http://bdelespierre.fr/article/la-poo-en-php-en-10-minutes-ou-moins/

Si vous avez vécu sous un rocher dans une grotte sous une montagne au fin fond du Jura ces 40 dernière années et n'avez
jamais entendu parler de programmation orientée objet, je vais tenter de vous faire un rapide résumé.

La programmation orientée objet est l'évolution la plus significative dans le monde de l'informatique depuis l'invention du transistor.
Bien qu'elle ait été théorisée dans les années 60, son implémentation correcte devait attendre 2004 pour voir le jour en PHP.
Il s'agit d'une approche différente (mais complémentaire) de la programmation impérative, à la fois plus souple, plus puissante
mais surtout plus naturelle. Le concept de base est de créer une structure qui va permettre de rassembler à la fois des données et des traitements.

Comme vous le verrez tout au long de cet article, l'idée principale de la programmation orientée objet est de faciliter
la réutilisabilité et la généricité tout en favorisant la simplicité et la cohérence.

La programmation orientée objet repose sur 3 concepts clés: l'encapsulation, l'héritage et le polymorphisme (généralement regroupés
sous l'appelation de paradigme objet) que je me propose d'exposer. L'objet de cet article est de vous initier à cette forme
de programmation dans le cadre du langage PHP, il s'adresse donc à des débutants au fait de la programmation "classique"
(donc procédurale) en PHP.

Important: Cet article couvre le paradigme objet en PHP mais pas les petites spécificités propres à PHP comme la résolution statique
à la volée (ou late static binding) ni les traits. Ces concepts feront l'objet d'un futur article, dans l'immédiat je vous recommande
de maîtriser les bases de l'OOP en PHP avant d'aller plus loin.

L'encapsulation
Le principe fondamental de la programmation orienté objet est la notion de classe. Vous savez déjà manier des tableaux et
des variables (qui sont des structures de données) et des fonctions (qui sont des structures de comportements).
La classe vous permet de les rassembler dans une structure commune.

On appelle classe tout regroupement cohérent de données (ou propriétés) et de comportements (ou méthodes).
Les proprétés et méthodes d'une classe sont égalements appelés membres. En fait, on peut penser à une classe comme
à un "moule" destiné à la fabrication d'objets (ou instances de classes).

Pour créer un nouvel objet, on utilise le mot clé new suivi du nom de la classe à instancier. Une fois l'objet crée,
son état interne (les valeurs de ses propriétés) sera préservé jusqu'a sa destruction explicite avec la fonction
unset ou jusqu'a la fin du script.

Note: une propriété peut avoir une valeur par défaut mais cette valeur ne peut pas être une expression (comme 1+2 par exemple).
Si vous avez besoin d'initialiser une propriété à l'aide d'une expression, utilisez le constructeur comme montré ci-dessous.

On accède aux membres par l'intermédiaire de l'opérateur flèche (->). Au sein d'une classe, le mot clé [b]$this[/b] permet
d'accéder au membres d'instance en cours ($this représente en fait l'objet).

Exemple d'instanciation:

```php
<?php

class MaClasse
{
    // propriété
    var $a = 1;

    // méthode
    function afficher()
    {
        echo $this->a;
    }
}

$mon_instance = new MaClasse;
$mon_instance->afficher(); // affiche 1
$mon_instance->a = 2;
$mon_instance->afficher(); // affiche 2

?>
```

Visibilité d'un membre
Au sein d'une classe, un membre peut également avoir une visibilité:

public (ou var) le membre est visible à l'intérieur et à l'exterieur de la classe
protected le membre n'est visible qu'à l'intérieur de la classe et de ses classes filles (voir plus bas pour l'héritage)
private le membre n'est visible qu'a l'intérieur de la classe
La notion de visibilité d'un membre est intimement liée à la notion d'état. En restreignant la visibilité d'un membre,
l'objet est protégé contre les altérations de son état depuis l'exterieur qui pourraient nuire à son fonctionnement.
Par exemple, si vous créez une classe de manipulation de fichier et que cette classe porte une propriété qui est
la ressource du fichier, vous ne voulez pas qu'on puisse la fermer de l'exterieur avec fclose car cela perturberai l'éxécution de vos méthodes.

Définir la visibilité d'un membre n'est pas obligatoire, en l'absence de visibilité un membre sera public (les propriétés doivent
au moins être déclarées avec le mot clé var). Je vous recommande d'utiliser efficacement la visibilité afin de protéger
le fonctionnement et les données internes de la classe (qui n'ont pas besoin d'être exposés). Dans la pratique, il vaut mieux

définir les propriétés comme protégées par défaut afin qu'on ne puisse pas les modifier depuis l'extérieur de la classe et
de définir des accesseurs (des méthode d'accès à ces propriétés) au besoin.

Démonstration:

```php
<?php

class MaClasse
{
    public $proprietePublique;

    protected $_proprieteProtegee;

    private $_proprietePrivee;

    public function methodePublique()
    {
    }

    protected function _methodProtegee()
    {
    }

    private function _methodePrivee()
    {
    }
}

$objet = new MaClasse;

$objet->proprietePublique = 1;  // ok
$objet->_proprieteProtegee = 2; // FATAL_ERROR
$objet->_proprietePrivee = 3;   // FATAL_ERROR

$objet->methodePublique();      // ok
$objet->methodProtegee();       // FATAL_ERROR
$objet->methodePrivee();        // FATAL_ERROR

?>
```

Par convention, les membres protégés et privés sont préfixés d'un caractère '_' afin de les identifier plus facilement à la lecture du code.

Le constructeur
La création d'une instance provoque systématiquement l'appel au constructeur de la classe (la méthode d'instance __construct).
Le rôle du constructeur est de construire l'objet, le plus souvent il s'agit d'initialiser ses propriétés mais on peut également
faire appel à d'autres méthodes lors de l'instanciation. Cette méthode n'est pas obligatoire, en son absence, le constructeur
par défaut de PHP sera utilisé (comme dans l'exemple plus haut).

Exemple d'utilisation du constructeur:

```php
<?php

class MaClasse
{
    // propriété protégée
    protected $a;

    // constructeur
    function __construct($valeur)
    {
        $this->a = $valeur;
    }

    // méthode
    function afficher()
    {
        echo $this->a;
    }
}

$mon_instance = new MaClasse("hello");
$mon_instance->afficher(); // affiche hello

?>
```

Le destructeur
La destruction d'une instance provoque systématiquement l'appel du destructeur de la classe (la méthode d'instance __destruct).
Le rôle avoué du destructeur est de pouvoir détruire proprement l'objet, par exemple en fermant les ressources utilisées par
l'instance (comme un fichier ouvert avec fopen par exemple). Comme pour le constructeur, cette méthode n'est pas obligatoire.
Dans la réalité, les usages du destructeur sont assez rares.

Exemple d'utilisation du destructeur:

```php
<?php

class MaClasse
{
    protected $ressource;

    function __construct($fichier)
    {
        $this->ressource = fopen($fichier, 'r');
    }

    function lireUneLigne()
    {
        return fgets($this->ressource);
    }

    function __destruct()
```

```
18      {
19          // fermer proprement le fichier
20          fclose($this->ressource);
21      }
22  }
23
24  $mon_instance = new MaClasse('fichier.txt');
25  echo $mon_instance->lireUneLigne(); // lit la première ligne
26  echo $mon_instance->lireUneLigne(); // lit la seconde ligne
27  unset($mon_instance); // le destructeur est appellé, le fichier est fermé
28
29  ?>
```

Membres de classes et membres d'instances
Un membre peut être lié à l'instance de la classe (on parle alors de membre d'instance), c'est le cas dans tous les exemples
ci-dessus avec $this->nomDeLaPropriete. Un membre d'instance ne peut pas être utilisé sans l'objet (l'instance) qui va avec.

Mais un membre peut également être directement lié à la classe elle-même plutôt qu'a ses instances; on parle alors de membre
de classe ou de membre statique. Il est pour cela précédé du mot clé static et on y accède à l'aide de l'opérateur de résolution
de portée ::. Au sein d'une classe le mot clé self permet d'accéder aux membres statiques, self représente la classe en cours de
la même façon que $this représente l'instance en cours (le mot clé $this n'est pas disponible au sein d'un membre statique et
son utilisation provoquera une erreur fatale).

La visibilité s'applique de la même façon sur les membres d'instances et sur les membres de classes, leurs effets sont basiquement
les mêmes sauf dans le cas de l'héritage de méthodes statiques que nous détaillerons plus bas.

Exemple de classe disposant de membres statiques:

```
 1  <?php
 2
 3  class MaClasse
 4  {
 5      // propriété de classe
 6      public static $prefixe = "hello";
 7
 8      // propriété d'instance
 9      protected $mot;
10
11      // constructeur
12      public function __construct($mot)
13      {
14          $this->mot = $mot;
15      }
16
17      // méthode d'instance
18      public function afficher()
19      {
```

```php
        echo self::$prefixe . ' ' . $this->mot;
    }

    // méthode de classe
    public static function definirPrefixe($prefixe)
    {
        self::$prefixe = $prefixe;
    }
}

$obj1 = new MaClasse("world");
$obj1->afficher(); // affiche hello world

MaClasse::definirPrefixe("strange");

$obj2 = new MaClasse("world");
$obj2->afficher(); // affiche strange world
$obj1->afficher(); // affiche strange world également

?>
```

L'héritage

Avec l'héritage, on entre dans la vraie plus-value des langages objets par rapport aux langages impératifs, il s'agit d'un moyen simple de favoriser la réutilisabilité du code.

