



La programmation orientée objet en PHP

Informations sur le tutoriel



Auteur : [vyk12](#)

Difficulté :

Temps d'étude estimé : 1 mois

Licence :

[Plus d'informations](#)

Popularité

Visualisations : 1 599 928

Appréciation 1

des lecteurs : 5

19

90

126 personnes souhaitent voir ce tutoriel publié en livre !

[Vous aussi ?](#)

Historique des mises à jour

- Le 15/09/2010 à 21:45:57
Modification design pattern Singleton (+ injection de dépendances) + modification TP News
- Le 14/08/2010 à 22:03:07
Réorganisation des chapitres sur l'héritage
- Le 02/07/2010 à 23:28:40
ticket par MP

Bienvenue dans ce tutoriel portant sur la programmation orientée objet (souvent abrégé par ses initiales POO) en PHP. 😊

Vous ne savez pas ce qu'est la POO ? Ne vous inquiétez pas, je vais me charger de vous en apprendre le principe du début et le maximum possible. 😊

Ce qui doit être acquis

Afin de suivre au mieux ce tutoriel, il est indispensable voire obligatoire :

- D'être à l'aise avec PHP et sa syntaxe ;
- D'avoir bien pratiqué ;
- D'être patient ;
- D'avoir PHP 5 sur son serveur. Je ne parlerai pas de POO en PHP 4 car sous cette version de PHP, certaines fonctions indispensables de la POO ne sont pas présentes (on ne peut donc pas vraiment parler de POO).

Si vous avez déjà pratiqué d'autres langages apportant la possibilité de programmer orienté objet, c'est un gros plus, surtout si vous savez programmer en Java (PHP a principalement tiré son modèle objet de ce langage).

Remerciements

Je tiens à dire merci à toutes ces personnes qui ont contribué d'une façon ou d'une autre à la qualité du tutoriel :

- [Talus](#) et [Lpu8er](#) concernant l'UML ;
- [christophetd](#) pour la correction orthographique de quelques chapitres ;
- [prs513rosewood](#) pour son aide concernant l'installation de Dia sous Mac OS ;
- [waaaou](#) pour avoir créé les icônes du tutoriel ;
- Et à tous les autres qui ont posté un commentaire, soit pour m'encourager, soit pour donner leur avis et permettre ainsi l'amélioration de la qualité du tutoriel !

Questions

Je reçois beaucoup de MP me demandant des précisions sur le cours. Si vous êtes dans ce cas-là, je vous invite à poster dans [le topic dédié au tutoriel](#). Je peux ainsi voir au sein du même sujet les problèmes auxquels les membres sont le plus souvent confrontés. 😊

Ce cours est composé des parties suivantes :

- [Les bases de la POO](#)
 - [Techniques avancées](#)
 - [Bonnes pratiques](#)
 - [Annexes](#)
-

Partie 1 : Les bases de la POO

Voici une première partie qui vous expliquera les bases de la POO en PHP. Vous ne connaissez rien à la POO en PHP ? Aucun problème, tout commence de Zéro ! 🐧

○

[1\) Introduction à la POO](#)



- [La POO, kékako ?](#)
- [Créer une classe](#)

○

[2\) Utiliser la classe](#)



- [Implémenter ses méthodes](#)
- [Créer et manipuler un objet](#)
- [Le constructeur](#)

○

[3\) L'opérateur de résolution de portée ::](#)



- [Les constantes de classe](#)
- [Les attributs et méthodes statiques](#)

○

[4\) TP : Mini-jeu de combat](#)



- [Ce qu'on va faire](#)
- [Correction](#)

○

[5\) L'héritage](#)



- [Notion d'héritage](#)
- [Un nouveau type de visibilité : protected](#)
- [Imposer des contraintes](#)
- [Résolution statique à la volée](#)

○

[6\) TP : Des personnages spécialisés](#)



- [Ce qu'on va faire](#)
- [Correction](#)

○

[7\) Les méthodes magiques](#)



- [Le principe](#)
- [Surcharger les attributs et méthodes](#)
- [Linéariser ses objets](#)
- [Autres méthodes magiques](#)

Voilà cette première partie posant les bases terminée. Avec les connaissances que vous avez, vous pouvez déjà créer de belles classes. 😊

•

Partie 2 : Techniques avancées

Vous croyez tout savoir sur la POO en PHP ? Vous êtes loin du compte ! Cette partie vous montrera beaucoup de choses possibles avec la POO. 😊

○

[1\) Les objets en profondeur](#)



- [Un objet, un identifiant](#)
- [Comparons nos objets](#)
- [Parcourons nos objets](#)

○

[2\) Les interfaces](#)



- [Présentation et création d'interfaces](#)
- [Hériter ses interfaces](#)
- [Interfaces prédéfinies](#)

○

[3\) Les exceptions](#)



- [Une différente gestion des erreurs](#)
- [Des exceptions spécialisées](#)
- [Gérer les erreurs facilement](#)

○

4) L'API de réflexivité



- [Obtenir des informations sur ses classes](#)
- [Obtenir des informations sur les attributs de ses classes](#)
- [Obtenir des informations sur les méthodes de ses classes](#)

Voici cette deuxième partie terminée. Je suis heureux de vous annoncer que vous avez toutes les connaissances requises pour faire ce que vous voulez ! 😊

Aucune syntaxe ou notion n'a été laissée de côté, je vous ai parlé de tout (ou du moins, presque tout). Cependant, il vous reste quelques choses à voir en relation avec la POO. Rendez-vous en troisième partie. 😊

Partie 3 : Bonnes pratiques

Cette troisième partie a pour but de vous aider à vous organiser et vous présenter de bonnes pratiques. Vous n'apprendrez donc pas de nouvelles notions ou de nouvelle syntaxe en PHP : vous allez apprendre à organiser votre code. Vous découvrirez quelques trucs et astuces et enfin, après ça, la super récompense : un TP qui aura pour objectif de créer... un système de news orienté objet ! Eh oui, je pense que maintenant, vous êtes fin prêts. 😊

Alors, on y va ? 😊

○

1) UML : présentation (1/2)



- [UML, kézako ?](#)
- [Modéliser une classe](#)
- [Modéliser les interactions](#)

○

2) UML : modélisons nos classes (2/2)



- [Ayons les bons outils](#)
- [Modéliser une classe](#)
- [Modéliser les interactions](#)
- [Exploiter son diagramme](#)

○

3) Les design patterns



- [Laisser une classe créant les objets : le pattern Factory](#)
- [Écouter ses objets : le pattern Observer](#)
- [Séparer ses algorithmes : le pattern Strategy](#)
- [Une classe, une instance : le pattern Singleton](#)

○

4) TP : un système de news



- [Ce qu'on va faire](#)
- [Correction](#)

Partie 4 : Annexes

Voici ici quelques chapitres vous présentant quelques notions que je n'ai pas pu glisser dans le cours au risque de compliquer celui-ci inutilement. Au début de chaque chapitre seront précisés les pré-requis afin de pouvoir suivre sans difficulté. 😊

○

1) L'opérateur instanceof



- [Présentation de l'opérateur](#)
- [instanceof et l'héritage](#)
- [instanceof et les interfaces](#)

Partie 1 : Les bases de la POO

Voici une première partie qui vous expliquera les bases de la POO en PHP. Vous ne connaissez rien à la POO en PHP ? Aucun problème, tout commence de Zéro ! 😊

Introduction à la POO

Vous voici dans le premier chapitre de ce tutoriel. On va commencer par découvrir ce qu'est la POO en PHP, ce à quoi ça peut bien servir puis on codera un petit peu. 😊

La POO, kézako ?

Il était une fois, le procédural

Commençons ce cours en vous posant une question : comment est représenté votre code ? La réponse est unique : vous avez utilisé la représentation procédurale. Qu'est-ce donc que cette représentation ? La représentation procédurale consiste à séparer le traitement des données des données elles-mêmes. Par exemple, vous avez un système de news sur votre site. D'un côté, vous avez les données (les news, une liste d'erreurs, une connexion à la BDD, etc.) et de l'autre côté vous avez une suite d'instructions qui viennent modifier ces données. Si je ne me trompe pas, c'est de cette manière que vous codez.

Cette façon de se représenter votre application vous semble sans doute la meilleure puisque c'est la seule que vous connaissez. D'ailleurs, vous ne voyez pas trop comment votre code pourrait être représenté de manière différente. Eh bien cette époque d'ignorance est révolue : voici maintenant la programmation orientée objet.

Puis naquit la programmation orientée objet

Alors, qu'est-ce donc que cette façon de représenter son code ? La POO, c'est tout simplement faire de son site un ensemble d'objets qui interagissent entre eux. En d'autres termes : tout est objet.

Définition d'un objet

Je suis sûr que vous savez ce que c'est. D'ailleurs, vous en avez pas mal à côté de vous : je suis sûr que vous avez un ordinateur, une lampe, une chaise, un bureau, ou que sais-je encore. Tout ceci sont des objets. En programmation, les objets sont sensiblement les mêmes choses.

L'exemple le plus pertinent quand on fait un cours sur la POO est d'utiliser l'exemple du personnage dans un jeu de combat. Ainsi, imaginons que nous ayons un objet Personnage dans notre application. Un personnage a des caractéristiques :

- Une force ;
- Une localisation ;

- Une certaine expérience ;
- Et enfin des dégâts.

Toutes ses caractéristiques correspondent à des valeurs. Comme vous le savez sûrement, les valeurs sont stockées dans des variables. C'est toujours le cas en POO. Ce sont des variables un peu spéciales, mais nous y reviendrons plus tard. 😊

Mis à part ces caractéristiques, un personnage a aussi des capacités. Il peut :

- Frapper un autre personnage ;
- Gagner de l'expérience ;
- Se déplacer.

Ces capacités correspondent à des fonctions. Comme pour les variables, ce sont des fonctions un peu spéciales et on y reviendra en temps voulu. Bref, le principe est là en tous cas.

Vous savez désormais qu'on peut avoir des objets dans une application. Mais d'où ils sortent ? Dans la vie réelle, un objet ne sort pas de nulle part. En effet, il faut une sorte de machine qui construit des objets. En POO, ces machines sont appelées classes.

Définition d'une classe

Comme je viens de le dire, les classes sont des machines à fabriquer des objets. Prenons l'exemple le plus simple du monde : les gâteaux et leur moule. Le moule, il est unique. Il peut produire une quantité infinie de gâteaux. Dans ces cas-là, les gâteaux sont les objets et le moule est la classe. La classe est donc une sorte de machine à produire des objets. Elle contient le plan de fabrication d'un objet et on peut s'en servir autant qu'on veut afin d'obtenir une infinité d'objets.

Concrètement, une classe, c'est quoi ?

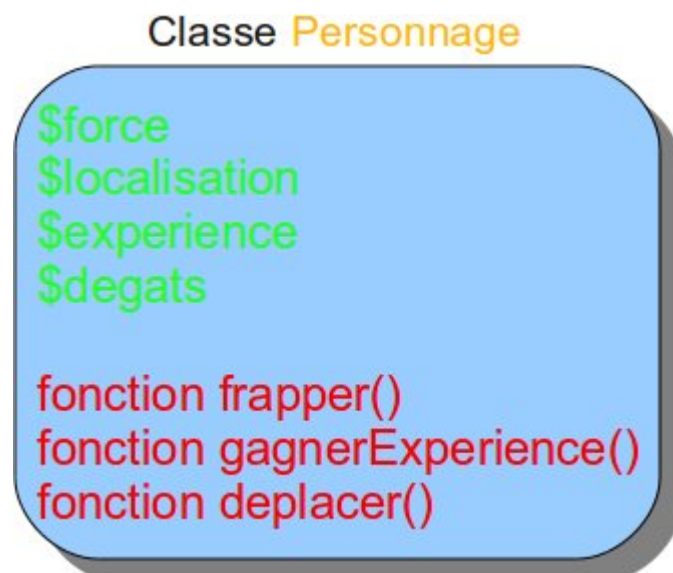
Une classe est une entité regroupant des variables et des fonctions. Chacune de ces fonctions aura accès aux variables de cette entité. Dans le cas du personnage, nous aurons une fonction `frapper()`. Cette fonction devra simplement modifier la variable `$degats` du personnage en fonction de la variable `$force`. Une classe est donc un regroupement logique de variables et fonctions que tout objet issu de cette classe possèdera.

