2();
}

interface iC extends iA, iB
{
    public function test3();
```

Les interfaces - récap

- Les interfaces permettent d'imposer une structure
- Mot-clé: `implements`
- Si héritage + interface, alors il faut d'abord spécifier la classe à hériter avec le mot-clé `extends`, puis les interfaces à implémenter avec le mot-clé `implements`
- Les interface peuvent hériter de plusieurs autres interfaces entre elles, contrairement à l'héritage entre classes.

Les exceptions

- On a vu les erreurs fatales, les alertes, les erreurs d'analyse ou encore les notices.
- Il est aussi possible de créer nos propres types d'erreurs, les exceptions.
- La gestion d'erreurs est importante sur un site pour identifier et gérer les problèmes.

Lancer une exception

```php
<?php
function additionner($a, $b)
{
    if (!is_numeric($a) || !is_numeric($b))
    {
        // On lance une nouvelle exception grâce à throw et on instancie un objet de la classe Exception.
        throw new Exception('Les deux paramètres doivent être des nombres');
    }

    return $a + $b;
}

echo additionner(12, 3), '<br />';
echo additionner('azerty', 54), '<br />';
echo additionner(4, 8);
```

```
15
Fatal error: Uncaught Exception 'Exception' with message 'Les deux paramètres doivent être des nombres' in Standard inpu
```

Attraper une exception

```php
<?php
function additionner($a, $b)
{
    if (!is_numeric($a) || !is_numeric($b))
    {
        throw new Exception('Les deux paramètres doivent être des nombres');
    }

    return $a + $b;
}

try // Nous allons essayer d'effectuer les instructions situées dans ce bloc.
{
    echo additionner(12, 3), '<br />';
    echo additionner('azerty', 54), '<br />';
    echo additionner(4, 8);
}

catch (Exception $e) // Nous allons attraper les exceptions "Exception" s'il y en a une qui est levée.
{
    echo 'Une exception a été lancée. Message d\'erreur : ', $e->getMessage();
}

echo 'Fin du script'; // Ce message s'affiche, ça prouve bien que le script est exécuté jusqu'au bout.
```

```
15
Une exception a été lancée. Message d'erreur : Les deux paramètres doivent être des nombresFin du script
```

- La troisième instruction du bloc `try` n'a pas été exécutée.
- La deuxième instruction a interrompu la lecture du bloc.
- Si on intercepte les exceptions, le script global, lui, n'est pas interrompu.

Des exceptions spécialisées

- La classe `Exception` est la classe de base pour toute exception qui doit être lancée.
- On peut lancer n'importe quelle autre instance d'une classe, du moment qu'elle hérite de la classe `Exception`

```php
<?php
class Exception
{
  protected $message = 'exception inconnu'; // Message de l'exception.
  protected $code = 0; // Code de l'exception défini par l'utilisateur.
  protected $file; // Nom du fichier source de l'exception.
  protected $line; // Ligne de la source de l'exception.

  final function getMessage(); // Message de l'exception.
  final function getCode(); // Code de l'exception.
  final function getFile(); // Nom du fichier source.
  final function getLine(); // Ligne du fichier source.
  final function getTrace(); // Un tableau de backtrace().
  final function getTraceAsString(); // Chaîne formattée de trace.

  /* Remplacable */
  function __construct ($message = NULL, $code = 0);
  function __toString(); // Chaîne formatée pour l'affichage.
}
```

- On a accès aux attributs protégés de la classe et on peut réécrire les méthodes `__construct` et `__toString`.
- Toutes les autres méthodes sont finales, pas le droit de les réécrire.

Des exceptions spécialisées - exemple

```php
<?php
class MonException extends Exception
{
    public function __construct($message, $code = 0)
    {
        parent::__construct($message, $code);
    }

    public function __toString()
    {
        return $this->message;
    }
}
function additionner($a, $b)
{
    if (!is_numeric($a) || !is_numeric($b))
    {
        throw new MonException('Les deux paramètres doivent être des nombres'); // On lance une exception "MonException
    }

    return $a + $b;
}

try // Nous allons essayer d'effectuer les instructions situées dans ce bloc.
{
    echo additionner(12, 3), '<br />';
```