L'héritage va permettre une réutilisation verticale du code, c'est à dire que nos classes vont former une hiérarchie dans laquelle les enfants (les classes filles) pourront réutiliser les comportements et les données de leur(s) parent(s) (les classes mères).

Important: l'héritage entre deux classes caractérise la relation "est un espèce de..." Par exemple: une voiture est une espèce de véhicule, un chien est une espèce d'animal. L'héritage doit être sémantiquement valide afin d'éviter les dérives comme l'héritage fonctionnel. Comme le disait mon prof de Java: "ce n'est pas parce que le chien pisse sur le poteau que le chien est un espèce de poteau"... Donc ce n'est pas parce que le chien utilise le poteau qu'il doit hériter de poteau.

Concrêtement, lors qu'une classe B (la fille) hérite d'une classe A (la mère), elle dispose alors de tous les membres publics et protégés de sa mère (d'ou l'importance de cette différentiation entre membre protégé et privé). On réalise un héritage quand il y a lieu d'effectuer une spécialisation, c'est à dire que la classe fille va redéfinir (on parle aussi de surcharge) ou ajouter des comportements.

Pour que la classe B hérite de A, on utilise le mot clé extends comme montré ci-dessous. On peut, depuis une classe fille, faire explicitement appel au méthode de la classe parente à l'aide du mot clé parent, c'est pratique lorsqu'il s'agit de réutiliser une partie du comportement de la classe parent sans pour autant devoir le réécrire.

```php
<?php

class Vehicule
{
    protected $nombreDeRoues;

    public function __construct($nombreDeRoues)
    {
        $this->nombreDeRoues = $nombreDeRoues;
    }

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

class Voiture extends Vehicule
{
    public function __construct()
    {
        // appel du constructeur parent
        parent::__construct(4);
    }
}

class Moto extends Vehicule {

    public function __construct()
    {
        // appel du constructeur parent
        parent::__construct(2);
    }
}

$maVoiture = new Voiture;
echo "Ma voiture a " . $maVoiture->nombreDeRoues() . " roues"; // Ma voiture a 4 roues

$maMoto = new Moto;
echo "Ma moto a " . $maMoto->nombreDeRoues() . " roues"; // Ma moto a 2 roues

?>
```

Important: En PHP, l'héritage multiple n'existe pas. Ce qui signifie que vous ne pouvez par déclarer une classe qui hérite de deux autres classes. En somme, class A extends B, C c'est interdit. Vous pouvez en revanche réaliser plusieurs interfaces (voir le chapitre sur le polymorphisme ci-dessous).

Filliation, parents, enfants et arbre généalogique

Une classe n'est pas seulement un moyen commode de rassembler des données et des comportements au sein d'une structure, c'est surtout un excellent moyen de créer de nouveaux types de données, car en réalité une classe peut être considérée comme un type au même sens qu'une chaine ou un entier.

Quand on réalise un héritage, la classe fille caractérise un nouveau type mais est toujours du type de la mère. Par exemple une pomme est toujours considérée un fruit, qui est toujours considéré comme un végétal. En programmation orientée objet c'est le même concept, on parle alors de hiérarchie de types.

Comme tout bon langage objet, PHP dispose de l'opérateur instanceOf qui permet de savoir si une instance est d'un type donné. Cet opérateur renvoie true si l'objet est du type (ou super-type) spécifié.

Exemple de hiérarchie:

```php
<?php

class Vivant
{
}

class Vegetal extends Vivant
{
}

class Fruit extends Vegetal
{
}

class Apple extends Fruit
{
}

class GoldenLady extends Apple
{
}

$object = new GoldenLady;

var_dump( $object instanceOf Apple );  // true
var_dump( $object instanceOf Fruit );  // true
var_dump( $object instanceOf Vegetal); // true
// ... etc.

?>
```

Il est d'ailleurs possible de spécifier sur un prototype quel type d'objet est attendu pour un paramètre donné:

```php
<?php

function manger(Fruit $fruit)
{
    echo "I'm eating " . get_class($fruit);
}

$object = new Apple;
manger($object); // I'm eating Apple
```

```
10
11 ?>
```

On appelle cette syntaxe de paramètres le type-hinting; c'est à dire qu'on demande explicitement des objets du type (ou super-type) spécifié. N'importe quoi d'autre provoquera une erreur. Si on avait passé un type natif comme un entier ou encore une instance qui hérite de la classe Vegetal mais qui n'est pas un fruit, on se serait fait jeter.

Le type-hinting est extrèmement pratique pour sécuriser vos fonctions et vos méthodes en empêchant le développeur d'y mettre n'importe quoi.

Note: contrairement à d'autres langages objets comme Java, les objets n'héritent pas par défaut de la class Object, vous devez donc recourrir à is_object si vous voulez valider le type objet.

Le polymorphisme

Le polymorphisme (du grec plusieurs formes) est le mécanisme grâce auquel une même méthode peut être implémentée par plusieurs types (donc plusieurs classes) favorisant ainsi la généricité. Concrêtement, si une méthode est déclarée et/ou définie dans la classe parente d'une classe donnée, vous n'avez pas à vous préocuper du type précis de la fille avec laquelle vous travaillez pour l'utiliser.

Par exemple, une méthode obtenirAire est commune aux classes Cercle, Carre et Rectangle qui héritent toutes de Forme. Quand vous travaillez avec une instance de Forme, vous n'avez donc plus à vous soucier de savoir si c'est un cercle, un carré ou un rectangle pour obtenir son aire.

Comme dans la plupart des langages objet, le polymorphisme en PHP est un polymorphisme par sous-typage (ou dérivation) c'est à dire qu'on va se servir de la redéfinition de comportements prévue par l'héritage pour faire du polymorphisme. Nous avons vu plus haut avec Vehicule, Voiture et Moto qu'il est possible de redéfinir une méthode de la classe mère dans sa fille (en l'occurence, il s'agissait de son constructeur). Nous allons maintenant voir qu'il est possible d'aller plus loin.

Abstraction

En programmtion objet, il est possible de déclarer des méthodes dont la définition n'est pas encore connue dans la classe mère mais le seront dans une classe fille. On parle alors de méthodes abstraites. Une classe qui contient au moins une méthode abstraite doit elle aussi être déclarée abstraite et ne peut plus être instanciée - vu que tout son code n'est pas implémenté. On parle alors de classe abstraite.

Une méthode abstraite est précédée du mot clé abstract (idem pour la classe qui la porte) et ne peut pas avoir de corp.

Exemple de classe abstraite:

```php
<?php

abstract class Animal
{
    protected $nom;

    public function __construct($nom)
    {
        $this->nom = $nom;
    }
```

```php

    abstract public function parler();
}

class Chien extends Animal
{
    public function parler()
    {
        echo "$this->nom: Wouf Wouf\n";
    }
}

class Chat extends Animal
{
    public function parler()
    {
        echo "$this->nom: Miaou\n";
    }
}

$chien = new Chien("Rex");
$chien->parler(); // Rex: Wouf Wouf

$chat = new Chat("Sac-a-puces");
$chat->parler();  // Sac-a-puces: Miaou

?>
```

Par opposition, une classe qui définit l'intégralité de ses comportements est souvent appellé classe concrête.

Interfaces

Si toutes les méthodes d'une classe sont abstraites, cette classe est alors une interface. Dès lors, on utilise le mot clé interface en lieu et place du mot clé class. L'intérêt principal des interfaces est que plusieurs d'entre elles peuvent être réalisées (ou implémentées) par une même classe. En somme class A extend B implements C,D,E est tout à fait valide.

On croise d'ailleurs fréquement des classes qui réalisent plusieurs interfaces, c'est le cas notamment de la classe PHP ArrayIterator qui réalise Iterator, Traversable, ArrayAccess, SeekableIterator, Countable et Serializable.

En fait l'interface donne une "forme" a notre classe: elle va définir comment celle-ci doit se présenter (par là on entends quelles devront être ses méthodes) mais pas comment elle doit se comporter. Les iterateurs de la SPL en sont un bel exemple: tous les itérateurs partagent des prototypes de méthodes communs mais chacun les implémente différement.

Note: vu que toutes les méthodes d'une interface sont obligatoirement abstraites, il est inutile de le spécifier avec le mot clé abstract sur les méthodes et la classe.