Définition d'une instance

Une instance, c'est tout simplement le résultat d'une instanciation. Une instanciation, c'est le fait d'instancier une classe. Instancier une classe, c'est se servir d'une classe afin qu'elle nous crée un objet. En gros, une instance c'est un objet. 😊

Exemple : création d'une classe Personnage

On va désormais créer une classe Personnage (sous forme de schéma bien entendu). Celle-ci doit contenir la liste des variables et des fonctions que l'on a citées plus haut : c'est la base de tout objet Personnage. Chaque instance de cette classe possèdera ainsi toutes ces variables et fonctions. Voici donc cette fameuse classe :



Vous voyez donc les variables et fonctions stockées dans la classe Personnage. Sachez qu'en réalité, on ne les appelle pas comme ça : il s'agit d'attributs (ou propriétés) et de méthodes. Un attribut désigne une variable de cette classe et une méthode désigne une fonction de celle-ci.

Ainsi, tout objet Personnage aura ces attributs et méthodes. On pourra modifier ces attributs et invoquer ces méthodes sur notre objet afin de modifier ses caractéristiques ou son comportement.

Le principe d'encapsulation

L'un des gros avantages de la POO est que l'on peut masquer le code à l'utilisateur (l'utilisateur est ici celui qui se servira de la classe, pas celui qui chargera la page depuis son navigateur). Le concepteur de la classe a englobé dans celle-ci un code qui peut être assez complexe et il est donc inutile voire dangereux de laisser l'utilisateur manipuler ces objets sans aucune restriction. Ainsi, il est important d'interdire à l'utilisateur de modifier directement les attributs d'un objet.

Prenons l'exemple d'un avion où sont disponibles des centaines de boutons. Chacun de ces boutons constituent des actions que l'on peut effectuer sur l'avion. C'est l'interface de l'avion. Le pilote se moque de quoi est composé l'avion : son rôle est de le piloter. Pour cela, il va se servir des boutons afin de manipuler les composants de l'avion. Le pilote ne doit pas se charger de modifier manuellement ces composants : il pourrait faire de grosses bêtises.

Le principe est exactement le même pour la POO : l'utilisateur de la classe doit se contenter d'invoquer les méthodes en ignorant les attributs. Il s'en fiche : comme le pilote de l'avion, il n'a pas à les trifouiller. Pour instaurer une telle contrainte, on dit que les attributs sont privés, et c'est primordial. C'est l'un des piliers fondamentaux de la POO.

Tout attribut doit être privé

Oui, c'est très important : c'est la base même de la POO. 😊

Bon, je pense que j'ai assez parlé, on va commencer par créer notre première classe. 😊

Créer une classe

Syntaxe de base

Le but de cette sous-partie va être de traduire l'image que je vous ai donnée en code PHP. Avant cela, je vais vous donner la syntaxe de base de toute classe en PHP. La syntaxe, elle, est très simple, la voici :

Code : PHP

```
<?php
class Personnage // Présence du mot-clé class suivi du nom de la classe.
{
    // Déclaration des attributs et méthodes ici.
}
?>
```

Cette syntaxe est à retenir absolument. J'espère pour vous que vous avez retenu sinon vous êtes mal partis. 😊

Ce qu'on vient de faire est donc de créer le moule, la machine qui fabriquera divers objets. On verra dans le prochain chapitre comment utiliser cette machine afin d'en faire sortir un objet. Pour l'instant, on se contente de construire la machine et de lui ajouter des fonctionnalités. 😊

La déclaration d'attributs dans une classe se fait en écrivant le nom de l'attribut à créer, précédé de sa visibilité.

La visibilité d'un attribut ou d'une méthode

La visibilité d'un attribut ou d'une méthode indique à partir d'où on peut avoir accès à telle méthode ou tel attribut. Nous allons voir ici 2 types de visibilité : `public` et `private`.

Le premier, `public`, est le plus simple. Si un attribut ou une méthode est `public`, alors on pourra avoir accès à cet attribut ou cette méthode depuis n'importe où, autant depuis l'intérieur de l'objet (dans les méthodes qu'on a créées, on aura accès aux éléments `public`), mais aussi depuis l'extérieur. Je m'explique. Quand on créera un objet, c'est principalement pour pouvoir exploiter ses attributs et méthodes. L'extérieur de l'objet, c'est tout le code qui n'est pas dans votre classe. En effet, quand vous créerez un objet, cet objet sera représenté par une variable, et c'est à partir d'elle qu'on pourra modifier l'objet, appeler des méthodes, etc. Vous allez donc dire à PHP « dans cet objet, donne-moi cet attribut » ou « dans cet objet, appelle cette méthode » : c'est ça, appeler des attributs ou méthodes depuis l'extérieur de l'objet. 😊

Le second, `private`, impose quelques restrictions. On aura accès aux attributs et méthodes que depuis l'intérieur de la classe, c'est-à-dire que seul le code voulant accéder à un attribut privé ou une méthode privée écrit(e) à l'intérieur de la classe fonctionnera. Sinon, une jolie erreur fatale s'affichera disant que vous ne devez pas accéder à telle méthode ou tel attribut parce qu'il ou elle est privé(e).

Là, ça devrait faire tilt dans votre tête : le principe d'encapsulation. C'est de cette manière qu'on peut interdire l'accès à nos attributs. 😊

Création d'attributs

Pour déclarer des attributs, on va donc les écrire entre les accolades, les uns à la suite des autres, en faisant précéder leurs noms du mot-clé `private`, comme ça :

Code : PHP

```
<?php
class Personnage
{
    private $force; // La force du personnage.
    private $localisation; // Sa localisation.
    private $experience; // Son expérience.
    private $degats; // Ses dégâts.
}
?>
```

Vous pouvez initialiser les attributs lorsque vous les déclarez (par exemple, leur mettre une valeur de 0 ou je ne sais quoi). Exemple :

Code : PHP

```
<?php
class Personnage
{
    private $force = 50; // La force du personnage, par défaut à 50.
    private $localisation = 'Lyon'; // Sa localisation, par défaut à Lyon.
    private $experience = 1; // Son expérience, par défaut à 1.
    private $degats = 0; // Ses dégâts, par défaut à 0.
}
?>
```

La valeur que vous leur donnez par défaut doit être une expression. Par conséquent, leur valeur ne peut être issue d'un appel à une fonction (`private $attribut = intval('azerty')`), d'une opération (`private $attribut = 1 + 1`) ou d'une concaténation (`private $attribut = 'Mon ' . 'super ' . 'attribut'`).

Création de méthodes

Pour la déclaration de méthodes, il suffit de faire précéder le mot-clé `function` de la visibilité de la méthode. Les types de visibilité des méthodes sont les mêmes que les attributs. Les méthodes n'ont en général pas besoin d'être masquées à l'utilisateur, vous les mettez en général en `public` (à moins que vous teniez absolument à ce que l'utilisateur ne puisse pas appeler cette méthode, par exemple si il s'agit d'une fonction qui simplifie certaines tâches sur l'objet mais qui ne doit pas être appelée n'importe comment).

Code : PHP

```
<?php
class Personnage
{
    private $force; // La force du personnage.
    private $localisation; // Sa localisation.
    private $experience; // Son expérience.
    private $degats; // Ses dégâts.

    public function deplacer() // Une méthode qui déplacera le personnage (modifiera
    {

    }

    public function frapper() // Une méthode qui frappera un personnage (suivant la
    {

    }

    public function gagnerExperience() // Une méthode augmentant l'attribut $experien
    {

    }
}
```

Et voilà. 😊

Par convention de nommage, le nom de votre classe doit commencer par une majuscule. Ce n'est pas obligatoire, c'est juste une convention. De même pour la déclaration des attributs avant la déclaration des méthodes : ce n'est pas obligatoire mais plus lisible.



Ce premier chapitre s'arrête là. Il est très important que vous compreniez bien ce qu'est la POO et la syntaxe de base, sans quoi vous ne pourrez pas suivre.

Ce qu'il faut retenir :

- La différence entre « classe » et « objet ». Une classe, c'est un ensemble de variables et fonctions (attributs et méthodes). Un objet, c'est une instance de la classe pour pouvoir l'utiliser.
- Le principe d'encapsulation. Tous vos attributs doivent être privés. Pour les méthodes, peu importe leur visibilité.
- On déclare une classe avec le mot-clé `class` suivi du nom de la classe, et enfin deux accolades ouvrantes et fermantes qui encercleront la liste des attributs et méthodes.

Utiliser la classe

Bien, on a créé notre classe, cool, mais... nos méthodes sont vides ! Dans ce chapitre, nous allons voir comment les implémenter, c'est-à-dire écrire du code dedans.

Nous verrons enfin comment utiliser notre classe : on l'instanciera, on créera un objet quoi ! 😊

Implémenter ses méthodes

Bien. On a notre classe toute faite mais... les méthodes ne contiennent rien, ce qui est assez dommage vous avouerez. 🤔

Quand nous implémenterons nos méthodes, nous aurons besoin d'avoir accès aux attributs de notre objet et de les modifier. Nous allons avoir besoin de la pseudo-variable `$this`.

La pseudo-variable \$this

Qu'est-ce que c'est que cette bête-là ? La pseudo-variable `$this` est une simple variable disponible dans toutes les méthodes lorsque l'on se servira de notre classe (ce que nous verrons plus tard). La variable `$this` est tout simplement l'objet que nous sommes en train d'utiliser, elle le représente. 🤖

Puisque la variable `$this` représente l'objet créé, elle contient ainsi tous les attributs de celui-ci. C'est à partir de cette variable que l'on pourra appeler des méthodes de notre objet ou modifier un attribut. Comment avoir accès à un attribut ou appeler une méthode avec `$this` ? Grâce à l'opérateur `->` !

L'opérateur `->` doit être placé juste après `$this`. Cet opérateur signifie « dans cet objet, je veux cet attribut » ou « dans cet objet, je veux appeler telle méthode ». Ainsi, le code pourrait ressembler à ceci :

Code : PHP

```
<?php
class Personnage
{
    private $force;
    private $localisation;
    private $experience;
    private $degats;

    public function deplacer()
    {

    }

    public function frapper()
    {

    }

    public function gagnerExperience()
    {
        $this->experience = $this->experience + 1;
    }
}
?>
```

Des petites explications s'imposent. Ici j'ai décidé d'augmenter de 1 l'expérience de mon personnage à chaque appel de la méthode `gagnerExperience`. J'ai donc fait appel à l'attribut `experience` de mon objet pour l'augmenter de 1. La pseudo-variable `$this` pourrait donc être traduite par « Cet objet ». Ainsi, `$this->experience` pourrait être traduit par « Dans cet objet, je veux l'attribut `experience` ». Enfin, la formule complète de la ligne surlignée dans le code pourrait être traduite par « Dans cet objet, assigne à l'attribut `experience` cette valeur : dans cet objet, donne-moi l'attribut `experience` et ajoute 1 au résultat ».

J'ai mis la manière la plus longue d'écrire cette ligne surlignée pour ne pas vous embrouiller. Sachez que ces 2 manières reviennent au même :

Code : PHP

```
<?php
$this->experience += 1;
$this->experience++;
?>
```

Voilà tout ce que j'avais à dire. Libre à vous d'implémenter vos méthodes comme bon vous semble. 😊

Ne faites jamais précéder le nom de l'attribut du signe `$` lorsque vous l'appellez de cette façon !

Petit exercice

Oublions notre classe `Personnage` un petit moment pour nous concentrer sur une vision plus globale et ainsi créer un petit exercice pour bien mettre en application ce qu'on vient de voir. Je voudrais une classe qui contienne un attribut : `$texte`, par défaut à 'Hello world !', et une méthode : `changerTexte()`. Cette méthode, à chaque appel, doit changer la valeur de l'attribut `$texte`. Si celui-ci vaut 'Hello world !', alors l'attribut devra avoir pour valeur 'Bonjour tout le monde !', et vice-versa. Pas très utile, certes, mais ça fait pratiquer. 😊

La correction :

Secret ([cliquez pour afficher](#))

Exiger un type d'argument

Voici un dernier point que je voudrais aborder. Quand vous passez des paramètres à une méthode, vous pouvez exiger de ces paramètres un type à respecter. Par exemple, on peut exiger de la méthode `frapper` un paramètre de type `Personnage` (car on veut frapper un personnage 😊). Ainsi, le code pourrait ressembler à ceci :

Code : PHP

```
<?php
class Personnage
{
    private $force;
    private $localisation;
    private $experience;
    private $degats;

    public function deplacer()
    {

    }

    public function frapper (Personnage $persoAFrapper) // On veut un paramètre de type Pers
    {
        $persoAFrapper->recevoirDegats ($this->force);
    }

    public function gagnerExperience()
    {
        $this->experience++;
    }

    public function recevoirDegats ($force)
    {
        $this->degats += $force;
    }
}
```

Ici, quand on appellera la méthode `frapper`, on sera obligé de lui passer un objet de type `Personnage` car l'argument est précédé du mot `Personnage`. 😊

Je vais vous expliquer un peu plus en profondeur la ligne surlignée au risque de perdre des zéros en route. Reprenons le code en simplifié (suppression des autres méthodes et du mot `Personnage` précédant le paramètre `$persoAFrapper`) :

Code : PHP

```

<?php
class Personnage
{
    private $force;

    public function frapper ($persoAFrapper)
    {
        $persoAFrapper->recevoirDegats ($this->force);
    }
}
?>

```

Normalement, ce code ne doit pas poser de problème. La méthode **frapper** reçoit un argument : `$persoAFrapper`. Cet argument est un argument tout ce qu'il y a de plus banal. Dans cette méthode, l'argument est un objet, on va donc appeler la méthode **recevoirDegats** de cet objet. Jusque-là, rien de compliqué. Mais un problème se pose : imaginons que dans le script, on passe un nombre comme argument à cette méthode, et non un objet de type `Personnage`. Ceci causera une erreur fatale car on appelle une méthode sur un nombre ! Il faut donc trouver le moyen d'obliger l'utilisateur à passer un objet de type `Personnage` et pas autre chose. On va donc pour cela précéder l'argument `$persoAFrapper` du mot `Personnage` : pour PHP, ça veut dire que l'argument `$persoAFrapper` doit obligatoirement être un objet de type `Personnage` ! Sinon, PHP interrompt le script.