```
15
Les deux paramètres doivent être des nombres
Fin du script
```

Emboîter plusieurs blocs catch

```php
<?php
function additionner($a, $b)
{
    if (!is_numeric($a) || !is_numeric($b))
    {
        // On lance une exception "MonException".
        throw new MonException('Les deux paramètres doivent être des nombres');
    }

    if (func_num_args() > 2)
    {
        // Cette fois-ci, on lance une exception "Exception".
        throw new Exception('Pas plus de deux arguments ne doivent être passés à la fonction');
    }

    return $a + $b;
}
try // Nous allons essayer d'effectuer les instructions situées dans ce bloc.
{
    echo additionner(12, 3), '<br />';
    echo additionner(15, 54, 45), '<br />';
}

catch (MonException $e) // Attrape les exceptions "MonException" s'il y en a une qui est levée.
{
    echo '[MonException] : ', $e; // Affiche le message d'erreur grâce à la méthode __toString écrite
```

```
15
[Exception] : Pas plus de deux arguments ne doivent être passés à la fonction
Fin du script
```

Exemple concret : la classe PDOException

```php
<?php
try
{
    $db = new PDO('mysql:host=localhost;dbname=tests', 'root', ''); // Tentative de connexion.
    echo 'Connexion réussie !'; // Si la connexion a réussi, alors cette instruction sera exécutée.
}

catch (PDOException $e) // On attrape les exceptions PDOException.
{
    echo 'La connexion a échoué.<br />';
    echo 'Informations : [', $e->getCode(), '] ', $e->getMessage(); // On affiche le n° de l'erreur ainsi que le messag
}
```

Exécuter un code même si l'exception n'est pas attrapée

```php
  <?php
$db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');

try
{
  // Quelques opérations sur la base de données
}
finally
{
  echo 'Action effectuée quoi qu\'il arrive';
}
```

En résumé

- Une exception est une erreur que l'on peut attraper grâce aux mots-clé `try` et `catch`.
- Une exception est une erreur que l'on peut personnaliser, que ce soit au niveau de son affichage ou au niveau de ses renseignements (fichier concerné par l'erreur, le numéro de la ligne, etc.).
- Une exception se lance grâce au mot-clé `throw`.
- Il est possible d'hériter des exceptions entre elles.
- Il existe un certain nombre d'exceptions déjà disponibles dont il ne faut pas hésiter à se servir pour respecter la logique du code.

Les Traits

- Depuis PHP 5.4
- Intègre un moyen de réutiliser le code d'une méthode dans deux classes indépendantes.
- Permet ainsi de repousser les limites de l'héritage simple (pour rappel, en PHP, une classe ne peut hériter que d'une seule classe mère).

Le problème

- Deux classes `Writer` et `Mailer`: La première est chargée d'écrire du texte dans un fichier, tandis que la seconde envoie un texte par mail.
- On veut formater le texte en HTML.
- On va devoir effectuer la même opération (celle de formater en HTML) dans deux classes complètement différentes et indépendantes.

```php
<?php
class Writer
{
    public function write($text)
    {
        $text = '<p>Date : '.date('d/m/Y').'</p>'."\n".'<p>'.nl2br($text).'</p>';
        file_put_contents('fichier.txt', $text);
    }
}

<?php
class Mailer
{
    public function send($text)
    {
        $text = '<p>Date : '.date('d/m/Y').'</p>'."\n".'<p>'.nl2br($text).'</p>';
        mail('login@fai.tld', 'Test avec les traits', $text);
    }
}
```

- Ici, le code est petit et la duplication n'est pas énorme, mais elle est belle et bien présente. Il faut l'imaginer sur des projets conséquents.
- Les traits sont un moyen d'externaliser du code.

Les traits - exemple

```php
<?php
trait MonTrait
{
    public function hello()
    {
        echo 'Hello world !';
    }
}

class A
{
    use MonTrait;
}

class B
{
    use MonTrait;
}

$a = new A;
$a->hello(); // Affiche « Hello world ! ».