Exemple d'interface:

```php
<?php

interface Forme2D
```

```php
{
    // toutes les formes en deux dimention ont une aire...
    public function obtenirAire();

    // ... et un périmètre
    public function obtenirPerimetre();
}

class Carre implements Forme2D
{
    protected $cote;

    public function __construct($cote)
    {
        $this->cote = $cote;
    }

    public function obtenirAire()
    {
        return pow($this->cote, 2);
    }

    public function obtenirPerimetre()
    {
        return 4 * $this->cote;
    }
}

class Rectangle implements Forme2D
{
    protected $longueur;
    protected $largeur;

    public function __construct($longueur, $largeur)
    {
        $this->longueur = $longueur;
        $this->largeur = $largeur;
    }

    public function obtenirAire()
    {
        return $this->longueur * $this->largeur;
    }

    public function obtenirPerimetre()
    {
        return 2 * ($this->longueur + $this->largeur);
    }
}

class Cercle implements Forme2D
{
    protected $rayon;
```

```
57
58      public function __construct($rayon)
59      {
60          $this->rayon = $rayon;
61      }
62
63      public function obtenirAire()
64      {
65          return M_PI * pow($this->rayon, 2);
66      }
67
68      public function obtenirPerimetre()
69      {
70          return M_PI * ($this->rayon * 2);
71      }
72  }
73
74  ?>
```

Dans l'exemple ci-dessous, connaissant les méthodes déclarées par Forme2D, nous pouvons décrire des classes qui travaillent avec n'importe quelle instance de Forme2D. C'est ça la généricité !

Exemple d'utilisation générique d'une forme:

```
 1  <?php
 2
 3  // en reprennant l'exemple précédent
 4
 5  class Figure2D
 6  {
 7      protected $formes;
 8
 9      public function ajouter(Forme2D $forme)
10      {
11          $this->formes[] = $forme;
12      }
13
14      public function surfaceTotale()
15      {
16          $surface = 0;
17          foreach ($this->formes as $forme)
18              $surface += $forme->obtenirAire();
19
20          return $surface;
21      }
22  }
23
24  $figure = new Figure2D;
25  $figure->ajouter(new Cercle(3));
26  $figure->ajouter(new Carre(4));
27  $figure->ajouter(new Rectangle(5,6));
28
29  echo "Ces trois figures ont une surface totale de " . $figure->surfaceTotale(); // 74.27...
30
```

```
31 ?>
```

On voit rapidement l'intérêt du polymorphisme dans ce cas: grâce à l'interface Forme2D, on peut créer autant de type de formes qu'on veut, par exemple le triangle, le losange, le polygone etc. Et tous ces objets, aussi longtemps qu'ils hériteront de Forme2D, seront utilisables avec la classe Figure2D.

Contrairement aux classes, une interface peut étendre plusieurs interfaces avec le mot clé extends. On notera également que les cas qui justifient un héritage d'interfaces en vue d'un sous typage sont assez rares, il est plus fréquent de voir des interfaces rassemblant les déclarations de plusieurs autres dans un but pratique:

Exemple d'héritage d'interfaces:

```
1 <?php
2
3 interface MaCollection extends Serializable, Countable, ArrayAccess, Iterator
4 {
5     // ...
6 }
7
8 ?>
```

En PHP il existe plusieurs interfaces pré-définies qui sont assez pratiques et dont on se sert souvent notamment pour la surcharge des opérateurs comme:

Serializable qui permet de spécialiser le comportement de la fonction fonction serialize() pour un objet
Countable qui permet d'utiliser directement la fonction count() sur l'objet
ArrayAccess qui permet d'utiliser l'opérateur
Iterator spécifie le comportement de l'objet comme un iterateur, utilisable avec foreach
Ces interfaces et leurs usages feront l'objet d'un futur article.

Vous trouverez la liste de ces interfaces pré-déclarées sur la page du manuel.

Conclusion
Vous l'aurez compris, la programmation orientée objet c'est aussi complexe que puissant. Une fois les base de l'OOP maîtrisées, vous pourrez commencer à aller plus loin dans l'architecture logicielle, la programmation par composants, les design-patterns etc.

De nos jours, la plupart des infrastructure logicielles importantes ou complexes sont codées en Objet, ce n'est pas un hasard. La simplicité grâce à l'encapsulation, la réutilisabilité grâce à l'héritage et la généricité grâce au polymorphisme expliquent à elles trois l'engoument persistant pour les langages objets et leur large domination parmi les langages de programmation.

Vous verrez, une fois habitué, penser objet vous sera naturel.

/************* Les traits en PHP ********************/**

Comme nous l'avons vu dans l'article La POO en PHP en 10 minutes (ou moins), l'héritage est un moyen simple de réutiliser des comportements existants et donc d'éviter la dupplication de code. C'est d'ailleurs un pilier fondamental de la programmation orientée objet.

En revanche, la relation d'héritage est en fait une spécialisation. Ce qui signifie que la classe qui hérite doit partager une nature commune avec sa mère; par exemple, la classe Voiture

peut être une spécialisation de la classe Vehicule mais certainement pas de la classe Canapé. Pourtant, dans le canapé comme dans la voiture on peut s'asseoir à plusieurs. Comment faire dans ce cas pour implémenter la méthode faireAsseoir(Personne $personne) dans deux classes qui ne partagent pas du tout les mêmes classes de base et ne peuvent pas s'hériter entre-elles ?

Jusqu'ici, la solution à ce genre de problème était de soit:

duppliquer le code (et tant pis pour les bonnes pratiques)
réaliser une composition (par exemple en injectant une instance de Fauteuil, ce qui introduit du couplage et de la complexité)
L'un comme l'autre peut poser problème. C'est pourquoi il existe depuis PHP 5.4 la notion de traits.

Encore un type de classe ?
Oui et non. Un trait est une structure qui rassemble les avantages d'une classe et d'une interface:

comme une classe, un trait peut avoir des méthode définies (et pas seulement déclarées)
comme une classe, un trait peut avoir des propriétés (contrairement aux interfaces)
comme une interface, on peut utiliser plusieurs traits dans une même classe
comme une interface, un trait ne peut être instancé
Un trait caractérise en quelque sorte un hértiage horizontal, indépendant de la hiérachie des types. Pour utiliser un trait, il n'est pas nécéssaire que celui-ci partage une nature commune avec la classe qui l'utilise. On peut le voir en fait comme un "morceau" de classe fournissant des fonctionnalités indépendantes de tout typage.

Pour ceux qui travaillent avec JS ou Ruby, on retrouve avec les traits la notion de mixin, c'est à dire de la fabrication de classes en "mélangeant" des objets. Par ailleurs, les traits existent aussi en Scala et bientôt en SmallTalk.

A quoi ça ressemble ?
Prennons un exemple utile: l'EventEmitter. Il s'agit de permettre à un objet d'émettre des évènements, par exemple lors d'un changement de son état, qui seront capturés par les écouteurs. Avant les traits on devait disposer une classe dédiée et passer ses instances aux classes susceptibles d'émettre des évènements, maintenant on peut tout simplement créer un jeu de méthodes qui feront exactement la même chose:

```php
<?php

trait EventEmitter
{
 protected $listeners;

 public function on($event, Closure $listner)
 {
     $this->listeners[$event][] = $listener->bindTo($this);
 }

 public function emit($event, array $data = [])
 {
     if (!isset($this->listeners[$event]))
         return;

```

```
17      foreach ($this->listeners[$event] as $listener)
18          $listener($data);
19  }
20 }
21
22 ?>
```

Pour utiliser un trait au sein d'une classe, on utilise le mot clé use:

```
 1 <?php
 2
 3 class Personne
 4 {
 5  use EventEmitter;
 6
 7  public function __construct($name)
 8  {
 9      $this->name = $name;
10  }
11
12  public function naitre($date)
13  {
14      $this->emit('naissance', [$date]);
15  }
16
17  public function mourrir($date)
18  {
19      $this->emit('mort', [$date]);
20  }
21 }
22
23 $p = new Personne("Denis Ritchie");
24 $p->on("naissance", function($data) {
25  echo "{$this->name} est néé le {$data[0]}";
26 });
27 $p->on("mort", function($data) {
28  echo "{$this->name} est mort le {$data[0]}";
29 });
30
31 $p->naitre('9 Septembre 1941');
32 $p->mourrir('12 Octobre 2011');
33
34 ?>
```

Et qu'est-ce qu'on peut mettre dans un trait ?
Presque tout ce qu'on peut mettre dans une classe en réalité: des membres statiques, protégés, ou abstrait. On peut même utiliser des traits dans les traits

```
 1 <?php
 2
 3 trait Foo
 4 {
 5  // un autre trait
 6  use Bar;
 7
```

```php
 // une propriété protégée
 protected $a;

 // une propriété statique
 public static $b = 2;

 // une méthode abstraite
 abstract public function c();

 // une méthode statique, protégée et abstraite
 abstract protected static function e();

 // etc.
}

?>
```

La classe qui utilise le trait peut à son tour changer le nom et la visibilité des méthodes du trait. Pour cela, on utilise la syntaxe suivante:

```php
<?php

class Foo
{
 // un seul trait utilisé
 use Bar {
     uneMethodePublique as protected unAutreNomDeMethode;
 }
}

class Baz
{
 // plusieurs traits utilisés
 use A, B {
     A::hello as traitHello;
     B::world as traitWorld;
 }
}

?>
```

En revanche, on ne peut pas changer le nom des propriétés. Avoir la même propriété dans la classe et dans le trait résulte soit d'une erreur stricte (E_STRICT) si les propriétés sont compatibles, soit d'une erreur fatale (E_FATAL_ERROR).

```php
<?php

trait Foo
{
 public $a = true;
 public $b = false;
}

class Bar
{
```

```
11  use Foo;
12  public $a = true; // E_STRICT
13  public $b = true; // E_FATAL_ERROR !
14 }
15
16 ?>
```