Code : PHP

```

<?php
class Personnage
{
    private $force;

    public function frapper (Personnage $persoAFrapper) // Le mot Personnage précédant
    {
        $persoAFrapper->recevoirDegats ($this->force);
    }
}
?>

```

Le type de la variable à spécifier doit obligatoirement être un nom de classe ou alors un tableau. Si vous voulez exiger un tableau, faites précéder le nom du paramètre devant être un tableau du mot-clé `array` comme ceci : `public function frapper (array $coups)`. Vous ne pouvez pas exiger autre chose : par exemple, il est impossible d'exiger un nombre entier ou une chaîne de caractères de cette façon.

Créer et manipuler un objet

On va donc voir comment créer un objet, c'est-à-dire que l'on va utiliser notre classe (qui peut être comparée à une machine) afin qu'elle nous fournisse un objet. Pour créer un nouvel objet, vous devez faire précéder le nom de la classe à instancier du mot-clé `new`, comme ceci :

Code : PHP

```

<?php
$perso = new Personnage();
?>

```

Ainsi, `$perso` sera un objet de type `Personnage`. On dit que l'on instancie la classe `Personnage`, qu'on crée une instance de la classe `Personnage`.

Auto-chargement des classes

Pour une question d'organisation, il vaut mieux créer un fichier par classe. Vous appelez votre fichier comme bon vous semble et vous placez votre classe dedans. Pour ma part, mes fichiers sont toujours appelés `MaClasse.class.php`.

Euh, cool, et alors ?

Si vous essayez de créer une instance de la classe alors que vous n'avez pas inclus le fichier déclarant la classe, une erreur fatale sera levée pour dire que vous instanciez une classe qui n'existe pas. Et c'est là qu'intervient l'auto-chargement des classes. Vous pouvez créer dans votre fichier principal (c'est-à-dire celui où vous créez une instance de votre classe) une ou plusieurs fonction(s) qui tenteront de charger le fichier déclarant la classe. Dans la plupart des cas, une seule fonction suffit. Ces fonctions doivent accepter un paramètre, c'est le nom de la classe qu'on doit tenter de charger. Par exemple, voici une fonction qui aura pour rôle de charger les classes :

Code : PHP

```
<?php
function chargerClasse ($classe)
{
    require $classe . '.class.php'; // On inclue la classe correspondante au paramètre
}
```

Essayons maintenant de créer un objet pour voir si il sera chargé automatiquement (je prends pour exemple la classe Personnage et prends en compte le fait qu'un fichier Personnage.class.php est créé).

Code : PHP

```
<?php
function chargerClasse ($classe)
{
    require $classe . '.class.php'; // On inclue la classe correspondante au paramètre
}

$perso = new Personnage(); // Instanciation de la classe Personnage qui n'est pas déclarée
```

Et là... Bam ! Erreur fatale ! La classe n'a pas été trouvée, elle n'a donc pas été chargée... Normal quand on y réfléchit ! PHP ne sait pas qu'il doit appeler cette fonction lorsqu'on essaye d'instancier une classe non déclarée. On va donc utiliser la fonction `spl_autoload_register` en spécifiant en premier paramètre le nom de la fonction à charger :

Code : PHP

```
<?php
function chargerClasse ($classe)
{
    require $classe . '.class.php'; // On inclue la classe correspondante au paramètre
}

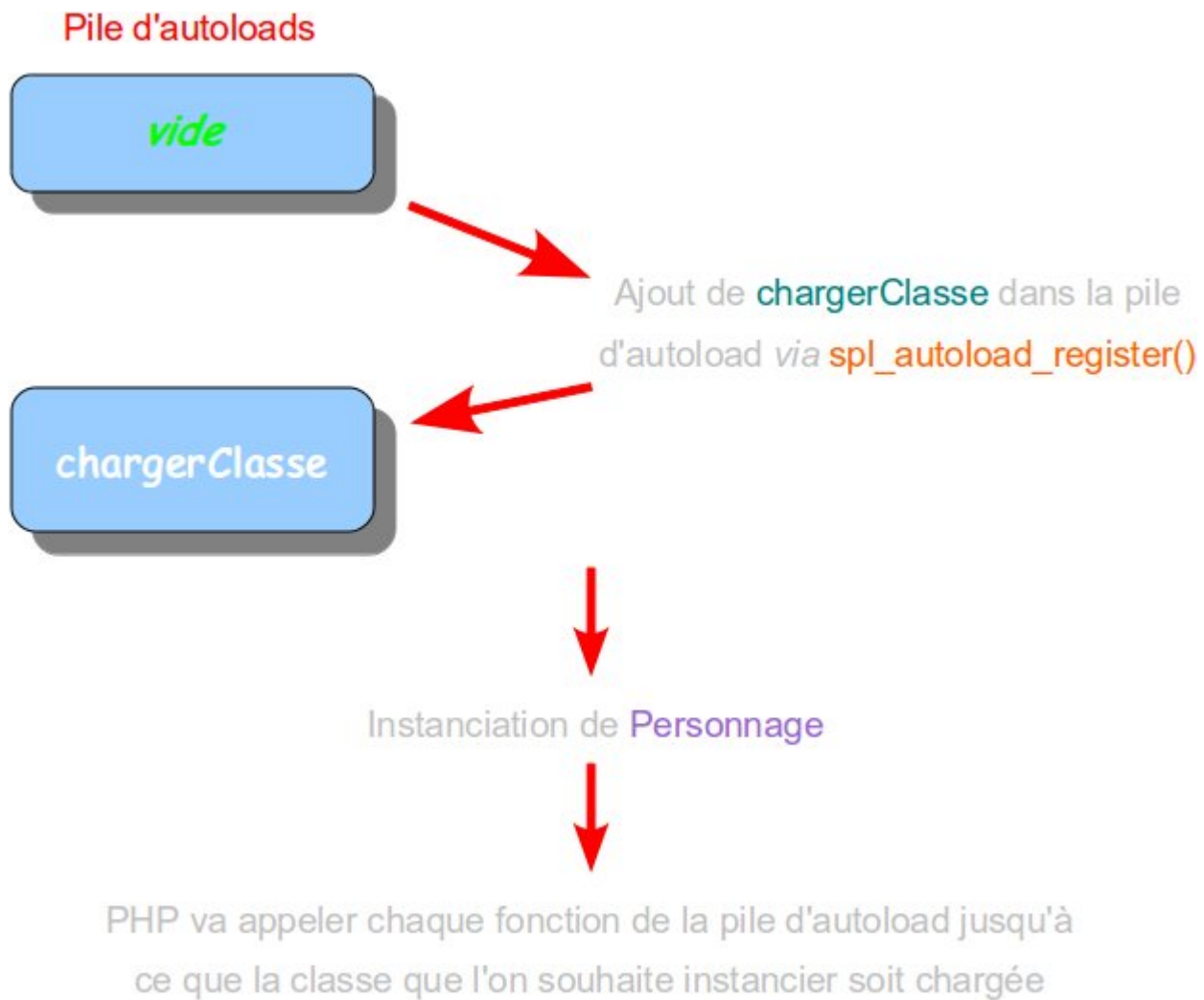
spl_autoload_register ('chargerClasse'); // On enregistre la fonction en autoload pour PHP

$perso = new Personnage();
```

Et là, comme par magie, aucune erreur ne s'affiche ! Notre auto-chargement a donc bien fonctionné. 😊

Décortiquons ce qui s'est passé. En PHP, il y a ce qu'on appelle une pile d'autoloads. Cette pile contient une liste de fonctions. Chacune d'entre elles sera appelée automatiquement par PHP lorsque l'on essaye d'instancier une classe non déclarée. Nous avons donc ici ajouté notre fonction à la pile d'autoloads afin qu'elle soit appelée à chaque fois qu'on essaye d'instancier une classe non déclarée.

Schématiquement, voici ce qui s'est passé :



Sachez que vous pouvez enregistrer autant de fonctions en autoload que vous le voulez avec `spl_autoload_register`. Si vous enregistrez plusieurs, elles seront appelées dans l'ordre de leur enregistrement jusqu'à ce que la classe soit chargée. Pour y parvenir, il suffit d'appeler `spl_autoload_register` pour chaque fonction à enregistrer.

Appeler les méthodes de l'objet

Tout d'abord, je vous remets notre classe `Personnage` de tout à l'heure :

Code : PHP

```

<?php
class Personnage
{
    private $force;
    private $localisation;
    private $experience;
    private $degats;

    public function deplacer()
    {

    }

    public function frapper (Personnage $persoAFrapper) // On veut un paramètre de type Personnage
    {
        $persoAFrapper->recevoirDegats ($this->force);
    }

    public function gagnerExperience()
    {
        $this->experience++;
    }

    public function recevoirDegats ($force)
    {
        $this->degats += $force;
    }
}

```

Il serait intéressant d'appeler les méthodes de notre objet Personnage. Vous vous souvenez de `$this` et l'opérateur `->` ? Le principe est exactement le même. La pseudo-variable `$this` représentait l'objet actuellement créé. Et ici, quelle variable représente notre objet ? `$perso`, tout simplement. 😊

On va donc pouvoir utiliser l'opérateur `->` comme on l'a fait avec `$this`. Souvenez-vous : l'opérateur `->` veut dire « dans cet objet, donne-moi cet attribut » ou « dans cet objet, appelle cette méthode ».

Il va falloir créer des accesseurs pour récupérer la force et l'expérience du personnage pour pouvoir les afficher. Rappelez-vous le principe d'encapsulation : tous les attributs doivent être privés. Pour pouvoir quand même accéder à leur valeur, nous avons inventé les accesseurs. Ce sont des méthodes simples qui ne font que renvoyer le contenu de l'attribut. Par convention, chacune de ces méthodes porte le même nom que l'attribut dont elle renvoie la valeur, comme par exemple `experience()`. Libre à vous de les appeler comme vous voulez, c'est une convention parmi tant d'autres. 😊

Le code est donné plus bas.

Code : PHP

```

<?php
$perso1 = new Personnage(); // Un premier personnage.
$perso2 = new Personnage(); // Un second personnage.

$perso1->frapper ($perso2); // $perso1 frappe $perso2.
$perso1->gagnerExperience(); // On gagne de l'expérience.

$perso2->frapper ($perso1); // $perso2 frappe $perso1.
$perso2->gagnerExperience(); // On gagne de l'expérience.

echo 'Le personnage 1 a ', $perso1->force(), ' de force, contrairement au deuxième personnage qui a ' . $perso2->force() . ' de force.';
echo 'Le personnage 1 a ', $perso1->experience(), ' d\'expérience, contrairement au deuxième personnage qui a ' . $perso2->experience() . ' d\'expérience.';

```

Pfff c'est nul, ton code m'affiche les mêmes résultats !

Ah oui. 🤖

Normal, car nos deux personnages ont exactement la même structure et par défaut, la même force et expérience. Il faudrait donc modifier un peu notre code de telle sorte à ce qu'ils n'aient pas la même force et expérience au départ.

Pour modifier les attributs, nous avons le même problème que plus haut pour récupérer leurs valeurs. La solution est la même : il va falloir créer des méthodes qui le font pour nous ! Généralement, ces fonctions commencent par set.