$b = new B;
$b->hello(); // Affiche aussi « Hello world ! ».
```

- Un trait est une "mini-classe".
- L'utilisation d'un trait dans une classe se fait grâce au mot-clé `use`

Les traits - exemple du formatage HTML

```php
<?php
trait HTMLFormater
{
    public function format($text)
    {
        return '<p>Date : '.date('d/m/Y').'</p>'."\n".'<p>'.nl2br($text).'</p>';
    }
}

class Writer
{
    use HTMLFormater;

    public function write($text)
    {
        file_put_contents('fichier.txt', $this->format($text));
    }
}

class Mailer
{
    use HTMLFormater;

    public function send($text)
    {
        mail('login@fai.tld', 'Test avec les traits', $this->format($text));
```

Utilisation de plusieurs traits:

```php
<?php
trait HTMLFormater
{
    public function formatHTML($text)
    {
        return '<p>Date : '.date('d/m/Y').'</p>'."\n".'<p>'.nl2br($text).'</p>';
    }
}

trait TextFormater
{
    public function formatText($text)
    {
        return 'Date : '.date('d/m/Y')."\n".$text;
    }
}

class Writer
{
    use HTMLFormater, TextFormater;

    public function write($text)
    {
        file_put_contents('fichier.txt', $this->formatHTML($text));
    }
}
```

Résolution des conflits

- Si des traits ont des méthodes similaires -> Erreur fatale: « Trait method format has not been applied, because there are collisions with other trait methods »
- Pour pallier ce problème: priorité à une méthode

```php
<?php
class Writer
{
    use HTMLFormater, TextFormater
    {
        HTMLFormater::format insteadof TextFormater;
    }

    public function write($text)
    {
        file_put_contents('fichier.txt', $this->format($text));
    }
}
```

Le trait plus fort que la mère

```php
<?php
trait MonTrait
{
    public function speak()
    {
        echo 'Je suis un trait !';
    }
}

class Mere
{
    public function speak()
    {
        echo 'Je suis une classe mère !';
    }
}

class Fille extends Mere
{
    use MonTrait;
}

$fille = new Fille;
$fille->speak(); // Affiche « Je suis un trait ! »
```

```
Je suis un trait !
```

Méthodes de traits vs. méthodes de classes

- La classe plus forte que le trait

```php
<?php
trait MonTrait
{
    public function sayHello()
    {
        echo 'Hello !';
    }
}

class MaClasse
{
    use MonTrait;

    public function sayHello()
    {
        echo 'Bonjour !';
    }
}

$objet = new MaClasse;
$objet->sayHello(); // Affiche « Bonjour ! ».
```

Plus loin avec les traits

- Définition d'attributs

```php
<?php
trait MonTrait
{
    protected $attr = 'Hello !';
    public function showAttr()
    {
        echo $this->attr;
    }
}

class MaClasse
{
    use MonTrait;
}

$fille = new MaClasse;
$fille->showAttr();
```

- Une propriété de trait peut être statique. Mais attention, dans ce cas, chaque classe utilisant ce trait aura une instance indépendante de cette propriété.

Conflit entre attributs

```php
<?php
trait MonTrait
{
    protected $attr = 'Hello !';
}

class MaClasse
{
    use MonTrait;

    protected $attr = 'Hello !'; // Lèvera une erreur
    private $attr = 'Hello !'; // Lèvera une erreur
}
```

```php
<?php
trait A
{
    public function saySomething()
    {
        echo 'Je suis le trait A !';
    }
}

trait B
{
    use A;

    public function saySomething()
    {
        echo 'Je suis saySomething() le trait B !';
    }

    public function saySomethingElse()
    {
        echo 'Je suis le trait B !';
    }
}

class MaClasse
{
```

```
Je suis saySomething() le trait B !Je suis le trait B !
```

Changer la visibilité et le nom des méthodes

```php
<?php
trait A
{
    public function saySomething()
    {
        echo 'Je suis le trait A !';
    }
}

class MaClasse
{
    use A
    {
        saySomething as protected;
    }
}

$o = new MaClasse;
$o->saySomething(); // Lèvera une erreur fatale car on tente d'accéder à une méthode protégée.
```