Priorisation

Les traits surchargent les méthodes de la classe parente mais les méthodes du trait sont surchargées par les méthodes déjà définies dans la classe. En fait c'est comme si le trait était "entre" la classe parente et la classe au niveau de la hiérarchie d'héritage.

```
 1 <?php
 2
 3 class A
 4 {
 5  public function foo() { echo 1; }
 6 }
 7
 8 trait B
 9 {
10  public function foo() { echo 2; }
11 }
12
13 class C extends A
14 {
15  use B;
16 }
17
18 $c = new C;
19 $c->foo(); // affiche 2 car A::foo est écrasée par B::foo
20
21 class D extends A
22 {
23  use B;
24
25  public function foo() { echo 3; }
26 }
27
28 $d = new D;
29 $d->foo(); // affiche 3 car D::foo écrase B::foo (qui écrase A::foo)
30
31 ?>
```

Collisions

Quand une classe utilise plusieurs traits, des colisions de noms de méthodes peuvent se produire. Pour résoudre le problème, PHP 5.4 introduit un nouvel opérateur: insteadof pour signifier "utilise celle-ci plutôt que celle-là". Changer manuellement le nom d'une méthode reste cependant possible. Il est important de noter que si le conflit n'est pas explicitement résolu, une erreur fatale (E_FATAL_ERROR) est émise.

```
 1 <?php
 2
 3 trait Foo
 4 {
```

```php
 public function hello() { echo "Hello"; }
}

trait Bar
{
 public function hello() { echo "World"; }
}

class FooBar
{
 use Foo, Bar {
     Foo::hello insteadof Bar; // utiliser la méthode de Foo plutôt que celle de Bar
     Bar::hello as world;      // changer le nom de la méthode de Bar
 }

 // la classe dispose des méthode hello() et world()
}

?>
```

Conclusion
En résumé les traits sont utiles pour:

rassembler des comportements sans devoir créer des classes fourre-tout
partager des comportements entre des classes qui n'ont aucun lien entre-elles
fournir des bases d'implémentation concrètes pour des interfaces
limiter le trop grand sous typage des classes
limiter le nombre d'instances à créer par rapport à des compositions
Selon moi, le trait n'est pas une alternative à l'injection de dépendance mais un complément. L'injection de dépendance, parce qu'elle suppose que les fonctionnalités soient isolées dans des composants séparés, encourage la création de petits composants sans réelle valeur ajoutée à encapsuler. Les traits permettent de réduire le nombre de ces composants (et donc le nombre de classes) sans pour autant sacrifier à l'encapsulation ni à la séparation des responsabilités. L'exemple de l'EventEmitter décrit plus haut en est une parfaite illustration.

Coté performances, je n'ai pas encore pensé à regarder l'impact de l'utilisation d'un trait, je ne pense pas qu'il soit très important mais si quelqu'un dispose des chiffres n'hésitez pas à les partager dans les commentaires.

/************* Injections de dépendances et composants en PHP ******************/**
Les développeurs sont par nature assez paresseux, s'ils peuvent réutiliser des fonctionnalités existantes pour se simplifier la vie ils le font en règle générale. C'est encore mieux si ces fonctionnalités leurs sont fournies directement par des librairies ou par l'API du langage. Ne pas réinventer la roue est un concept qui revient souvent en programmation, en effet, si la solution à un problème posé existe déjà, il est inutile voire absurde de dépenser du temps et de l'énergie à fournir une autre solution.

Il n'est donc pas étonant que, très tôt dans l'informatique, les langages offrants ces fonctionnalités génériques soient rapidement devenus populaires. De nos jours, la vaste majorité des langages offrent nativement des fonctionnalités élémentaires pour afficher du texte, manipuler des données ou la mémoire ainsi que des fonctionnalités plus évolués généralement liées à la nature du langage en question.

Dans un précédent article, je vous ai parlé de la révolution introduite par l'approche orientée-object dans les langages de programmation. Appliquée à l'architecture logicielle, cette approche à donné naissance à la programmation orientée composant, une forme d'architecture au sein de laquelle les librairies sont encapsulées dans des briques logicielles indépendantes dont le développeur se sert pour construire le système d'information. L'objectif étant de favoriser la réutilisabilité et la flexibilité par une approche modulaire de l'architecture. En effet, grâce à l'utilisation de composants il devient aisé de créer des briques génériques pour des besoins réccurents et de changer si nécéssaire un composant pour adresser au mieux un problème donné.

Nous allons donc nous intéresser à la notion de composant dans le cadre du langage PHP. La notoriété croissante des frameworks PHP tels que Symfony ou Zend Framework aidant, la programmation orientée composant est rapidement devenue populaire auprès de la communauté. Il m'apparaît donc essentiel de faire le point sur les problématiques liées à cette forme d'architecture et comment les adresser à l'aide d'un pattern aujourd'hui célèbre: l'injection de dépendances.

L'interopérabilité
Le concept de composant ne date pas d'hier en PHP car depuis 1999 PEAR favorise la réutilisation de composants génériques. Mais il faudra attendre la fin des années 2000 pour que le phénomène prenne de l'ampleur notamment grâce à Composer et son dépôt Packagist. De nos jours, la plupart des nouvelles librairies PHP sont fournies sous la forme de composants afin de faciliter leur installation et leur mise à jour, favorisant de ce fait leur utilisation par la communauté des développeurs.

Les développeurs ont dès lors commencé à concevoir leurs application non plus comme des briques logicielles monolithiques mais comme un assemblage de composants liés les uns aux autres. S'est alors posé très vite la question de l'interraction entre ces composants.

En effet, si une application à besoin d'un routeur capable d'invoquer des contrôleurs et que ces derniers font appels à des vues après avoir effectué les opération modèle nécéssaires à l'aide d'un mappeur objet-relationnel (ORM), comment lier tous ces composants entre eux ?

La façon la plus simple est encore de coder ces liens "en dur", donc dans un composant A faire explicitement appel à un composant B. Mais cette approche nuit considérablement à la réutilisabilité et à la flexibilité car A devient indissociable de B. De fait, plus le nombre de composants augmente, plus le couplage augmente, et on finit par détruire toute modularité dans l'application et l'approche orientée composants devient inutile.

Pour résoudre ce problème, il est préférable que A utilise B sans avoir connaissance de l'implémentation conrête de B, ce qui nous permet de fournir à A la forme de B qui nous convient. On retrouve ici plus ou moins l'idée de polymorphisme héritée de la programmation orientée objet, nos composants ne vont pas travailler avec les implémentations concrêtes d'autres composants mais avec leurs représentations abstraites.

Il existe de nombreux patrons de conception qui résolvent ce problème: Adaptateur, Stratégie, Médiateur... Mais nous allons nous intéresser plus particulièrement à l'injecteur de dépendances (Dependency Injector en anglais).

L'injection de dépendances
L'injection de dépendances est un patron de conception qui permet de supprimer ces dépendances codées "en dur", rendant possible le remplacement des éléments de l'application lors du

runtime. Pour reprendre notre exemple, le composant A ne fera pas appel explicitement au composant B, c'est l'injecteur qui va le lui fournir (la plupart du temps sous la forme d'un object prêt à l'emploi). Le principe clé de ce pattern est la séparation du comportement vis-a-vis de la résolution de la dépendance.

L'injection de dépendance implique donc au moins 3 entitées:

un consommateur de dépendances
une définition des dépendances
un injecteur capable de créer les instances de dépendances
Le consommateur est reponsable de la description des types avec lesquels il travaille, mais il n'a pas à savoir quel est le type précis qu'il va utiliser, c'est justement le rôle de l'injecteur de le déterminer en utilisant la définition des dépendances. En fait, vous pouvez penser la définition de dépendances comme un moyen commode de "câbler" votre application.

Par exemple, si votre contrôleur doit travailler avec un objet représentant une vue, il demandera explicitement une instance de ViewInterface pour s'initialiser et l'injecteur pourra lui passer une instance de HTMLView ou de JSONView en fonction du contexte d'éxécution.

Un autre avantage de l'injection de dépendances est qu'il devient aisé de tester nos composants, on va pour cela leur fournir de fausses implémentations. Ces bouchons (mocks en anglais) sont des composants dont le comportement est totalement déterministe, ce qui nous permet de tester les composants dans lesquels ils sont injectés. Ces mocks également nous permettre de tester des cas d'erreur difficiles à reproduire.

On peut également créer des mocks pour émuler des traitements dans l'idée de mettre en place une démonstration du produit (les équipes marketing sont friandes de ce genre de flexibilité).

Par exemple, si votre contrôleur doit travailler avec un objet modèle dont le comportement dépend des données que vous avez en base, il devient difficile de tester le contrôleur car la base de données n'est par définition pas un élément déterministe et les données reçues ne sont pas forcément identiques d'un appel sur l'autre. Dans ce cas, nous allons passer à notre contrôleur un mock qui va émuler le bon ou le mauvais comportement des requêtes en base et on va regarder comment se comporte le contrôleur.