Code : PHP

```
<?php
    $persol = new Personnage(); // Un premier personnage.
    $perso2 = new Personnage(); // Un second personnage.

    $persol->setForce (10);
    $persol->setExperience (2);

    $perso2->setForce (90);
    $perso2->setExperience (58);

    $persol->frapper ($perso2); // $persol frappe $perso2.
    $persol->gagnerExperience(); // On gagne de l'expérience.

    $perso2->frapper ($persol); // $perso2 frappe $persol.
    $perso2->gagnerExperience(); // On gagne de l'expérience.

    echo 'Le personnage 1 a ', $persol->force(), ' de force, contrairement au deuxième pe
    echo 'Le personnage 1 a ', $persol->experience(), ' d\'expérience, contrairement au c
    echo 'Le personnage 1 a ', $persol->degats(), ' de dégâts, contrairement au deuxième
```

Et les nouvelles méthodes :

Code : PHP


```

<?php
class Personnage
{
    private $force;
    private $localisation;
    private $experience;
    private $degats;

    public function deplacer()
    {

    }

    public function frapper(Personnage $persoAFrapper) // On veut un paramètre de type Personnage
    {
        $persoAFrapper->recevoirDegats ($this->force);
    }

    public function gagnerExperience()
    {
        $this->experience++;
    }

    public function recevoirDegats ($force)
    {
        $this->degats += $force;
    }

    public function setForce ($force)
    {
        if (!is_int ($force)) // S'il ne s'agit pas d'un nombre entier
        {
            trigger_error('La force d\'un personnage doit être un nombre entier', E_USER_WARNING);
            return;
        }

        if ($force > 100) // On vérifie bien qu'on ne souhaite pas assigner une valeur supérieure à 100
        {
            trigger_error('La force d\'un personnage ne peut dépasser 100', E_USER_WARNING);
            return;
        }

        $this->force = $force;
    }

    public function setExperience ($experience)
    {
        if (!is_int ($experience)) // S'il ne s'agit pas d'un nombre entier
        {
            trigger_error('L\'expérience d\'un personnage doit être un nombre entier', E_USER_WARNING);
            return;
        }

        if ($experience > 100) // On vérifie bien qu'on ne souhaite pas assigner une valeur supérieure à 100
        {
            trigger_error('L\'expérience d\'un personnage ne peut dépasser 100', E_USER_WARNING);
            return;
        }

        $this->experience = $experience;
    }

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

    public function force()

```

Ce qui affichera :

Citation : Résultat

Le personnage 1 a 10 de force, contrairement au deuxième personnage qui a 90 de force.
Le personnage 1 a 3 d'expérience, contrairement au deuxième personnage qui a 59 d'expérience.
Le personnage 1 a 90 de dégâts, contrairement au deuxième personnage qui a 10 de dégâts.

Le constructeur

Vous vous demandez peut-être à quoi servent les parenthèses juste après `Personnage` lorsque vous créez un objet ? C'est ce que je vais vous expliquer juste après vous avoir dit ce qu'est un constructeur.

Le constructeur est la méthode appelée dès que vous créez l'objet avec la technique montrée ci-dessus. Cette méthode peut demander des paramètres, auquel cas nous devons les placer entre les parenthèses que vous voyez après le nom de la classe.

Faisons un retour sur notre classe `Personnage`. Ajoutons-lui un constructeur. Le constructeur ne peut pas avoir n'importe quel nom (sinon, comment PHP sait quel est le constructeur ?). Le constructeur a tout simplement le nom suivant : `__construct`, avec deux underscores au début.

Comme son nom l'indique, le constructeur sert à construire la classe. Ce que je veux dire par là, c'est que si des attributs doivent être initialisés ou qu'une connexion à la BDD doit être faite, c'est par ici que ça se passe. Comme dit plus haut, le constructeur est exécuté dès la création de l'objet et par conséquent, aucune valeur ne doit être retournée, même si ça ne générera aucune erreur.



Code : PHP

```
<?php
class Personnage
{
    private $force;
    private $localisation;
    private $experience;
    private $degats;

    public function __construct ($force, $degats) // Constructeur demandant 2 paramètres
    {
        echo 'Voici le constructeur !'; // Message s'affichant une fois que tout objet est créé
        $this->force = $force; // Initialisation de la force.
        $this->degats = $degats; // Initialisation des dégâts.
        $this->experience = 1; // Initialisation de l'expérience à 1.
    }

    public function deplacer()
    {

    }

    public function frapper()
    {

    }

    public function gagnerExperience()
    {

    }
}
```

Notez que je n'ai pas réécrit toutes les méthodes, ce n'est pas le but de ce que je veux vous montrer ici. 😊

Ici, le constructeur demande la force et les dégâts initiaux du personnage que l'on vient de créer. Il faudra donc lui spécifier ceci en paramètre.

Code : PHP

```
<?php
    $perso1 = new Personnage (60, 0); // 60 de force, 0 dégât.
    $perso2 = new Personnage (100, 10); // 100 de force, 10 dégâts.
?>
```

Et à la sortie s'affichera à la suite :

Citation : Retour

Voici le constructeur ! Voici le constructeur !

Ne mettez jamais la méthode `__construct` avec le type de visibilité `private` car elle ne pourra jamais être appelée, vous ne pourrez donc pas instancier votre objet ! Cependant, sachez qu'il existe certains cas particuliers qui nécessitent le constructeur en privé, mais ce n'est pas pour tout de suite. 😊

Notez que si la classe n'a pas implémenté de constructeur ou si le constructeur ne requiert aucun argument, alors les parenthèses placées après le nom de la classe lorsque vous l'instancierez sont inutiles. Ainsi, vous pourrez faire `$classe = new MaClasse;`. C'est d'ailleurs sans parenthèses que j'instancierai les classes sans constructeur ou avec constructeur sans argument désormais. 😊

Voici un chapitre aussi essentiel que le premier et toujours aussi riche en nouveautés, fondant les bases de la POO. Prenez bien le temps de lire et relire ce chapitre si vous êtes un peu perdus, sinon vous ne pourrez jamais suivre !

Ce qu'il faut retenir dans ce chapitre, c'est l'utilisation de l'opérateur `->` pour accéder aux attributs et méthodes de l'objet qui précède l'opérateur. Le prochain chapitre portera sur un autre opérateur tout aussi utilisé en POO. Cependant, avant de continuer, prenez bien le temps de comprendre tout ce qui a été dit, sinon vous risquez de confondre ces deux opérateurs.

L'opérateur de résolution de portée ::

Cet opérateur de résolution de portée, appelé « double deux points » (en anglais « Scope Resolution Operator », littéralement « opérateur de résolution de portée »), est utilisé pour appeler des éléments appartenant à telle classe et non à tel objet. En effet, nous pouvons définir des attributs et méthodes appartenant à la classe : ce sont des éléments statiques. Nous y reviendrons en temps voulu dans une partie étant dédiée à ce sujet.

Parmi les éléments appartenant à la classe (et donc appelés via cet opérateur), il y a aussi les constantes de classe. Des constantes de classes sont des sortes d'attributs dont la valeur est constante, c'est-à-dire qu'elle ne change pas. C'est d'ailleurs à propos des constantes de classe que ce chapitre va commencer.

Note : cet opérateur est aussi appelé « Paamayim Nekudotayim ». Enfin bon, je ne vais pas vous demander de le retenir (si vous y arrivez, bien joué 🍪).

Les constantes de classe

Commençons par les constantes de classe. Le principe est à peu près le même que lorsque vous créez une définition à l'aide de la fonction [define](#).

L'utilité des constantes de classe permet d'éviter tout code muet. Voici un code muet :

Code : PHP

```
<?php
    $perso = new Personnage(50);
?>
```

Pourquoi est-il muet ? Tout simplement parce qu'on ne sait pas à quoi « 50 » correspond. Qu'est-ce que cela veut dire ? Étant donné que je viens de faire le script, je sais que ce « 50 » correspond à la force du personnage. Cependant, ce paramètre ne peut prendre que 3 valeurs possibles :

- 20, qui veut dire que le personnage aura une faible force ;
- 50, qui veut dire que le personnage aura une force moyenne ;
- 80, qui veut dire que le personnage sera très fort.

Au lieu de passer ces valeurs telles quelles, on va plutôt passer une constante au constructeur. Ainsi, quand on lira le code, on devinera facilement que l'on passe une force moyenne au constructeur. C'est bien plus facile à comprendre qu'un nombre quelconque.

Une constante est une sorte d'attribut appartenant à la classe dont la valeur ne change jamais. Ceci est peut-être un peu flou, c'est pourquoi on va passer à la pratique. Pour déclarer une constante, vous devez faire précéder son nom du mot-clé `const`. Une constante ne prend pas de \$ devant son nom ! Ainsi, voilà comment créer une constante :

Code : PHP

```
<?php
class Personnage
{
    // TOUS LES ATTRIBUTS EN PRIVÉ SVP !!!

    private $force;
    private $localisation;
    private $experience;
    private $degats;

    // Déclarations des constantes en rapport avec la force.

    const FORCE_PETITE = 20;
    const FORCE_MOYENNE = 50;
    const FORCE_Grande = 80;

    public function __construct ()
    {

    }

    public function deplacer()
    {

    }

    public function frapper()
    {

    }

    public function gagnerExperience()
    {

    }

}
?>
```

Et voilà, facile n'est-ce pas ? 😊

Bien sûr, vous pouvez assigner à ces constantes d'autres valeurs. 😊

Et c'est quoi le rapport avec tes « double deux points » ?

J'y viens. 😊

Contrairement aux attributs, vous ne pouvez accéder à ces valeurs via l'opérateur -> depuis un objet (ni \$this ni \$perso ne fonctionneront) mais avec l'opérateur :: car une constante appartient à la classe, et non à un quelconque objet.

Pour accéder à une constante, vous devez spécifier le nom de la classe, suivi du symbole double deux points, suivi du nom de la constante. Ainsi, on pourrait imaginer un code comme celui-ci :

Code : PHP

```
<?php
class Personnage
{
    private $force;
    private $localisation;
    private $experience;
    private $degats;

    // Déclarations des constantes en rapport avec la force.

    const FORCE_PETITE = 20;
    const FORCE_MOYENNE = 50;
    const FORCE_GRANDE = 80;

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

    public function deplacer()
    {
    }

    public function frapper()
    {
    }

    public function gagnerExperience()
    {
    }
}
?>
```

Et lors de la création de notre personnage :

Code : PHP

```
<?php
$perso = new Personnage (Personnage::FORCE_MOYENNE); // On envoie une « FORCE_MOYENNE
```

Notez que les noms de constantes sont en majuscules : c'est encore et toujours une convention de nommage. 😊

Reconnaissez que ce code est plus lisible que celui montré au début de cette sous-partie. 😊

Les attributs et méthodes statiques

Voici la dernière partie de ce chapitre dans laquelle nous aborderons les attributs et méthodes statiques.

Les méthodes statiques

Comme je l'ai brièvement dit dans l'introduction, les méthodes statiques sont des méthodes qui sont faites pour agir sur la classe et non sur un objet. Par conséquent, je ne veux voir aucun **\$this** dans la méthode ! En effet, la méthode n'étant appelé sur aucun objet, il serait illogique que cette variable existe.

Même si la méthode est dite statique, il est possible de l'appeler depuis un objet (`$obj->methodeStatique()`), mais même dans ce contexte là, la variable `$this` n'existera toujours pas !

Pour déclarer une méthode statique, vous devez faire précéder le mot-clé `function` du mot-clé `static`, après le type de visibilité.

Code : PHP

```
<?php
class Personnage
{
    private $force;
    private $localisation;
    private $experience;
    private $degats;

    const FORCE_PETITE = 20;
    const FORCE_MOYENNE = 50;
    const FORCE_GRADE = 80;

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

    public function deplacer()
    {

    }

    public function frapper()
    {

    }

    public function gagnerExperience()
    {

    }

    // Notez que le mot-clé static peut être placé avant la visibilité de la méthode
    public static function parler()
    {
        echo 'Je vais tous vous tuer !';
    }
}
```

Et dans le code, vous pourrez faire :

Code : PHP

```
<?php
    Personnage::parler();
?>
```

Comme je l'ai dit, vous pouvez aussi appeler la méthode depuis un objet, mais cela ne changera rien au résultat final :

Code : PHP

```
<?php
    $perso = new Personnage (Personnage::FORCE_Grande);
    $perso->parler();
?>
```

Cependant, préférez appeler la méthode avec l'opérateur :: comme le montre le premier de ces deux codes. De cette façon, on voit directement de quelle classe on décide d'invoquer la méthode. De plus, appeler de cette façon une méthode statique lèvera une erreur de degré E_STRICT.

Je me répète mais j'insiste là-dessus : il n'y a pas de variable \$this dans la méthode dans la mesure où la méthode est invoquée afin d'agir sur la classe et non sur un quelconque objet !

Les attributs statiques

Le principe est le même, c'est-à-dire qu'un attribut statique appartient à la classe et non à un objet. Ainsi, tous les objets auront accès à cet attribut, et cet attribut aura la même valeur pour tous les objets.