```php
<?php
trait A
{
    public function saySomething()
    {
        echo 'Je suis le trait A !';
    }
}

class MaClasse
{
    use A
    {
        saySomething as sayWhoYouAre;
    }
}

$o = new MaClasse;
$o->sayWhoYouAre(); // Affichera « Je suis le trait A ! »
$o->saySomething(); // Affichera « Je suis le trait A ! »
```

```php
<?php
trait A
{
    public function saySomething()
    {
        echo 'Je suis le trait A !';
    }
}

class MaClasse
{
    use A
    {
        saySomething as protected sayWhoYouAre;
    }
}

$o = new MaClasse;
$o->saySomething(); // Affichera « Je suis le trait A ! ».
$o->sayWhoYouAre(); // Lèvera une erreur fatale, car l'alias créé est une méthode protégée.
```

Méthodes abstraites dans les traits

```php
<?php
trait A
{
    abstract public function saySomething();
}

abstract class Mere
{
    use A;
}

// Jusque-là, aucune erreur n'est levée.

class Fille extends Mere
{
    // Par contre, une erreur fatale est ici levée, car la méthode saySomething() n'a pas été implémentée.
}
```

En résumé

- Les traits sont un moyen pour éviter la duplication de méthodes.
- Un trait s'utilise grâce au mot-clé `use`
- Il est possible d'utiliser une infinité de traits dans une classe en résolvant les conflits éventuels avec `insteadof`
- Un trait peut lui-même utiliser un autre trait.
- Il est possible de changer la visibilité d'une méthode ainsi que son nom grâce au mot-clé as.

L'API de réflexivité

- Permet d'obtenir des informations sur les classes qui sont manipulées, par exemple: Savoir si telle classe possède telle méthode ou tel attribut

```php
<?php
$magicien = new Magicien(['nom' => 'vyk12', 'type' => 'magicien']);
$classeMagicien = new ReflectionObject($magicien);

if ($classeMagicien->hasProperty('magie'))
{
    echo 'La classe Magicien possède un attribut $magie';
}
else
{
    echo 'La classe Magicien ne possède pas d\'attribut $magie';
}
if ($classeMagicien->hasMethod('lancerUnSort'))
{
    echo 'La classe Magicien implémente une méthode lancerUnSort()';
}
else
{
    echo 'La classe Magicien n\'implémente pas de méthode lancerUnSort()';
}
if ($classePersonnage->hasConstant('NOUVEAU'))
{
    echo 'La classe Personnage possède une constante NOUVEAU';
}
else
{
```

Les relations entre classes

- L'héritage

```php
<?php
$classeMagicien = new ReflectionClass('Magicien');

if ($parent = $classeMagicien->getParentClass())
{
    echo 'La classe Magicien a un parent : il s\'agit de la classe ', $parent->getName();
}
else
{
    echo 'La classe Magicien n\'a pas de parent';
}

<?php
if ($classeMagicien->isSubclassOf('Personnage'))
{
    echo 'La classe Magicien a pour parent la classe Personnage';
}
else
{
    echo 'La classe Magicien n\'a la classe Personnage pour parent';
}
```

```php
<?php
$classePersonnage = new ReflectionClass('Personnage');



// Est-elle abstraite ?
if ($classePersonnage->isAbstract())
{ echo 'La classe Personnage est abstraite'; }
else
{ echo 'La classe Personnage n\'est pas abstraite'; }



// Est-elle finale ?
if ($classePersonnage->isFinal())
{ echo 'La classe Personnage est finale'; }
else
{ echo 'La classe Personnage n\'est pas finale'; }
```

```php
<?php
$classeIMagicien = new ReflectionClass('iMagicien');

if ($classeIMagicien->isInterface())
{
    echo 'La classe iMagicien est une interface';
}
else
{
    echo 'La classe iMagicien n\'est pas une interface';
}