Il existe d'ailleurs des méthodologies de développement comme le développement piloté par les tests (test driven development ou TDD en anglais) qui placent l'écriture des tests avant l'écriture du code. Dans ce contexte de travail, il est impératif de prévoir toute une série de mocks afin de pouvoir dérouler les tests dans de bonnes conditions puis de les remplacer par leur réelle implémentation au fur et à mesure que le projet avance.

Exemple pratique
L'injection de dépendances est un patron qui se marie à merveille avec un patron qu'on ne présente plus désormais: le Modèle-Vue-Contrôleur ou MVC pour les intimes. Nous allons pour cet exemple utiliser l'injecteur de dépendances Pimple et nous allons définir nos dépendances au sein d'un composant central: Application.

Je ne vais pas mettre l'intégralité des classes impliquées car l'implémentation du MVC à l'aide de composants n'est pas au chapitre de cet article. Si nécéssaire, j'écrirais un article dédié là dessus.

```
1 <?php
2
```

```php
class Application extends Pimple {

    public function __construct (array $values) {

        $app = $this;

        $this['controllers'] = array(
            'default' => 'DefaultController',
        );

        $this['views_path'] = '/views';
        $this['views'] = array(
            'html' => $this->share(function () use ($app) {
                return new HTMLView($app['views_path']);
            }),
        );

        $this['models_path'] = '/models';
        $this['models'] = array();

        $this['router'] = $this->share(function () use ($app) {
            return new Router($app['controllers']);
        });

        $this['request_class'] = 'HTTPRequest';

        foreach ($values as $key => $value)
            $this[$key] = $value;
    }

    public function addController ($name, $class) {
        $this['controllers'][$name] = $class;
        return $this;
    }

    public function addViewHandler ($name, $handler) {
        $this['views'][$name] = $handler;
        return $this;
    }

    public function addModel ($name, $model) {
        $this['models'][$name] = $model;
        return $this;
    }

    public function run (Request $request = null) {
        if ($request === null)
            $request = new $this['request_class']($this);

        $this['router']->run($request);
    }
}
```

Cette classe est à la fois une façade qui nous simplifie grandement l'usage du MVC et un

injecteur de dépendances. Son constructeur représente l'étape d'initialisation de l'application, c'est là que toute la définition des composants à utiliser prends place. Cette définition peut être surchargé directement en passant un tableau au constructeur ou bien par la suite en utilisant les différentes méthodes.

On notera au passage l'usage de la méthode Pimple::share qui permet de définir des services partagés qui sont en fait des singletons.

Voici un exemple d'utilisation:

```php
<?php

// bla bla autoloader bla bla

$app = new Application;

// nos modèles sont des modèles MySQL
$app['models_path'] = 'models/MySQL';

// décommenter pour utiliser les mocks
// $app['models_path'] = 'models/mock';

// le modèle Articles est un itérateur
// qui s'initialise avec une instance
// vide de la classe article à utiliser
// pour l'itération.
$app->addModel('articles', function () use ($app) {
    include_once $app['models_path'] . '/articles.php';
    return new Articles($app['article']);
});

// le modèle Article est le CRUD
// permettant d'obtenir les données
// d'un article.
$app->addModel('article', function ($id = null) use ($app) {
    include_once $app['models_path'] . '/article.php';
    return new Article($id);
});

// enregistrer le contrôleur
// d'articles
$app->addController("article", "ArticleController");

// lancer l'application
$app->run();
```

Les modèles nous sont fournis par des fonctions annonymes qui sont en fait leurs fabriques. Cela permet à la fois de déterminer quelle famille de modèles utiliser (on inclut les classes en fonction du paramètre models_path) et de n'initialiser les modèles que lors que c'est utile: si les contrôleurs ne les utilisent pas, il n'y a pas d'instanciation.

Voyons enfin le contrôleur ArticleController:

```php
<?php
```

```php
class ArticleController extends Controller {

    public function index ($app, $q) {
        // récupérer la liste des articles
        // optionnellement filtrée par $q
        $articles = $app['models']['articles']->get($q);

        //
        return $app['views']['html']->load('article/articles.php', $articles);
    }

    public function article ($app, $id) {
        // retourner sur l'index de la section
        // article si l'id n'est pas fourni
        if (!$id)
            return $this->redirect($this, 'index');

        $article = $app['models']['article']($id);

        return $app['views']['html']->load('article/article.php', $article);
    }

    public function commenter ($app, $id) {
        $author = filter_input(INPUT_POST, 'comment_author', FILTER_SANITIZE_STRING);
        $body   = filter_input(INPUT_POST, 'comment_body',   FILTER_SANITIZE_STRING);

        if (!$id || !$author || !$body)
            return $this->redirect($this, 'index');

        $article = $app['models']['article']($id);
        $article->addComment($author, $body);

        return $app['views']['html']->load('article/article.php', $article);
    }
}
?
```

Dans cet exemple, on suppose qu'il est de la responsabilité de la classe Router de déterminer pour chaque action de contrôleur quels sont les paramètres à passer ($app représente l'application et les autres paramètres sont des paramètres GET). Pour la petite histoire, il suffit d'utiliser les classes de réflexion pour obtenir les noms des paramètres d'un méthode ou fonction et les faire correspondre à des valeurs dans $_GET par exemple.

On remarque immédiatement qu'il existe un couplage entre nos composants (en l'occurence les contrôleurs) et l'injecteur de dépendances. Il existe bien des moyens de supprimer ce couplage mais ce n'est pas toujours souhaitable.

Il est parfois beaucoup plus simple de fournir directement l'injecteur de dépendances aux composants pour qu'ils "tirent" eux-même leurs dépendances plutôt que d'imaginer un mécanisme complexe d'injection automatique. En PHP notamment, cette approche nous satisfait en général car nous obtenons suffisament de flexibilité par ce biais : dans notre exemple il sufft de décommenter une ligne pour utiliser directement les mocks au lieu de l'implémentation réelle ou encore d'enregistrer de nouveaux gestionnaires de vues pour les rendre directement utilisables au niveau des contrôleurs.

De plus, les contrôleurs faisant partie intégrante de l'application, cette dépendance entre controlleurs et application est tout à fait justifiée.

Conclusion
Comme le MVC, l'injection de dépendances peut prendre bien des formes. On aurait pu par exemple parler de la programmation par contrats qui nous aurait permi de définir pour chaque composant ses dépendances ou encore détailler comment effectuer une injection automatique à l'aides des classes de réflexion mais ce n'est pas notre objectif car il existe autant d'implémentation et d'usages de ce patron que les développeurs sont en mesure d'en imaginer.

J'espère que cet article vous aura permi de mieux comprendre les enjeux du découplage dans une architecture orientée-objet et vous a donné quelques pistes pour améliorer vos sources.

Je terminerai en vous mettant en garde (comme à mon habitude) contre les dérives du découplage et de la généricité en général: à un moment donné il faut arrêter de découpler et de banaliser au risque de voir la solution devenir plus complexe que le problème ce qui n'est jamais une bonne chose. L'injection de dépendances, comme la plupart des patterns existants, existe pour vous simplifier le travail, pas pour le rendre obscur et incompréhensible donc n'allez pas trop loin (croyez-moi, on a tôt fait de se retrouver avec des composants tellements "souples" qu'on ne sait même plus ce qu'ils font concrêtement).

Références
Programmation orientée composant
Dependency Injection
Pimple

/********************************* La résolution statique à la volée
***/

Si vous vous souvenez de mon article La poo en PHP en 10 minutes (ou moins), vous vous rappellerez sans doute que je n'ai pas souhaité parler des spécificités de PHP en matière de programmation orientée objet. C'est justement le sujet qui nous intéresse aujourd'hui; nous allons nous pencher sur une fonctionnalité relativement complexe mais puissante qu'est la résolution statique à la volée ou en anglais late static binding (LSB). Cette fonctionnalité fait partie des évolutions de la version 5.3 de PHP, les version antérieures étant à ce jour dépreciées depuis longtemps vous devriez pouvoir l'utiliser sans risque sur votre environnement.

Résolu-quoi ?
J'avoue que le nom est un peu barbare mais c'est tout à fait cohérent (troll ce qui est rare dans le monde PHP endtroll). Il désigne en fait la capacité du moteur à déterminer dans quel contexte exécuter une méthode statique, en d'autres termes effectuer la résolution du contexte d'éxécution lors du déroulement du script.

Vous vous souvenez que vous pouvez utiliser le mot-clé $this pour faire référence à l'instance en cours d'utilisation et à self pour faire référence à la classe. Le late static binding vous permet d'utiliser le mot-clé static en lieu et place du mot clé self.

D'accord mais à quoi ça sert ?
J'y viens. Pour bien comprendre la nuance, il faut penser au cas de l'héritage de méthode statiques. Considérons deux classes A et B liées par une relation d'héritage:

```php
<?php
class A
{
 public static $message = "Bonjour!";

 public static function hello()
 {
     echo self::$message;
 }
}

class B extends A
{
 public static $message = "Au revoir!";
}
?>
```

B hérite naturellement la méthode hello de A mais vous avez peut être déjà remarque que quand on exécute B::hello(), c'est "Bonjour!" et non "Au revoir!" qui s'affiche:

```php
<?php
A::hello(); // affiche "Bonjour!"
B::hello(); // affiche également "Bonjour!"
?>
```

Pourquoi ? Tout simplement parce que self "pointe" toujours sur la classe A, peu importe que la méthode hello soit executée sur B ou A. Donc au sein de hello, la propriété statique $message lue est toujours celle de A et jamais celle de B. C'est justement dans ce genre de cas de figure que la résolution statique à la volée est utile.