La déclaration d'un attribut statique se fait en faisant précéder son nom du mot-clé static, comme ceci :

Code : PHP

```

<?php
class Personnage
{
    private $force;
    private $localisation;
    private $experience;
    private $degats;

    const FORCE_PETITE = 20;
    const FORCE_MOYENNE = 50;
    const FORCE_Grande = 80;

    // Variable statique PRIVÉE.
    private static $texteADire = 'Je vais tous vous tuer !';

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

    public function deplacer()
    {
    }

    public function frapper()
    {
    }

    public function gagnerExperience()
    {
    }

    public static function parler()
    {
        echo self::$texteADire; // On donne le texte à dire.
    }
}
?>

```

Quelques nouveautés dans ce code nécessitent des explications. Premièrement, à quoi sert un attribut statique ?

Nous avons vu que les méthodes statiques sont faites pour agir sur la classe. Ok, mais qu'est-ce qu'on peut faire sur une classe ? Et bien tout simplement modifier les attributs de celle-ci, et comme je l'ai déjà dit, des attributs appartenant à une classe ne sont autre que des attributs statiques ! Les attributs statiques servent en particulier à pouvoir avoir des attributs indépendants de tout objet. Ainsi, vous aurez beau créer des tas d'objets, votre attribut aura toujours la même valeur (sauf si l'objet modifie sa valeur, bien sûr). Mieux encore : si l'un des objets modifie sa valeur, tous les autres objets qui accéderont à cet attribut obtiendront la nouvelle valeur ! C'est logique quand on y pense, car un attribut statique appartenant à la classe, il n'existe qu'en un seul exemplaire. Si on le modifie tout le monde pourra accéder à sa nouvelle valeur.

Bon, décortiquons un peu plus la ligne surlignée. La ligne 38 commence avec le mot-clé `self`, ce qui veut dire (en gros) « moi-même » (= la classe). Notre ligne veut donc dire : « Dans moi-même, donne-moi l'attribut statique `$texteADire` » (je sais c'est pas bien français mais c'est la meilleure traduction mot à mot que j'ai pu trouver 🤔).

N'oubliez pas de mettre un `$` devant le nom de l'attribut. C'est souvent source d'erreur, donc faites bien attention. 🤔

On va faire un petit exercice. Je veux que vous me fassiez une classe toute bête qui ne sert à rien. Seulement, je veux, à la fin du script, pouvoir afficher le nombre de fois que la classe a été instanciée. Pour cela, vous aurez besoin d'un attribut appartenant à la classe (admettons `$compteur`) qui est incrémenté dans le constructeur. Voici la correction :

Code : PHP


```

<?php
class Test
{
    // Déclaration de la variable $compteur
    private static $compteur = 0;

    public function __construct()
    {
        // On instancie la variable $compteur qui appartient à la classe (donc utilis
        self::$compteur++;
    }

    public static function getCompteur() // Méthode statique qui renverra la valeur c
    {
        return self::$compteur;
    }
}

$test1 = new Test;
$test2 = new Test;
$test3 = new Test;

echo Test::getCompteur();

```

Eh oui, le retour du mot-clé self... Pourquoi pas \$this ? Souvenez-vous : on n'accède pas à un attribut statique avec \$this mais avec self ! self représente la classe tandis que \$this représente l'objet actuellement créé. Si un attribut statique est modifié, il n'est pas modifié que dans l'objet créé mais dans la structure complète de la classe ! Je me répète, mais il faut bien que vous compreniez ça, c'est ultra important.

Ce qu'il y a à retenir à présent c'est qu'on a vu deux opérateurs au rôle bien différent :

- L'opérateur -> : cet opérateur permet d'accéder à un élément de tel objet ;
- L'opérateur :: : cet opérateur permet d'accéder à un élément de telle classe.

Si vous avez compris le rôle de ces deux opérateurs et surtout quand il faut les utiliser, alors vous avez tout compris. 😊

TP : Mini-jeu de combat

Il est temps de mettre en pratique ce que l'on vient de voir. J'ai donc fait un TP autour du thème du personnage, celui sur lequel on travaille depuis le début. 😊

Nous allons créer un mini (vraiment mini) jeu qui le mettra en scène.

J'utilise ici PDO. Si vous ne savez toujours pas maîtriser cette API, alors allez lire le tutoriel [PDO : Interface d'accès aux BDD](#). Vous avez les connaissances requises pour pouvoir manipuler cette bibliothèque. 😊

Ce qu'on va faire

Cahier des charges

Ce qu'on va réaliser est très simple. On va créer une sorte de jeu. Chaque visiteur pourra créer un personnage (pas de mot de passe requis pour faire simple) avec lequel il pourra frapper d'autres personnages (pas plus de 3 par jour sinon un joueur peut tous les tuer d'un coup 🤖). Le personnage frappé recevra des dégâts en fonction de la force qui l'a frappé. L'utilisateur peut voir ses dégâts se soustraire de 10 s'il se connecte toutes les 24 heures.

Un personnage est défini d'après 6 caractéristiques :

- Son identifiant (juste utile pour son identification en BDD) ;
- son nom (unique en BDD) ;
- sa force ;
- ses dégâts ;
- son niveau (sur 100) ;
- son expérience (sur 100).

Au début, le personnage a un niveau de 1. Plus il frappe de personnages, plus son expérience monte. Une fois arrivée à 100, elle revient à 0 et son niveau monte de 1. S'il a 100 de dégâts, il est mort (supprimé de la BDD). Radicale comme méthode, mais bon, il est mort donc pourquoi le garder ? 🤔

Voici ce qu'on va utiliser pour créer notre classe Personnage :

- Les attributs et méthodes (oui, je préfère le rappeler 🗣️) ;
- l'instanciation de la classe ;
- les constantes de classe ;
- certaines méthodes magiques.

S'organiser

La première question est de savoir combien de classes on va avoir besoin. Déjà, on sait qu'on aura des personnages, donc on a une classe Personnage qui pourra :

- Frapper un autre personnage ;
- Gagner de l'expérience ;
- Perdre et recevoir des dégâts.

Techniquement, un personnage c'est aussi une ligne présente dans notre BDD. Les attributs de cette classe sont donc tout trouvés : ce seront les colonnes de notre BDD.

Bien, on a notre classe Personnage. Mais... qui va s'occuper de faire le lien entre la BDD et le personnage ? Qui va s'occuper de gérer nos personnages ? Eh bien une autre classe, tout simplement. Elle s'appellera PersonnagesManager. Cette classe nous donnera la possibilité :

- D'ajouter un personnage à la BDD ;
- De modifier un personnage de la BDD ;
- De supprimer un personnage de la BDD ;
- De vérifier si tel personnage est présent en BDD (avec un identifiant ou un nom) ;
- De récupérer un personnage ;
- De récupérer la liste des personnages ;
- De compter le nombre de personnages.

Ce sera la page principale, à savoir index.php, qui fera le lien entre les deux classes. Par exemple, quand vous aurez appelé la méthode `frapper()`, celle-ci aura modifié des attributs de deux personnages. Vous devrez alors appeler la méthode `save()` (admettons) de votre manager afin qu'il enregistre les modifications apportées sur les personnages. 😊

Au niveau des fichiers, nous en aurons donc 3 :

- Personnage.class.php qui déclarera la classe Personnage ;
- PersonnagesManager.class.php qui déclarera la classe PersonnagesManager ;
- index.php qui contiendra notre jeu (je vous l'avais dit, il est vraiment petit 🤗).

Voici la structure de la table :

Code : SQL

```
CREATE TABLE IF NOT EXISTS `personnages` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `nom` varchar(50) COLLATE latin1_general_ci NOT NULL,
  `force_perso` tinyint(3) unsigned NOT NULL DEFAULT '5',
  `degats` tinyint(3) unsigned NOT NULL DEFAULT '0',
  `niveau` tinyint(3) unsigned NOT NULL DEFAULT '1',
  `experience` tinyint(3) unsigned NOT NULL DEFAULT '0',
  `nombre_coups` tinyint(3) unsigned NOT NULL DEFAULT '0',
  `time_coups` int(10) unsigned NOT NULL DEFAULT '0', -- stocke le timestamp correspondant
  `time_connexion` int(11) unsigned NOT NULL, -- stocke le timestamp correspondant à la c
PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci;
```

Hydratez vos objets !

Lorsque l'on manipule des objets, et surtout lorsque ceux-ci représentent une ligne de notre BDD (ce qui est le cas ici avec nos personnages), alors il faut mettre en place une méthode qui permet d'hydrater notre objet. Hydrater consiste tout simplement à assigner des valeurs à certains attributs. Pour cela, on passe un tableau associatif (la clé est l'attribut et la valeur est celle de l'attribut). Il faut bien penser à contrôler ces valeurs afin que l'utilisateur ne mette pas n'importe quoi. Cette méthode (admettons `hydrate()`) sera appelée par le constructeur de l'objet si un tableau de valeurs a été passé.

Pré-fabrication

Si vous le souhaitez, je vous laisse le plan des classes. Cependant, essayez de les concevoir vous-mêmes, ce sera bien plus enrichissant. 😊

Secret ([cliquez pour afficher](#))

[Voir le résultat que vous devez obtenir](#)

Bon codage ! 😊

Correction

Voici la correction du TP. Il y avait plusieurs façons de le concevoir, voici la mienne (ma façon n'est pas forcément la meilleure 😊).

Code : PHP - `Personnage.class.php`

```

<?php
/**
 * Classe Personnage utilisée pour le TP du tutoriel sur la POO en PHP, disponible s
 * @author Victor T.
 * @version 2.0
 */
class Personnage
{
    private $degats,
            $experience,
            $force_perso,
            $id,
            $niveau,
            $nom,
            $nombre_coups,
            $time_connexion,
            $time_coups;

    /**
     * Constantes utilisées pour renvoyer une valeur dans une méthode
     * Ceci évite de fouiller dans la classe afin de modifier le message à renvoyer
     */
    const CEST_MOI = 1; // Constante renvoyée par la méthode `frapper` si on se fra
    const PERSONNAGE_TUE = 2; // Constante renvoyée par la méthode `frapper` si on a
    const PERSONNAGE_FRAPPE = 3; // Constante renvoyée par la méthode `frapper` si c
    const QUOTA_DEPASSE = 4; // Constante renvoyée par la méthode `frapper` si on a

    /**
     * Constructeur de la classe Personnage
     * @param $donnees array Les valeurs à assigner au personnage
     * @return void
     */
    public function __construct(array $donnees)
    {
        $this->hydrate($donnees);
    }

    /**
     * Méthode permettant de frapper un personnage
     * @param $perso Personnage Le personnage à frapper
     * @return int
     */
    public function frapper(Personnage $perso)
    {
        if ($perso->id == $this->id)
            return self::CEST_MOI;

        // S'il est à 3 coups
        if ($this->nombre_coups == 3)
        {
            if ($this->time_coups + 24 * 3600 >= time()) // Si son dernier coup rem
                return self::QUOTA_DEPASSE;
            else // Sinon, il faut bien penser à remettre son nombre de coups à 0
                $this->nombre_coups = 0;
        }

        // On met à jour le nombre de coups du personnage ainsi que son timestamp
        $this->nombre_coups++;
        $this->time_coups = time();

        // On indique au personnage qu'il doit recevoir des dégâts, en lui passant
        $action = $perso->recevoirDegats($this->force_perso);
        $this->gagnerExperience(); // On gagne de l'expérience

        return $action; // On retourne la valeur retournée par self::recevoirDegats
    }
}

```



```

<?php
/**
 * Classe PersonnagesManager utilisée pour le TP du tutoriel sur la POO en PHP, dispo
 * @author Victor T.
 * @version 2.0
 */
class PersonnagesManager
{
    private $db; // Instance de PDO

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

    private function add(Personnage $perso)
    {
        $q = $this->db->prepare('INSERT INTO personnages SET nom = :nom, time_conne

        $q->bindValue(':nom', $perso->nom());
        $q->bindValue(':time_connexion', time(), PDO::PARAM_INT);

        $q->execute();

        $perso->hydrate(array(
            'id' => $this->db->lastInsertId(),
            'force_perso' => 5,
            'degats' => 0,
            'niveau' => 1,
            'experience' => 0,
            'nombre_coups' => 0,
            'time_coups' => 0,
            'time_connexion' => time()
        ));
    }

    public function count()
    {
        return $this->db->query('SELECT COUNT(*) FROM personnages')->fetchColumn();
    }

    public function delete(Personnage $perso)
    {
        $this->db->exec('DELETE FROM personnages WHERE id = '.$perso->id());
    }

    public function exists($info)
    {
        if (is_int($info)) // On veut voir si tel personnage ayant pour id $info ex
            return (bool) $this->db->query('SELECT COUNT(*) FROM personnages WHERE :

        // Sinon, c'est qu'on veut vérifier que le nom existe ou pas

        $q = $this->db->prepare('SELECT COUNT(*) FROM personnages WHERE nom = :nom'
        $q->execute(array(':nom' => $info));

        return (bool) $q->fetchColumn();
    }

    public function get($info)
    {
        if (is_int($info))
        {
            return new Personnage(
                $this->db->query('SELECT id, nom, force_perso, degats, niveau, exper
                ->fetch(PDO::FETCH_ASSOC)
            );
        }
    }
}

```

Voilà pour les classes. Si vous avez suivi le plan que j'ai donné précédemment, on devrait avoir à peu près le même code. Si vous vous en êtes sorti, bien joué ! 😊

Voici la page d'index que je vous propose. Celle-ci importe peu, du moment que ça fonctionne bien.