<?php
if ($classeMagicien->implementsInterface('iMagicien'))
{
    echo 'La classe Magicien implémente l\'interface iMagicien';
}
else
{
    echo 'La classe Magicien n\'implémente pas l\'interface iMagicien';
}
```

Récapitulatif

- Nombreuses possibilités de réflexivité.
- Important de savoir que cette API existe et qu'il est possible d'obtenir des informations sur les classes.
    - Possible d'obtenir des informations sur ses classes, attributs et méthodes, respectivement grâce à ReflectionClass, ReflectionProperty et ReflectionMethod.
    - Utiliser des annotations sur ses classes permet, grâce à une bibliothèque telle que addendum, de récupérer dynamiquement leurs contenus.
    - L'utilisation d'annotations dans un but de configuration dynamique est utilisée par certains frameworks ou ORM tel que Doctrine par exemple, ce qui permet d'économiser un fichier de configuration en plaçant la description des tables et des colonnes directement dans des annotations.

Résolution statique à la volée

Cette section va vous montrer une possibilité intéressante de la POO en PHP : la résolution statique à la volée.

C'est une notion un peu complexe à comprendre au premier abord, alors n'hésitez pas à relire plusieurs fois.

Effectuons d'abord un petit flash-back sur `self::`.

Vous vous souvenez à quoi il sert ? À appeler un attribut, une méthode statique ou une constante **de la classe dans laquelle est contenu self::**.

Ainsi, si vous testez ce code :

```php
<?php
class Mere
{
    public static function lancerLeTest()
    {
        self::quiEstCe();
    }

    public static function quiEstCe()
    {
        echo 'Je suis la classe <strong>Mere</strong> !'
    }
}

class Enfant extends Mere
{
    public static function quiEstCe()
    {
        echo 'Je suis la classe <strong>Enfant</strong>
    }
}
Enfant::lancerLeTest();
```

Quel est le résultat de ce code ?

- **A** `Je suis la classe *Mere* !`
- **B** `Je suis la classe *Enfant* !`

A | B

Responses: 0

Résultat :

```php
<?php
class Mere
{
    public static function lancerLeTest()
    {
        self::quiEstCe();
    }

    public static function quiEstCe()
    {
        echo 'Je suis la classe <strong>Mere</strong> !'
    }
}

class Enfant extends Mere
{
    public static function quiEstCe()
    {
        echo 'Je suis la classe <strong>Enfant</strong>
    }
}
Enfant::lancerLeTest();
```

```
Je suis la classe <strong>Mere</strong> !
```

- *Mais qu'est-ce qu'il s'est passé ???*
  - appel de la méthode `lancerLeTest` de la classe Enfant
  - la méthode n'a pas été réécrite, on va donc « chercher » la méthode lancerLeTest de la classe mère
  - appel de la méthode `quiEstCe` de la classeMere.
- *Pourquoi c'est la méthode quiEstCe de la classe parente qui a été appelée ?*
- *Pourquoi pas celle de la classe fille puisqu'elle a été récrite ?*
- Tout simplement parce que `self::` fait appel à la méthode statique de la classe dans laquelle est contenu `self::`, donc de la classe parente.
- *Et la résolution statique à la volée dans tout ça ?*

Static:: à la rescousse

- Tout tourne autour de l'utilisation de `static::`.
- `static::` a exactement le même effet que `self::`, à l'exception près que `static::` appelle l'élément de la classe qui est appelée **pendant l'exécution**.
- C'est-à-dire que si j'appelle la méthode `lancerLeTest` depuis la classe Enfant et que dans cette méthode j'utilise `static::` au lieu de `self::`, c'est la méthode `quiEstCe` de la classe Enfant qui sera appelée et non de la classe Mere !
- Pour bien comprendre, voici un exemple :

```php
<?php
class Mere
{
    public static function lancerLeTest()
    {
        static::quiEstCe();
    }

    public static function quiEstCe()
    {
        echo 'Je suis la classe <strong>Mere</strong> !';
    }
}

class Enfant extends Mere
{
    public static function quiEstCe()
    {
        echo 'Je suis la classe <strong>Enfant</strong> !';
    }
}
Enfant::lancerLeTest();
```