En utilisant **static** au lieu de self on demande au moteur d'utiliser en tant que contexte d'éxécution de la méthode hello la classe avec laquelle on l'appelle et non la classe où hello est définie.

```php
<?php
class A
{
 public static $message = "Bonjour!";

 public static function hello()
 {
     echo static::$message;
 }
}

class B extends A
{
 public static $message = "Au revoir!";
}

A::hello(); // affiche "Bonjour!"
B::hello(); // affiche "Au revoir!"
?>
```

En somme, utiliser static au lieu de self revient à utiliser $this mais pour un contexte statique.

Et c'est utile ?
En fait oui car ça permet de reproduire l'héritage des membres d'instance au niveau des membres statiques. Il devient dès lors possible de créer des abstractions statiques :

```php
<?php
abstract class Log
{
 final public static function message($message)
 {
     static::_write($message);
 }

 abstract protected static function _write($message);
}

class FileLog extends Log
{
 protected static $_file;

 protected static function _write($message)
 {
     static::$_file->write($message);
 }
}

class DatabaseLog extends Log
{
 protected static $_database;

 protected static function _write($message)
 {
     static::$_database->query("INSERT INTO 'log' ('message') VALUES ('$message');");
 }
}
?>
```

Cela permet:

de rester flexible et ouvert au niveaux des services statiques car leur spécialisation devient possible
de conserver l'unicicité du fournisseur de service sans avoir recours à un Singleton
de bénéficier des performances des contextes statiques dont l'exécution est 4x plus rapide que les contextes d'instance (c'est à dire avec des objets)
Astuce: vous pouvez utiliser le mot-clé **static** pour une fabrique:

```php
<?php
class A
{
 public function __construct($a, $b, $c)
 {
     // ...
 }

```

```php
 public static function newInstance($a, $b, $c)
 {
     return new static($a, $b, $c);
 }
}

class B extends A
{
 public function sayHello()
 {
     // ...
 }
}

// ça permet notamment de faire du chainage
B::newInstance()->sayHello();
?>
```

Ça craint rien ?
Le vrai problème avec les classes statiques, c'est qu'on retrouve un peu partout ce genre d'appel dans le code Classe::methode(...) ce qui introduit forcément du couplage.

En reprennant les classes FileLog et DatabaseLog ci-dessus, comment faire pour permettre à une classe d'utiliser indifféremment l'une ou l'autre et ainsi éviter un couplage serré ? Une solution simple consiste à utiliser le dynamisme du langage et stocker le nom de la classe de service à utiliser, c'est une méthode proche de ce qui se pratique en Java:

```php
<?php
class User
{
 public function __construct($logServiceClass)
 {
     if (!is_subclass_of($logServiceClass, 'Log'))
         throw new InvalidArgumentException("invalid log service class");

     $this->_logServiceClass = $logServiceClass;
 }

 public function save()
 {
     $log = $this->_logServiceClass;
     $log::message("saving user $this->id");

     // ...
 }
}
?>
```

Ou tout simplement passer l'instance de la classe statique à utiliser (car toute classe, même entièrement statique, reste instanciable - sauf évidement les classes abstraites). L'avantage de cette méthode est de pouvoir utiliser directement l'opérateur **instanceOf** et le type-hinting:

```php
<?php
class User
{
```

```
 4  public function __construct(Log $logService)
 5  {
 6      $this->_log = $logService;
 7  }
 8
 9  public function save()
10  {
11      $this->_log->message("saving user $this->id");
12
13      // ...
14  }
15 }
16 ?>
```

Autre avantage de cette dernière méthode; vous pouvez enregistrer l'instance au sein d'un conteneur de dépendances comme Pimple.

/*************** De l'usage correct des closures en PHP ***********/

Cet article a pour objet de vous verser dans l'art d'utiliser les closures, l'une des nouveautés les plus utiles de PHP 5.3. Il s'adresse à des développeurs chevronnés, au fait de la programmation orientée objet en PHP.

Je ne reviendrai pas sur les concepts de fonction, de référence ou de portée des variables ni sur le paradigme objet. Vous avez à votre disposition d'autres cours pour ça.

Tout au long de cet article, je parlerai de closures et de scopes et autres termes anglophones, j'ai choisi de les conserver dans leur langue originale car je trouve leurs équivalents français (fermetures et portées) moins parlants et surtout moins usités.

Fonctions lambda, closures, callbacks et objets invoquables
Il est souvent utile de définir des comportements lors du déroulement de l'application (runtime), par exemple pour filtrer un tableau, valider les champs d'un formulaire ou encore pour créer une fonction de rappel. Définir ainsi ces comportements facilite grandement la délégation, plutôt que de créer des fonctions à usage unique ou sous-typer des classes existantes pour incorporer le comportement voulu, on passe simplement le comportement. Cela réduit l'entropie du projet (moins de classes / fonctions) et améliore la lisibilité.

En PHP, la définition de comportements lors du runtime peut se faire de 3 façons distinctes (nous n'admettrons pas ici l'utilisation d'**eval** comme une option acceptable bien que techniquement réalisable).

créer une fonction anonyme à l'aide de create_function

```
1 <?php // version 4+
2
3 $additionner = create_function('$a,$b', 'return $a + $b;');
4 $trois = $additionner(1,2);
```

instancier une classe dottée de la méthode magique __invoke

```
 1 <?php // version 5.3+
 2
 3 class Additionneur {
 4     public function __invoke ($a, $b) {
 5         return $a + $b;
```

```
 6     }
 7 }
 8
 9 $additioner = new Additionneur;
10 $trois = $additioner(1,2);
```

créer une fermeture (closure)

```
1 <?php // version 5.3+
2
3 $additionner = function ($a, $b) {
4     return $a + $b;
5 };
6 $trois = $additionner(1,2);
```

C'est sur cette dernière forme que nous allons nous pencher. Outre le fait que cette syntaxe est proche de celle de JavaScript et qu'elle est considérablement plus simple à mettre en place, elle présente une caractéristique unique: la possibilité de manipuler les variables dans sa portée.

NOTE: Le manuel de PHP rassemble ces 3 concepts sous le terme générique de Callback et, depuis PHP 5.4, il est possible dans les prototypes de fonction de spécifier le type callable pour un paramètre, ce qui évite d'avoir recours systématiquement à la fonction is_callable.

```
1 <?php // version 5.4
2
3 function executer_fonction (callable $fonction) {
4     return $fonction();
5 }
6
7 var_dump( executer_fonction(function () { echo "Salut!"; }) ); // Salut!
8
9 var_dump( executer_fonction(123) ); // erreur: 123 n'est pas une fonction
```

Manipulation du scope

Ce qui différencie une closure d'une fonction anonyme ou d'une instance invoquable, c'est sa capacité à manipuler des références de la portée (ou scope) dans laquelle elle à été définie. Concrêtement, cela signifie que peu importe le scope dans lequel la closure est exécutée, elle peut se "souvenir" des variables présentes dans le scope où elle à été créée. Ceci est rendu possible grâce au mot clé use.

Note: La documentation de PHP ne fait pas de distinction claire entre fermetures et fonctions anonymes, ce n'est pourtant pas exactement la même chose en programmation. Référez vous aux articles Wikipedia pour plus de détails:

fermetures
fonctions anonymes

Exemple:

```
 1 <?php // version 5.3+
 2
 3 function fabrique_closure () {
 4     $a = 1;
 5     return function ($b) use ($a) {
 6         return $a + $b;
 7     };
 8 }
```

```php

function executer_closure ($f, $b) {
    return $f($b); // additionner
}

$f = fabrique_closure();

echo executer_closure($f, 2); // 3
```

Dans l'exemple ci dessus, on obtiens bien 3 car même si $a n'existe pas dans le scope de executer_closure, la closure se "souvient" de la référence originale.

C'est de cette fonctionnalité que les closures (fermetures) tirent leur nom: une closure "ferme" le scope parent, plus simplement c'est comme si elle était une boîte autour du scope dans lequel elle est définie.

Cela étant, si en JavaScript n'importe quelle variable du scope parent peut être référencée dans le scope de la closure, en PHP il faut explicitement définir avec le mot clé use quelles variables seront importées. Malgré cette limitation, c'est extrêment pratique car on peut promener la closure dans n'importe quel scope, sans se soucier de la visibilité des variables référencées.

Exemple:

```php
<?php // version 5.3+

list($a,$b) = array(1,2);

$f = function () use (&$a, &$b) {
    $a *= 2;
    $b *= 3;
};

$f();
var_dump($a,$b); // 2,6

function executer_closure ($f) {
    list($a,$b) = array(0,1);
    $f();
    var_dump($a,$b);
}

executer_closure($f); // 0,1
```

Il est intéressant de remarquer dans cet exemple que le scope d'exécution de la closure n'influe pas sur les variables référencées. Dans le scope de executer_closure, ce sont toujours les variables du scope racine qui sont référencées par la closure et non les deux nouvelles définies dans executer_closure. Nous n'avons donc pas à nous soucier du scope dans lequel notre closure est exécutée, il n'y a pas de risque d'écrasement involontaire.