Code : PHP - index.php

```

<?php
// On enregistre notre autoload
function chargerClasse($classname)
{
    require $classname.'.class.php';
}

spl_autoload_register('chargerClasse');

session_start(); // On appelle session_start() APRÈS avoir enregistré l'autoload

if (isset($_GET['deconnexion']))
{
    session_destroy();
    header('Location: .');
    exit();
}

$db = new PDO('mysql:host=localhost;dbname=combats', 'root', '');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING); // On émet une alerte à

$manager = new PersonnagesManager($db);

if (isset($_SESSION['perso'])) // Si la session perso existe, on restaure l'objet
    $perso = $_SESSION['perso'];

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un perso
{
    $perso = new Personnage(array('nom' => $_POST['nom'])); // On crée un nouveau pe

    if (!$perso->nomValide())
    {
        $message = 'Le nom choisi est invalide.';
        unset($perso);
    }
    elseif ($manager->exists($perso->nom()))
    {
        $message = 'Le nom du personnage est déjà pris.';
        unset($perso);
    }
    else
        $manager->save($perso);
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser
{
    if ($manager->exists($_POST['nom'])) // Si celui-ci existe
    {
        $perso = $manager->get($_POST['nom']);

        $perso->perdreDegats();
        $manager->save($perso);
    }
    else
        $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affiche

}

elseif (isset($_GET['frapper'])) // Si on a cliqué sur un personnage pour le frapper
{
    if (!isset($perso))
        $message = 'Merci de créer un personnage ou de vous identifier.';

    else
    {
        if (!$manager->exists((int) $_GET['frapper']))
            $message = 'Le personnage que vous voulez frapper n\'existe pas !';

        else
    }
}

```


Et voilà le travail. 😊

Vous avez aussi pu voir que j'ai commenté ma classe Personnage de façon assez étrange. Il s'agit en fait de commentaires [phpdoc](#) permettant de générer automatiquement la documentation des attributs et méthodes. Sa syntaxe est similaire à la javadoc, détaillée par [ce tutoriel](#).

Et voici la fin de ce TP. Il vous aura fait pratiquer un peu tout en revoyant tout ce qu'on a vu depuis le début. Même si ce TP ne va sans doute pas être directement utile pour votre site, celui-ci utilise une architecture de base pour tout module, même pour un système de news ! Un TP réalisant un tel système est disponible à la fin du tutoriel, mais nous n'en sommes pas encore là. 😊

L'héritage

L'héritage en POO (que ce soit en C++, Java ou autre langage utilisant la POO) est une technique très puissante et extrêmement pratique. Ce chapitre sur l'héritage est le chapitre à connaître par cœur (ou du moins, le mieux possible). Pour être bien sûr que vous ayez compris le principe, un TP vous attend au prochain chapitre. 😊

Allez, j'arrête de vous mettre la pression, on y va !

Notion d'héritage

Définition

Quand on parle d'héritage, c'est qu'on dit qu'une classe B hérite d'une classe A. La classe A est donc considérée comme la classe mère et la classe B est considérée comme la classe fille.

Concrètement, l'héritage, c'est quoi ?

Lorsqu'on dit que la classe B hérite de la classe A, c'est que la classe B hérite de tous les attributs et méthodes de la classe A. Si l'on déclare des méthodes dans la classe A, et qu'on crée une instance de la classe B, alors on pourra appeler n'importe quelle méthode déclarée dans la classe A, du moment qu'elle est publique.

Schématiquement, une classe B héritant d'une classe A peut être représentée comme ceci :

Classe **Personnage**

```
$degats      $force  
$experience  $niveau  
$id          $nom  
  
__construct  frapper  
isNew       hydrate  
gagnerExperience
```



Classe **Magicien** héritant de **Personnage**

```
__construct  frapper  
isNew       hydrate  
gagnerExperience
```

Vous voyez que la classe **Magicien** a hérité de toutes les méthodes et d'aucun attribut de la classe **Personnage**. Souvenez-vous : toutes les méthodes sont publiques et tous les attributs sont privés. En fait, les attributs privés ont bien été hérités aussi, mais notre classe **Magicien** ne pourra s'en servir, c'est la raison pour laquelle je ne les ai pas représentés. Il n'y a que les méthodes de la classe parente qui auront accès à ces attributs. C'est comme pour le principe d'encapsulation : ici, les éléments privés sont masqués. On dit que la classe **Personnage** est la classe mère et que la classe **Magicien** est la classe fille.

Je n'ai pas mis toutes les méthodes du dernier TP dans ce schéma pour ne pas le surcharger, mais en réalité, toutes les méthodes ont bien été héritées. 😊

Quand est-ce que je sais si telle classe doit hériter d'une autre ?

Soit deux classes A et B. Pour qu'un héritage soit possible, il faut que vous puissiez dire que A est un B. Par exemple, un magicien est un personnage, donc héritage. Un chien est un animal, donc héritage aussi. Bref, vous avez compris le principe. 😊

Procéder à un héritage

Pour procéder à un héritage (c'est-à-dire faire en sorte qu'une classe hérite des attributs et méthodes d'une autre classe), il suffit d'utiliser le mot-clé `extends`. Vous déclarez votre classe comme d'habitude (`class MaClasse`) en ajoutant `extends NomDeLaClasseAHeriter` comme ceci :

Code : PHP

```

<?php
class Personnage // Création d'une classe simple.
{

}

class Magicien extends Personnage // Notre classe Magicien hérite des attributs et me
{

}

```

Comme dans la réalité, une mère peut avoir plusieurs filles, mais une fille ne peut avoir plusieurs mères. La seule différence avec la vie réelle, c'est qu'une mère ne peut (en général 🤖) avoir une infinité de filles. 🤖

Ainsi, on pourrait créer des classes Magicien, Guerrier, Brute, etc. qui héritent toutes de Personnage : la classe Personnage sert de modèle.

Code : PHP

```

<?php
class Personnage // Création d'une classe simple.
{

}

// Toutes les classes suivantes hériteront de Personnage.

class Magicien extends Personnage
{

}

class Guerrier extends Personnage
{

}

class Brute extends Personnage
{

}
?>

```

Ainsi, toutes ces nouvelles classes auront les mêmes attributs et méthodes que Personnage. 🤖

Super, tu me crées des classes qui sont exactement les mêmes qu'une autre... Super utile ! En plus, tout ce qui est privé j'y ai pas accès donc...

Si c'était ça l'héritage, ce serait le concept le plus idiot et le plus inutile de la POO. 🤖

Chaque classe peut créer des attributs et méthodes qui lui seront propres, et c'est là toute la puissance de l'héritage : toutes les classes que l'on a créées plus haut peuvent avoir des attributs et méthodes en plus des attributs et méthodes hérités. Pour cela, rien de plus simple. Il vous suffit de créer des attributs et méthodes comme on a appris jusqu'à maintenant. Un exemple ?

Code : PHP

```

<?php
class Magicien extends Personnage
{
    private $magie; // Indique la puissance du magicien sur 100, sa capacité à produi

    public function lancerUnSort($perso)
    {
        $perso->recevoirDegats($this->magie); // On va dire que la magie du magicien
    }
}

```

Ainsi, la classe Magicien aura, en plus des attributs et méthodes hérités, un attribut `$magie` et une méthode `lancerUnSort`.

Si vous essayez d'accéder à un attribut privé de la classe parente, aucune erreur fatale ne s'affichera, juste une notice si vous les avez activées disant que l'attribut n'existe pas.

Surcharger les méthodes

On vient de créer une classe Magicien héritant de toutes les méthodes de la classe Personnage. Que diriez-vous si l'on pouvait récrire certaines méthodes, afin de modifier leur comportement ? Pour cela, il vous suffit de déclarer à nouveau la méthode et d'écrire ce que bon vous semble à l'intérieur.

Un problème se pose pourtant. Si vous voulez accéder à un attribut de la classe parente pour le modifier, vous ne pourrez pas, car notre classe Magicien n'a pas accès aux attributs de sa classe mère Personnage puisqu'ils sont tous privés.

On va maintenant essayer de surcharger la méthode `gagnerExperience` afin de modifier l'attribut stockant la magie (admettons `$magie`) lorsque, justement, on gagne de l'expérience. Problème : si on la récrit, on va écraser toutes les instructions présentes dans la méthode de la classe parente (Personnage), ce qui aura pour effet de ne pas faire gagner d'expérience à notre magicien mais juste de lui augmenter sa magie. Solution : appeler la méthode `gagnerExperience` de la classe parente, puis ensuite ajouter les instructions modifiant la magie.

Il suffit pour cela d'utiliser le mot-clé `parent` suivi du symbole double deux points (le revoilà celui-là 🤖) suivi lui-même du nom de la méthode à appeler.

Code : PHP

```

<?php
class Magicien extends Personnage
{
    private $magie; // Indique la puissance du magicien sur 100, sa capacité à produi

    public function lancerUnSort($perso)
    {
        $perso->recevoirDegats($this->magie); // On va dire que la magie du magicien
    }

    public function gagnerExperience()
    {
        // On appelle la méthode gagnerExperience() de la classe parente
        parent::gagnerExperience();

        if ($this->magie < 100)
        {
            $this->setMagie($this->magie + 10);
        }
    }
}

```

Notez que si la méthode parente retourne une valeur, vous pouvez la récupérer comme si vous appeliez une méthode normalement. Exemple :

Code : PHP

```
<?php
class A
{
    public function test()
    {
        return 'test';
    }
}

class B extends A
{
    public function test()
    {
        $retour = parent::test();

        echo $retour; // Affiche 'test'
    }
}
```

Comme vous pouvez le constater, j'ai fait appel aux getters et setters correspondant à l'attribut **\$magie**. Pourquoi ? Car les classes enfant n'ont pas accès aux éléments privés, il fallait donc que la classe parente le fasse pour moi ! Il n'y a que les méthodes de la classe parente qui ne sont pas réécrites qui ont accès aux éléments privés. À partir du moment où l'on réécrit une méthode de la classe parente, la méthode appartient à la classe fille et n'a donc plus accès aux éléments privés.

Si vous surchargez une méthode, sa visibilité doit être la même que dans la classe parente ! Si tel n'est pas le cas, une erreur fatale sera levée. Par exemple, vous ne pouvez surcharger une méthode publique en disant qu'elle est privée.

Héritages multiples

Toute classe en POO peut être héritée si elle ne spécifie pas le contraire, vraiment toute. Vous pouvez ainsi reproduire un réel arbre avec autant de classes héritant les unes des autres que vous le souhaitez.