```
Je suis la classe <strong>Enfant</strong> !
```

Contexte statique

- Notez que tous les exemples ci-dessus utilisent des méthodes qui sont appelées dans un contexte statique. J'ai fait ce choix car pour ce genre de tests, il était inutile d'instancier la classe, mais sachez bien que la résolution statique à la volée a exactement le même effet quand on crée un objet puis qu'on appelle une méthode de celui-ci.
- Il n'est donc pas du tout obligatoire de rendre les méthodes statiques pour pouvoir y placer `static::`.
- Ainsi, si vous testez ce code, à l'écran s'affichera la même chose que précédemment.

```php
<?php
class Mere
{
    public function lancerLeTest()
    {
        static::quiEstCe();
    }

    public function quiEstCe()
    {
        echo 'Je suis la classe <strong>Mere</strong> !';
    }
}

class Enfant extends Mere
{
    public function quiEstCe()
    {
        echo 'Je suis la classe <strong>Enfant</strong> !';
    }
}
$enfant = new Enfant;
$enfant->lancerLeTest();
```

```
Je suis la classe <strong>Enfant</strong> !
```

Cas complexes

```php
<?php
class A
{
  public static function appelerQuiEstCe(){
    static::quiEstCe();
  }
  public static function quiEstCe(){
    echo 'A';
  }
}

class B extends A
{
  public static function test(){
    // parent::quiEstCe();
    parent::appelerQuiEstCe();
  }
  public static function quiEstCe(){
    echo 'B';
  }
}

class C extends B
{
  public static function quiEstCe(){
    echo 'C';
```

Quel est le résultat de ce code ?

A B C

Responses: 0

Cas complexes

```php
<?php
class A
{
  public static function appelerQuiEstCe(){
    static::quiEstCe();
  }
  public static function quiEstCe(){
    echo 'A';
  }
}

class B extends A
{
  public static function test(){
    // parent::quiEstCe();
    parent::appelerQuiEstCe();
  }
  public static function quiEstCe(){
    echo 'B';
  }
}

class C extends B
{
  public static function quiEstCe(){
    echo 'C';
```

```
C
```

- Appel de la méthode test de la classe C
- la méthode n'a pas été réécrite, on appelle donc la méthode `test` de la classe B
- on appelle maintenant la méthode `appelerQuiEstCe` de la classe A (avec `parent::`)
- résolution statique à la volée : on appelle la méthode `quiEstCe` de la classe qui a appelé la méthode `appelerQuiEstCe`
- la méthode `quiEstCe` de la classe C est donc appelée car c'est depuis la classe C qu'on a appelé la méthode `test`.

Autre cas

```php
<?php
class TestParent
{
  public function __construct()
  {
    static::qui();
  }
  public static function qui()
  {
    echo 'TestParent';
  }
}

class TestChild extends TestParent
{
  public function __construct()
  {
    static::qui();
  }
  public function test()
  {
    $o = new TestParent();
  }
  public static function qui()
  {
    echo 'TestChild';
```

Quel est le résultat de ce code ?

- **A** `TestParentTestParent`
- **B** `TestChildTestChild`
- **C** `TestChildTestParent`
- **D** `TestParentTestChild`

A B C D

Responses: 0

Autre cas

```php
<?php
class TestParent
{
  public function __construct()
  {
    static::qui();
  }
  public static function qui()
  {
    echo 'TestParent';
  }
}

class TestChild extends TestParent
{
  public function __construct()
  {
    static::qui();
  }
  public function test()
  {
    $o = new TestParent();
  }
  public static function qui()
  {
    echo 'TestChild';
```

```
TestChildTestParent
```

- Création d'une instance de la classe TestChild;
- appel de la méthode `qui` de la classe TestChild puisque c'est la méthode `__construct` de la classe TestChild qui a été appelée ;
- appel de la méthode test de la classe TestChild;
- création d'une instance de la classe TestParent;
- appel de la méthode `qui` de la classe TestParent puisque c'est la méthode `__construct` de cette classe qui a été appelée.