Faire référence aux membres d'une instance dans une closure

En PHP 5.4

Depuis PHP 5.4, les closures peuvent désormais se servir du mot clé $this comme n'importe quelle méthode. Il est même possible d'utiliser des membres protégés, ce qui présente en soi un risque de violation de l'encapsulation mais ouvre de nouvelles possibilité pour la conception

orientée objet.

Pour que le mot clé $this soit utilisable dans le contexte d'une closure, celle-ci doit soit:

être définie dans une méthode d'instance
être explicitement "attachée" à l'objet
Exemple:

```php
<?php // version 5.4+

class MaClasse {

    public $a = 1;

    public function fabriqueClosure () {
        return function ($b) {
            return $this->a += $b;
        }
    }
}

$o = new MaClasse;
$f = $o->fabriqueClosure();
$r = $f(2);

var_dump( $o->a ); // 3

$o2 = (object)['a'=>2];
$f2 = $f->bindTo($o2);
$r2 = $f2(2);

var_dump( $o2->a ); // 4
```

Pour les habitués de JavaScript, la méthode bindTo représente en quelque sorte la méthode call: elle permet d'exécuter notre closure dans un autre contexte que celui d'origine (au détail près qu'une nouvelle closure nous est renvoyée au lieu d'utiliser la closure existante mais vous l'aviez compris j'en suis sûr).

Note: Vous l'avez sûrement remarqué mais bindTo est bien une méthode et donc, par voie de condéquence, notre closure un objet. Ce qui était en PHP 5.3 un détail d'implémentation est maintenant officiel: les closures sont des instances de la classe Closure et on peut donc utiliser le type-hinting. A noter également qu'on peut cloner une closure avec le mot clé clone.

Il est également possible de redéfinir le scope de notre closure pour utiliser celui d'une classe ou d'une instance. Concrêtement, cela va nous permettre d'adresser des membres protégés et privés. On passe pour cela un second paramètre à la méthode bindTo qui est l'instance ou le nom de la classe à utiliser comme nouveau scope pour notre closure.

En PHP 5.3
Si vous n'avez pas la possibilité d'utiliser PHP 5.4, vous pouvez injecter la référence de l'objet lors de l'exécution, ce que vous pouvez faire en encapsulant votre closure.

Exemple:

```php
<?php // version 5.3+

class SuperClosure {

    protected $_closure;
    protected $_object;

    public function __construct (Closure $closure, $object = null) {
        $this->_closure = $closure;

        if ($object) {
            if (!is_object($object))
                throw new InvalidArgumentException("object is not object");

            $this->_object = $object;
        }
    }

    public function bindTo ($object) {
        if (!is_object($object))
            throw new InvalidArgumentException("object is expected to be a valid instance");

        return new static($this->_closure, $object);
    }

    public function __invoke () {
        $args = array_merge(func_get_args(), array($this->_object));
        return call_user_func_array($this->_closure, $args);
    }
}

$f = new SuperClosure(function ($b, $that) {
    return $that->a *= $b;
});

$obj1 = (object)array('a'=>1);
$obj2 = (object)array('a'=>2);

$f1 = $f->bindTo($obj1);
$f2 = $f->bindTo($obj2);

$f1(2);
$f2(2);

var_dump( $obj1->a ); // 2
var_dump( $obj2->a ); // 4
```

Le prix à payer est assez lourd en revanche :

on perd la possibilité d'adresser les membres privés et protégés
les closures ne pouvant pas se servir de $this, il faut ajouter à leur prototype de fonction un paramètre qui caractérise l'instance (généralement appellé $that ou $self)
Pour aller plus loin, jetez un oeil à mon projet prototype.php, il fonctionne en PHP 5.3 et reproduit (en gros) le comportement mentionné ci-dessus.

Cas concrets d'utilisation

Les cas d'utilisation des closures sont nombreux, il s'agit généralement de déléguer un traitement sans pour autant définir une nouvelle structure (fonction) ou un nouveau type (classe). En voici quelques uns.

Filtrer un tableau à partir d'un paramètre

Imaginons qu'on veuille filtrer un tableau pour ne conserver qu'un élément sur N, nous allons pour cela utiliser la fonction **array_filter** qui attends une callback en tant que second paramètre.

Exemple:

```php
<?php // version 5.3+

// soit un tableau de 20 entrées
$tableau = range(1,20);

// soit la fonction de filtre qui utilise $n comme multiple
$filtre = function ($valeur) use (& $n) {
    return (int)$valeur % $n == 0;
};

// filtrer pour obtenir 1 élément sur 4
$n = 4;
$un_sur_quatre = array_filter($tableau, $filtre);

// filtrer pour obtenir 1 élément sur 10
$n = 10;
$un_sur_dix = array_filter($tableau, $filtre);

var_dump( $un_sur_quatre ); // [4,8,12,16,20]
var_dump( $un_sur_dix );    // [10,20]
```

Implémentation du pattern observer

Le pattern observer est extrêmement utile pour modéliser un mécanisme d'évènements. Depuis l'apparition de la SPL avec PHP 5.1, on dispose d'ailleurs des interfaces SplObserver et SplSubject dont nous allons nous servir. Comme il n'est pas pratique de créer une classe d'observateur pour chaque gestionnaire d'évènement, nous allons créer un observateur générique dont le comportement sera injecté avec une closure.

Exemple:

```php
<?php // version 5.4+

class SampleSubject implements SplSubject {

    protected $_name;
    protected $_observers;

    public function __construct ($name = false) {
        $this->_name = $name ?: uniqid(__CLASS__.'_');
        $this->_observers = new SplObjectStorage;
    }
```

```php

    public function attach (SplObserver $observer) {
        $this->_observers->attach($observer);
    }

    public function detach (SplObserver $observer) {
        $this->_observers->detach($observer);
    }

    public function notify () {
        foreach ($this->_observers as $observer)
            $observer->update($this);
    }

    public function __toString () {
        return $this->_name;
    }
}

class GenericObserver extends SplObjectStorage implements SplObserver {

    protected $_name;
    protected $_closure;

    public function __construct (Closure $closure, $name = false) {
        $this->_name = $name ?: uniqid(__CLASS__.'_');
        $this->_closure = $closure->bindTo($this, $this);
    }

    public function update (SplSubject $subject) {
        $closure = $this->_closure;
        $closure($subject);
    }

    public function __toString () {
        return $this->_name;
    }
}

// soient deux instance distinctes de SampleSubject
$subject_1 = new SampleSubject;
$subject_2 = new SampleSubject;

// soit une méthode destinée aux observeurs
$observer_function = function (SplSubject $subject) {
    echo "$this notifié par $subject\n";
};

// attachons les observateurs aux sujets...
$subject_1->attach(new GenericObserver($observer_function));
$subject_2->attach(new GenericObserver($observer_function));

// ...et voyons ce qui se passe
```

```
65 $subject_1->notify();
66 $subject_2->notify();
```

Dans l'exemple ci-dessus, nous avons utilisé la même closure pour définir deux observateurs différents. On aurait aussi bien plus utiliser le meme observateur pour différents sujets ou encore définir plusieurs observateurs pour chaque sujet.

Je vous recommande de vous approprier cet exemple pour créer vos propres observateurs au sein de votre application. Ce pattern est très utile pour réduire le couplage d'un architecture et est très pratique pour effectuer des tâches de logging, audit, gestion des erreurs / exceptions etc.

Des filtres pour nos itérateurs

Pour ceux qui ont déjà utilisé les itérateurs de la SPL, vous avez sûrement déjà rencontré la classe abstraite FilterIterator qui agit de la même manière que **array_filter** sur ces structures. Dans le même esprit que l'exemple précédent, nous allons implémenter un filtre générique pour nos itérateurs.

Exemple:

```
 1 <?php // version 5.4+
 2
 3 class GenericFilterIterator extends FilterIterator {
 4
 5     protected $_closure;
 6
 7     public function __construct (Iterator $iterator, Closure $closure) {
 8         $this->_closure = $closure->bindTo($this, $this);
 9         parent::__construct($iterator);
10     }
11
12     public function accept () {
13         $closure = $this->_closure;
14         return $closure();
15     }
16 }
17
18 // soit une séquence de 20 entrées
19 $it = new ArrayIterator(range(1,20));
20
21 // soit un filtre qui ne laisse passer
22 // que les nombres pairs
23 $filtre_1 = new GenericFilterIterator($it, function () {
24     return !($this->current() & 1);
25 });
26
27 echo "Nombres pairs de 1 à 20:\n"
28 foreach ($filtre_1 as $nombre)
29     echo "> $nombre\n";
30
31 // soit un filtre qui s'ajoute au
32 // précédent pour ne laisser passer
33 // que les multiples de 5
34 $filtre_2 = new GenericFilterITerator($filtre_1, function () {
```

```
35     return $this->current() % 5 == 0;
36 });
37
38 echo "Nombres pairs multiples de 5 de 1 à 20:\n";
39 foreach ($filtre_2 as $nombre)
40     echo "> $nombre\n";
```

Cet exemple nous montre combien il est facile d'appliquer un comportement identique à array_filter pour des itérateurs en utilisant des closures. On peut en plus enchaîner les filtres à l'infini, ce qui peut être pratique pour des moteurs de recherches multi-critères par exemple.