Pour reprendre l'exemple du magicien dans le cours sur la POO en C++ de M@teo21, on peut créer deux autres classes MagicienBlanc et MagicienNoir qui héritent toutes les deux de Magicien. Exemple :

Code : PHP

```
<?php
class Personnage // Classe Personnage de base.
{
}

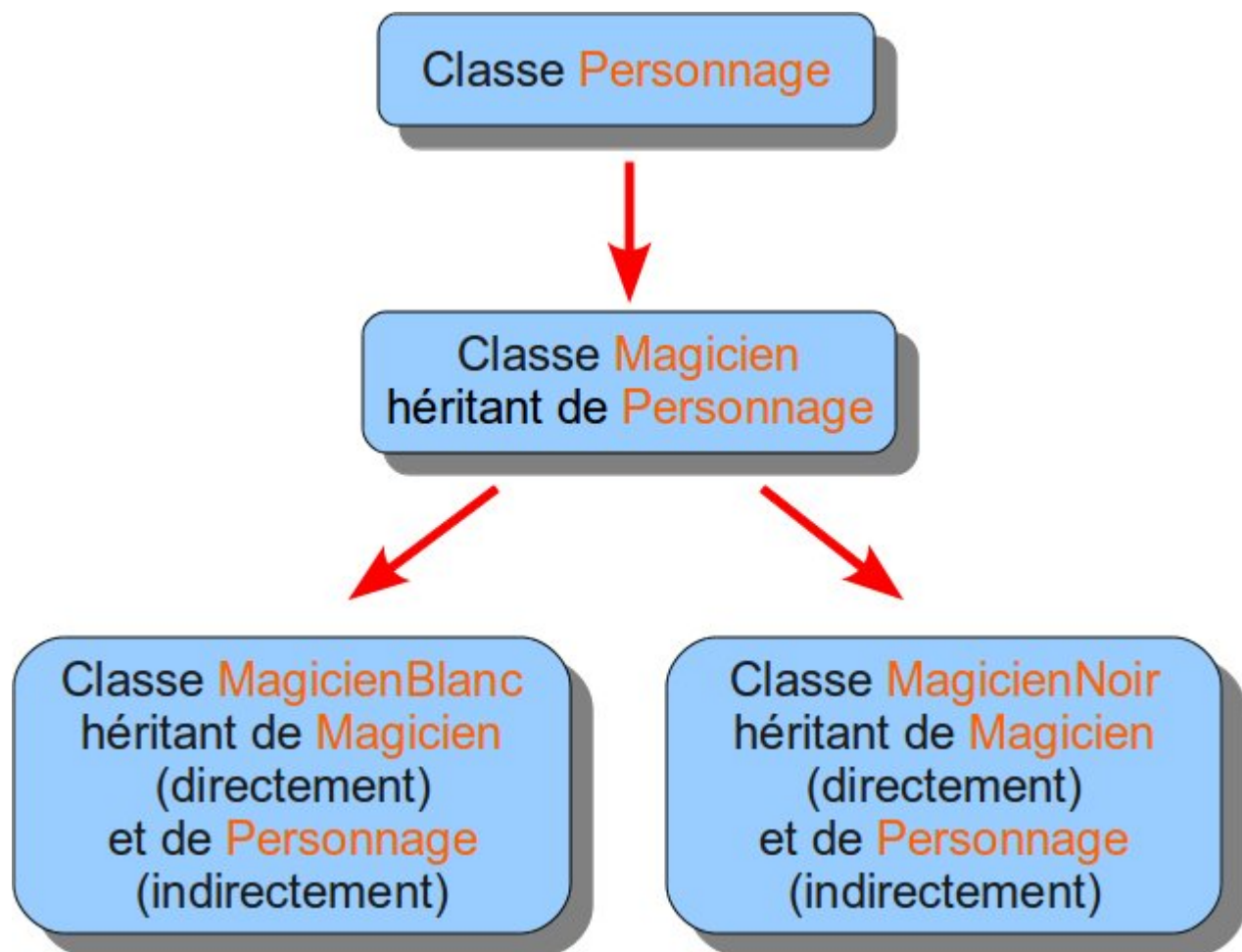
class Magicien extends Personnage // Classe Magicien héritant de Personnage.
{
}

class MagicienBlanc extends Magicien // Classe MagicienBlanc héritant de Magicien.
{
}

class MagicienNoir extends Magicien // Classe MagicienNoir héritant de Magicien.
{
}

?>
```

Et un petit schéma qui reproduit ce code 😊 :



Ainsi, les classes **MagicienBlanc** et **MagicienNoir** hériteront de tous les attributs et de toutes les méthodes des classes **Magicien** et **Personnage**. 😊

Un nouveau type de visibilité : protected

Je vais à présent vous présenter le dernier type de visibilité existant en POO : il s'agit de `protected`. Ce type de visibilité est, au niveau restrictif, à placer entre `public` et `private`.

Je vous rappelle brièvement les rôles de ces deux portées de visibilité :

- `public` : ce type de visibilité nous laisse beaucoup de liberté. On peut accéder à l'attribut ou à la méthode de n'importe où. Toute classe fille aura accès aux éléments publics.
- `private` : ce type est celui qui nous laisse le moins de liberté. On ne peut accéder à l'attribut ou à la méthode que depuis l'intérieur de la classe qui l'a créé. Toute classe fille n'aura pas accès aux éléments privés.

Le type de visibilité `protected` est en fait une petite modification du type `private` : il a exactement les mêmes effets que `private`, à l'exception que toute classe fille aura accès aux éléments protégés.

Exemple :

Code : PHP

```

<?php
class ClasseMere
{
    protected $attributProtege;
    private $attributPrive;

    public function __construct()
    {
        $this->attributProtege = 'Hello world !';
        $this->attributPrive = 'Bonjour tout le monde !';
    }
}

class ClasseFille extends ClasseMere
{
    public function afficherAttributs()
    {
        echo $this->attributProtege; // L'attribut est protégé, on a donc accès à ce
        echo $this->attributPrive; // L'attribut est privé, on n'a pas accès celui-ci
    }
}

$obj = new ClasseFille;

echo $obj->attributProtege; // Erreur fatale.
echo $obj->attributPrive; // Rien ne s'affiche (ou une notice si vous les avez activé

$obj->afficherAttributs(); // Affiche « Hello world ! » suivi de rien du tout ou d'un

```

Et pour le principe d'encapsulation, j'utilise quoi ? `private` ou `protected` ?

La portée `private` est, selon moi, bien trop restrictive et contraignante. Elle empêche toute classe enfant d'accéder aux attributs et méthodes privées alors que cette dernière en a souvent besoin. De manière générale, je vous conseille donc de toujours mettre `protected` au lieu de `private`, à moins que vous teniez absolument à ce que la classe enfant ne puisse y avoir accès. Cependant, je trouve cela inutile dans le sens où la classe enfant a été créée par un développeur, donc quelqu'un qui sait ce qu'il fait et qui par conséquent doit pouvoir modifier à souhait tous les attributs, contrairement à l'utilisateur de la classe.

Imposer des contraintes

Il est possible de mettre en place des contraintes. On parlera alors d'abstraction ou de finalisation suivant la contrainte instaurée.

Abstraction

Classes abstraites

On a vu jusqu'à maintenant que l'on pouvait instancier n'importe quelle classe afin de pouvoir exploiter ses méthodes. On va maintenant découvrir comment empêcher quiconque d'instancier telle classe.

Hein ? Mais à quoi ça sert de créer une classe si on ne peut pas s'en servir ?

On ne pourra pas se servir directement de la classe. La seule façon d'exploiter ses méthodes est de créer une classe héritant de la classe abstraite.

Vous vous demandez sans doute à quoi cela peut bien servir. L'exemple que je vais prendre est celui du personnage et de ses classes filles. Dans ce que nous venons de faire, nous ne créerons jamais d'objet `Personnage`, mais uniquement des objets `Magicien`, `Guerrier`, `Brute`, etc. En effet, à quoi cela nous servirait d'instancier la classe `Personnage` si notre but est de créer un tel type de

personnage ?

On va donc considérer la classe Personnage comme étant une classe modèle dont toute classe fille possèdera les méthodes et attributs.

Pour déclarer une classe abstraite, il suffit de faire précéder le mot-clé `class` du mot-clé `abstract` comme ceci :

Code : PHP

```
<?php
abstract class Personnage // Notre classe Personnage est abstraite.
{

}

class Magicien extends Personnage // Création d'une classe Magicien héritant de la c
{

}

$magicien = new Magicien; // Tout va bien, la classe Magicien n'est pas abstraite.
$perso = new Personnage; // Erreur fatale car on instancie une classe abstraite.
```

Simple et court à retenir, suffit juste de se souvenir où l'on doit le placer. 😊

Ainsi, si vous essayez de créer une instance de la classe Personnage, une erreur fatale sera levée. Ceci nous garantit que l'on ne créera jamais d'objet Personnage (suite à une étourderie par exemple).

Méthodes abstraites

Si vous décidez de rendre une méthode abstraite en plaçant le mot-clé `abstract` juste avant la visibilité de la méthode, vous forcerez toutes les classes filles à écrire cette méthode. Si tel n'est pas le cas, une erreur fatale sera levée. Puisque l'on force la classe fille à écrire la méthode, on ne doit spécifier aucune instruction dans la méthode, on déclarera juste son prototype (visibilité + fonction + nomDeLaMethode + parenthèses avec ou sans paramètres + point-virgule).

Code : PHP

```
<?php
abstract class Personnage
{
    // On va forcer toute classe fille à écrire cette méthode car chaque personnage a
    abstract public function frapper(Personnage $perso);

    // Cette méthode n'aura pas besoin d'être réécrite.
    public function recevoirDegats()
    {
        // Instructions.
    }
}

class Magicien extends Personnage
{
    // On écrit la méthode « frapper » du même type de visibilité que la méthode abs
    public function frapper(Personnage $perso)
    {
        // Instructions.
    }
}
```

Pour définir une méthode comme étant abstraite, il faut que la classe elle-même soit abstraite !

Finalisation

Classes finales

Le concept des classes et méthodes finales est exactement l'inverse du concept d'abstraction. Si une classe est finale, vous ne pourrez pas créer de classe fille héritant de cette classe.

Pour ma part, je ne rends jamais mes classes finales (au même titre que, à quelques exceptions près, je mets toujours mes attributs en `protected`) pour me laisser la liberté d'hériter de n'importe quelle classe.

Pour déclarer une classe finale, vous devez placer le mot-clé `final` juste avant le mot-clé `class`, comme `abstract`.

Code : PHP

```
<?php
    // Classe abstraite servant de modèle.

    abstract class Personnage
    {

    }

    // Classe finale, on ne pourra créer de classe héritant de Guerrier.

    final class Guerrier extends Personnage
    {

    }

    // Erreur fatale, car notre classe hérite d'une classe finale.

    class GentilGuerrier extends Guerrier
    {

    }

?>
```

Méthodes finales

Si vous déclarez une méthode finale, toute classe fille de la classe comportant cette méthode finale héritera de cette méthode mais ne pourra la surcharger. Si vous déclarez votre méthode `recevoirDegats` en tant que méthode finale, vous ne pourrez la surcharger.

Code : PHP

```
<?php
abstract class Personnage
{
    // Méthode normale.

    public function frapper(Personnage $perso)
    {
        // Instructions.
    }

    // Méthode finale.

    final public function recevoirDegats()
    {
        // Instructions.
    }
}

class Guerrier extends Personnage
{
    // Aucun problème.

    public function frapper(Personnage $perso)
    {
        // Instructions.
    }

    // Erreur fatale car cette méthode est finale dans la classe parente.

    public function recevoirDegats()
    {
        // Instructions.
    }
}

?>
```

Résolution statique à la volée

Fonctionnalité disponible depuis PHP 5.3 !

Cette sous-partie 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, donc n'hésitez pas à relire cette partie autant de fois que nécessaire.

On va faire un petit flash-back sur `self::`. Vous vous souvenez à quoi il sert ? À appeler un attribut ou une méthode statique ou une constante de la classe dans laquelle est contenu **`self::`**. Ainsi, si vous testez ce code :

Code : PHP

```

<?php
class Mere
{
    public static function lancerLeTest()
    {
        self::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::lancerLeTest();
?>

```

À l'écran s'affichera :

Citation : Résultat

Je suis la classe Mere !

Mais qu'est-ce qui 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 classe `Mere`.

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 réécrite ?

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 ?

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` !

Code : PHP

```

<?php
class Mere
{
    public static 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::lancerLeTest();
?>

```

donnera
nnera
era :
: Résultat
Résultat

Je suis la classe Enfant !
classe Enfant !

que tous les exemples ci
e tous les exemples ci

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

présenté ci-dessus. J'ai fait ce choix car pour ce genre
PHP
HP

```

<?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> !';
    }
}

$e = new Enfant;
$e->lancerLeTest();
?>

```

Cas complexes

À première vue, vous n'avez peut-être pas tout compris. Si tel est le cas, ne lisez pas la suite ça va encore plus vous embrouiller. Prenez bien le temps de comprendre ce qui est dit plus haut puis vous pourrez continuer. 😊

Comme le spécifie le titre, il y a quelques cas complexes (des pièges en quelque sorte).

Imaginons trois classes A, B et C qui héritent chacune d'une autre (A est la grand-mère, B la mère et C la fille 🤪). En PHP, on dirait plutôt :

Code : PHP

```

<?php
class A
{
}

class B extends A
{
}

class C extends B
{
}

?>

```

On va implémenter dans chacune des classes une méthode qui aura pour rôle d'afficher le nom de la classe pour pouvoir effectuer quelques tests.