Note: Je ne vous recommande pas d'utiliser les filtres sur des résultats de requêtes MySQL, les performances ne seront jamais supérieures à l'utilisation de la clause WHERE SQL.

Des classes maléables

Les afficionados du pattern strategy ont dû être aux anges à l'annonce de la mise à jour des closures avec PHP 5.4. En effet, grâce aux nouvelles possibilité offertes par les traits et l'usage de $this dans les closures, il est désormais possible de rendre n'importe quelle classe existante dynamique.

Exemple:

```
 1 <?php // version 5.4+
 2
 3 trait DynamicObject {
 4
 5     protected $_customMethods = [];
 6
 7     public function method ($name, Closure $closure = null) {
 8         if (!$closure)
 9             return isset($this->_customMethods[$name]) ? $this->_customMethods[$name] : null;
10         else
11             $this->_customMethods[$name] = $closure->bindTo($this, $this);
12     }
13
14     public function __call ($method, $args = []) {
15         if (!$custom_method = $this->method($method))
16             throw new BadMethodCallException("no such method $method");
17
18         return call_user_func_array($custom_method, $args);
19     }
20 }
21
22 class MyClass {
23     use DynamicObject;
24
25     protected $_a;
26     protected $_b;
27
28     public function __construct ($a, $b) {
29         $this->_a = $a;
30         $this->_b = $b;
31     }
```

```
32 }
33
34 $f = function () {
35     return $this->_a + $this->_b;
36 };
37
38 $obj1 = new MyClass(1,2);
39 $obj1->method('add', $f);
40
41 $obj2 = new MyClass(4,8);
42 $obj2->method('add', $f);
43
44 var_dump( $obj1->add() ); // 3
45 var_dump( $obj2->add() ); // 12
```

Dans l'exemple ci-dessus, nous créeons un trait qui une fois équipé sur une classe lui permet de se voir doter de nouvelles méthodes dynamiquement. Ce genre de concept est exrêmement pratique pour injecter de nouveaux comportements à des types existants, ce qui réduit la nécéssité du sous-typage. Imaginez que vous disposiez d'une classe Collection qui représente une séquence (un tableau si vous préférez) et vous voulez une méthode qui trouve des informations dedans mais cette dernière est simple et ne justifie pas que vous sous-typiez Collection inutilement. Vous pouvez grâce à ce trait ajouter là où ça vous intéresse le comportement qui vous intéresse et l'affaire est dans le sac: pas de nouvelle classe à créer, pas de nouveau fichier, on reste simple, cohérent et lisible.

Aller plus loin

Les closures de par leur nature ont quelques application amusantes, voyons-en quelques unes.

De la récursivité avec les closures

Il est parfois utile d'exécuter la closure dans son propre corp afin, par exemple, d'explorer récursivement un arbre. Mais comment faire dès lors qu'une closure n'est pas nommé comme se doit de l'être une fonction ou méthode ? Tout simplement en utilisant sa propre référence.

Exemple:

```
 1 <?php // version 5.3+
 2
 3 $explorer = function (array $tableau, $profondeur = 0) use (& $explorer) {
 4     foreach ($tableau as $valeur) {
 5         if (is_array($valeur))
 6             $explorer($valeur, $profondeur +1);
 7         else
 8             echo str_repeat('  ', $profondeur) . "> $valeur\n";
 9     }
10 };
11
12 $arbre = array(
13     1,2,3,
14     array(
15         4,5,6,
16         array(
17             7,8,9,
18         )
19     )
```

```
20 );
21
22 $explorer($arbre);
```

Tricher avec la syntaxe nowdoc

Comme vous le savez, grâce à la syntaxe nowdow (et heredoc aussi) vous pouvez insérer des variables dans des chaines de caractère délimitées par des guillemets-doubles. Mais vous ne pouvez pas insérer des constantes et encore moins des expressions. Comme ce n'est pas toujours pratique d'utiliser des variables temporaires le temps de construire la chaîne, nous allons avoir recours à un hack assez sympathique.

Exemple:

```
 1 <?php // version 5.3+
 2
 3 $_ = function ($v) { return $v; };
 4
 5 define('FOO', 'bar');
 6 $a = 1;
 7
 8 // fonctionne
 9 echo "{$_(FOO)} - {$_($a+$a)}";
10
11 //fonctionne aussi
12 echo <<< EOF
13 Une expression: {$_($a+1)}
14 Une constante : {$_(FOO)}
15 EOF;
```

Un autoloader plus malin

Il arrive qu'on doive configurer des paramètre statique d'une classe juste après l'avoir chargée, comme par exemple exécuter la méthode setConfig d'un singleton. Mais on ne veux pas pour autant tout charger à chaque fois, alors comment faire ? En utilisant le chargement automatique de classes avec les closures, on va pouvoir enregistrer des comportements a effectuer avant et/ou après le chargement d'une classe.

Exemple:

```
 1 <?php // version 5.3+
 2
 3 spl_autoload_register(function ($classe) use (&$autoload_before, &$autoload_after) {
 4     isset($autoload_before[$classe]) && $autoload_before[$classe]($classe);
 5     $r = include $classe . '.class.php';
 6     $r && isset($autoload_after[$classe]) && $autoload_after[$classe]($classe);
 7 });
 8
 9 // configurer automatiquement un singleton
10 $autoload_after['MonSingleton'] = function () use (& $config) {
11     if (!isset($config))
12         throw new RuntimeException("you must setup configuration first");
13
14     MonSingleton::setConfig($config);
15 };
16
17 // créer dynamiquement un alias
```

```php
$autoload_before['MonAlias'] = function (& $vrai_nom) {
    $vrai_nom = 'MaClasse';
};

$autoload_after['MaClass'] = function () {
    class_alias('MonAlias', 'MaClasse');
}
```

Un injecteur de dépendances

L'injection de dépendances est un patron d'architecture proche de l'inversion de contrôle. Sans rentrer dans les détails, l'idée globale est de définir une cartographie des composants et de leurs dépendances au sein d'un composant capable de les résoudre pour fournir un composant donné. Par exemple, une classe de modèle (destinée à effectuer des requêtes sur la BDD) nécessite un adaptateur comme PDO ou encore un contrôleur à besoin d'accéder à des composants métiers. L'injecteur de dépendances, comme son nom l'indique permet d'exprimer ce besoin.

Voici un exemple d'implémentation qui repose sur les paramètres des closures pour déterminer les dépendances entre les composants. Il s'agit ici d'une résolution par nom où le chargement et l'initialisation d'un composant sont modélisés par une closure, les noms de ses paramètres sont ceux des dépendances du composants à charger.

```php
<?php // version 5.3+

class IoC {

    protected static $_registry = array();

    public static function register ($name, Closure $fct) {
        static::$_registry[(string)$name] = $fct;
    }

    public static function resolve ($name) {
        $fct = static::$_registry[(string)$name];
        return inject($fct);
    }
}

function inject (Closure $fct) {
    $reflect = new ReflectionFunction($fct);
    foreach ($reflect->getParameters() as $parameter) {
        $name = $parameter->name;
        $context[$name] = IoC::resolve($name);
    }
    return !empty($context) ? $reflect->invokeArgs($context) : $reflect->invoke();
}

IoC::register('Composant_A', function ()            { echo "Composant A initialisé\n"; });
IoC::register('Composant_B', function ($Composant_A) { echo "Composant B initialisé\n"; });
IoC::register('Composant_C', function ($Composant_B) { echo "Composant C initialisé\n"; });

inject(function ($Composant_C) {
    // ici, on dispose du composant C qui dépends de B qui dépends de A
    // la chaîne de dépendance à été résolue récursivement par IoC
});
```

On peut sans grand effort effectuer une résolution par type en utilisant des **interface** (ce qui est bien mieux en termes de sécurité), je suis sûr que vous saurez adapter l'exemple ci-dessous en conséquence.

L'avantage marteau de l'injecteur de dépendances est le total découplage entre les classes au prix bien sûr d'une inversion de contrôle car il faut bien que l'injection soit contrôlée par quelque chose. Mon projet Cobalt utilise cette forme d'injection de dépendances pour les contrôleurs, la flexibilité apportée est extrêmement appréciable dans le contexte du MVC.

Conclusion

Je crois avoir suffisamment démontré à quel point les closures sont à la fois simples et puissantes, elles permettent de répondre à nombre de problématiques existantes et ouvrent de nouvelles pistes de réflexion pour les designs à venir. Pour moi, il n'est pas étonnant que le concept de closures soit central dans de nombreux langages, notamment JavaScript, compte tenu de la flexibilité qu'elles apportent.

Je vous recommande maintenant de faire quelques essais ou au moins d'exécuter les exemples ci-dessus (environement PHP 5.4 obligatoire), de les comprendre et de les maîtriser. Vous verrez, une fois qu'on a commencé à programmer avec des closures, on ne peut plus s'en passer.

Je terminerai simplement en vous mettant en garde contre l'utilisation abusive de ce concept qui peut mener, comme on s'en doute, à une "spaghettisation" du code. Sachez donc les utiliser convenablement et à ne pas en abuser.