Code : PHP

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

class B extends A
{
    public function quiEstCe()
    {
        echo 'B';
    }
}

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

?>
```

On va maintenant créer une méthode de test dans la classe B. Pourquoi dans cette classe ? Parce qu'elle hérite à la fois de A et est héritée par C, son cas est donc intéressant à étudier. 🤔

On va maintenant appeler cette méthode depuis la classe C dans un contexte statique (on n'a pas besoin de créer d'objet, mais ça marche tout aussi bien 😊).

Code : PHP

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

class B extends A
{
    public static function test()
    {

    }

    public function quiEstCe()
    {
        echo 'B';
    }
}

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

C::test();

?>
```

On va donc placer un peu de code dans cette méthode, sinon c'est pas drôle. 😊

On va essayer d'appeler la méthode parente `quiEstCe`. Là, il n'y a pas de piège, pas de résolution statique à la volée, donc à l'écran s'affichera « A » :

Code : PHP

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

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

    public function quiEstCe()
    {
        echo 'B';
    }
}

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

C::test();

?>
```

Maintenant on va créer une méthode dans la classe A qui sera chargée d'appeler la méthode `quiEstCe` avec `static::`. Là, si vous savez ce qui va s'afficher, vous avez tout compris ! 😊

Code : PHP


```

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

    public function quiEstCe()
    {
        echo 'A';
    }
}

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

    public function quiEstCe()
    {
        echo 'B';
    }
}

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

C::test();

?>

```

Alors ? Vous avez une petite idée ? À l'écran s'affichera... C ! Décortiquons ce qui s'est passé :

- 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`.

C'est super compliqué mais important à comprendre. 🤔

Remplaçons maintenant `parent::` par `self::` :

Code : PHP

```

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

    public function quiEstCe()
    {
        echo 'A';
    }
}

class B extends A
{
    public static function test()
    {
        self::appelerQuiEstCe();
    }

    public function quiEstCe()
    {
        echo 'B';
    }
}

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

C::test();
?>

```

Et là, qu'est-ce qui s'affiche à l'écran ? Eh bien toujours C ! Le principe est exactement le même que le code plus haut.

Si vous cassez la chaîne en appelant une méthode depuis une instance ou statiquement du genre `Classe::methode()`, la méthode appelée par `static::` sera celle de la classe contenant ce code ! Ainsi, ce code affichera « A » :

Code : PHP

```

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

    public function quiEstCe()
    {
        echo 'A';
    }
}

class B extends A
{
    public static function test()
    {
        // On appelle « appelerQuiEstCe » de la classe « A » normalement.
        A::appelerQuiEstCe();
    }

    public function quiEstCe()
    {
        echo 'B';
    }
}

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

C::test();

?>

```

Méthodes magiques et fonctions de rappel

Si vous utilisez `static::` à l'intérieur d'une méthode magique ou d'une fonction de rappel, c'est la méthode appelée par `static::` de la classe utilisant la méthode qui sera appelée. Exemple :

Code : PHP

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

    public function __get ($attribut)
    {
        static::quiEstCe();
    }
}

class B extends A
{
    public function quiEstCe()
    {
        echo 'B';
    }
}

$b = new B;
echo $b->attribut;

?>
```

À l'écran s'affichera « B ».

Utilisation de `static::` dans un contexte non statique

L'utilisation de `static::` dans un contexte non statique se fait de la même façon que dans un contexte statique. Je vais prendre l'exemple de la documentation pour illustrer mes propos :

Code : PHP

```

<?php
class TestChild extends TestParent
{
    public function __construct()
    {
        static::qui();
    }

    public function test()
    {
        $o = new TestParent();
    }

    public static function qui()
    {
        echo 'TestChild';
    }
}

class TestParent
{
    public function __construct()
    {
        static::qui();
    }

    public static function qui()
    {
        echo 'TestParent';
    }
}

$o = new TestChild;
$o->test();

?>

```

À l'écran s'affichera « TestChild » suivi de « TestParent ». Je vous explique ce qui s'est passé si vous n'avez pas tout suivi :

- 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.

Ouf ! Enfin terminé ! 😊

N'hésitez pas à le relire autant de fois que nécessaire afin de bien comprendre cette notion d'héritage, et toutes les possibilités que ce concept vous offre. Ne soyez pas pressés de continuer si vous n'avez pas tout compris, sinon vous allez vous planter au TP. 😊

TP : Des personnages spécialisés

Ce TP est en fait une modification du premier (celui qui mettait en scène notre personnage). On va donc ajouter une possibilité supplémentaire au script : le choix du personnage.

Cette modification vous fera voir ce qu'on a vu depuis le dernier TP, à savoir :

- L'héritage ;
- la portée protected ;
- l'abstraction.

Je ne mettrai pas en pratique la résolution statique à la volée car elle ne nous est pas utile ici. Aussi, la finalisation n'est pas utilisée car est plus une contrainte inutile qu'autre chose. 😊

Ce qu'on va faire

Le but à atteindre est simple. Je veux qu'on ait le choix de créer un certain type de personnage qui aura certains avantages. Il ne doit pas être possible de créer un personnage « normal » (donc la classe Personnage devra être abstraite). Comme précédemment, la classe Personnage aura la liste des colonnes de la table en guise d'attributs.

Je vous donne une liste de personnages différents qui pourront être créés. Chaque personnage a un atout différent sous forme d'entier.

- Un magicien : il aura une méthode `lancerUnSort` qui aura pour effet d'endormir un personnage pendant $\$atout / 2$ heures (l'attribut `$atout` représente la dose de magie du personnage). Lancer un sort doit ajouter 1 en BDD pour le nombre de coups du personnage ;
- Une brute : il doit réécrire la méthode `frapper()` afin d'ajouter sa puissance (son atout) à sa force, appeler la méthode parente puis soustraire son atout afin que la force ne soit pas augmentée en BDD ;
- Un guerrier : il doit réécrire la méthode `recevoirDegats()` afin de parer le coup en fonction de sa protection (son atout).

Ceci n'est qu'une petite liste. Libre à vous de créer d'autres personnages. 😊

La structure de la BDD ne sera pas la même. En effet, chaque personnage aura un attribut en plus, et surtout, il faut savoir de quel personnage il s'agit (magicien, brute, guerrier). On va donc créer une colonne `type` et une colonne `atout` (l'attribut qu'il a en plus). Une colonne `time_endormi` devra aussi être créée pour stocker le timestamp auquel le personnage se réveillera s'il a été ensorcelé. Je vous propose donc cette nouvelle structure (j'ai juste ajouté trois nouveaux champs en fin de table) :

Code : SQL

```
CREATE TABLE IF NOT EXISTS `personnages_v2` (  
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,  
  `nom` varchar(50) COLLATE latin1_general_ci NOT NULL,  
  `force_perso` tinyint(3) unsigned NOT NULL DEFAULT '5',  
  `degats` tinyint(3) unsigned NOT NULL DEFAULT '0',  
  `niveau` tinyint(3) unsigned NOT NULL DEFAULT '1',  
  `experience` tinyint(3) unsigned NOT NULL DEFAULT '0',  
  `nombre_coups` tinyint(3) unsigned NOT NULL DEFAULT '0',  
  `time_coups` int(10) unsigned NOT NULL DEFAULT '0',  
  `time_connexion` int(11) unsigned NOT NULL,  
  `time_endormi` int(10) unsigned NOT NULL DEFAULT '0',  
  `type` enum('magicien','brute','guerrier') COLLATE latin1_general_ci NOT NULL,  
  `atout` tinyint(3) unsigned NOT NULL DEFAULT '0',  
  PRIMARY KEY (`id`)  
) ENGINE=MyISAM DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci;
```

[Voir le résultat que vous devez obtenir](#)

Correction

On va maintenant corriger le TP. Je vais ici vous donner mes 5 classes, ainsi que `index.php`. Comme pour le premier TP, il y avait beaucoup de façons différentes pour concevoir le script, et la mienne n'est pas forcément la meilleure. 😊

Code : PHP - `Personnage.class.php`

```

<?php
/**
 * Classe Personnage utilisée pour le TP du tutoriel sur la POO en PHP, disponible s
 * @author Victor T.
 * @version 2.0
 */
abstract class Personnage
{
    protected $atout,
               $degats,
               $experience,
               $force_perso,
               $id,
               $niveau,
               $nom,
               $nombre_coups,
               $time_connexion,
               $time_coups,
               $time_endormi,
               $type;

    /**
     * Constantes utilisées pour renvoyer une valeur dans une méthode
     * Ceci évite de fouiller dans la classe afin de modifier le message à renvoyer
     */
    const CEST_MOI = 1; // Constante renvoyée par la méthode `frapper` si on se fra
    const PERSONNAGE_TUE = 2; // Constante renvoyée par la méthode `frapper` si on a
    const PERSONNAGE_FRAPPE = 3; // Constante renvoyée par la méthode `frapper` si c
    const PERSONNAGE_ENSORCELE = 4; // Constante renvoyée par la méthode `lancerUnSe
    const QUOTA_DEPASSE = 5; // Constante renvoyée par les méthodes `frapper` et `la
    const PAS_DE_MAGIE = 6; // Constante renvoyée par la méthode `lancerUnSort` (vo

    /**
     * Constructeur de la classe Personnage
     * @param $donnees array Les valeurs à assigner au personnage
     * @return void
     */
    public function __construct(array $donnees)
    {
        $this->hydrate($donnees);
    }

    /**
     * Méthode permettant de savoir si le personnage a été endormi par un magicien
     * @return bool
     */
    public function estEndormi()
    {
        return $this->time_endormi > time();
    }

    /**
     * Méthode permettant de frapper un personnage
     * @param $perso int Le personnage à frapper
     * @return int
     */
    public function frapper(Personnage $perso)
    {
        if ($perso->id == $this->id)
            return self::CEST_MOI;

        // S'il est à 3 coups
        if ($this->nombre_coups == 3)
        {
            if ($this->time_coups + 24 * 3600 >= time()) // Si son dernier coup reme
                return self::QUOTA_DEPASSE;
            else // Sinon, il faut bien penser à remettre son nombre de coups à 0

```

```
<?php
/**
 * Classe Brute utilisée pour le TP du tutoriel sur la POO en PHP, disponible sur le
 * @author Victor.T
 * @version 1.0
 * @see Personnage
 */
class Brute extends Personnage
{
    /**
     * Méthode permettant de frapper un personnage
     * @return int
     */
    public function frapper(Personnage $perso)
    {
        // On stocke l'atout actuel dans une variable car celui-ci augmentera peut-être
        $atout = $this->atout;

        // On ajoute l'atout à la force pour frapper plus fort
        $this->force += $atout;

        $retour = parent::frapper($perso);

        // On enlève l'atout qu'on avait ajouté précédemment
        $this->force -= $atout;

        return $retour;
    }

    /**
     * Méthode permettant de gagner de l'expérience
     * @return void
     */
    public function gagnerExperience()
    {
        parent::gagnerExperience();

        // On met à jour la puissance de la brute
        $this->atout = $this->niveau / 2;
    }
}
```



```

<?php
/**
 * Classe Guerrier utilisée pour le TP du tutoriel sur la POO en PHP, disponible sur
 * @author Victor.T
 * @version 1.0
 * @see Personnage
 */
class Guerrier extends Personnage
{
    /**
     * Méthode permettant de gagner de l'expérience
     * @return void
     */
    public function gagnerExperience()
    {
        parent::gagnerExperience();

        // On met à jour la protection du personnage
        $this->atout = $this->force_perso / 2;
    }

    /**
     * Méthode permettant de recevoir des dégâts
     * On la réécrit pour pouvoir diminuer les dégâts encaissés par le guerrier
     * @param $force int La force par laquelle le personnage est frappé
     * @return int
     */
    public function recevoirDegats ($force)
    {
        $parrade = 1 / $this->atout;

        if ($force < 10) // Si la force est inférieure à 10, on augmente de 5 les dégâts
            $this->degats += round (5 * $parrade);

        for ($i = 20; $i <= 100; $i += 10)
        {
            if ($force >= $i - 10 && $force <= $i)
                $this->degats += round (($i / 2) * $parrade);
        }

        // Si on a 100 de dégâts ou plus, on supprime le personnage de la BDD
        if ($this->degats >= 100)
            return self::PERSONNAGE_TUE;

        // Sinon, on se contente de mettre à jour les dégâts du personnage
        return self::PERSONNAGE_FRAPPE;
    }
}

```

Code : PHP - Magicien.class.php

```

<?php
/**
 * Classe Magicien utilisée pour le TP du tutoriel sur la POO en PHP, disponible sur
 * @author Victor.T
 * @version 1.0
 * @see Personnage
 */
class Magicien extends Personnage
{
    /**
     * Méthode permettant de gagner de l'expérience
     * @return void
     */
    public function gagnerExperience()
    {
        parent::gagnerExperience();

        // On met à jour la magie du magicien
        $this->atout = $this->niveau / 2;
    }

    /**
     * Méthode permettant de lancer un sort à un personnage
     * Ceci aura pour effet d'endormir le personnage durant un certain temps (suivant
     * @param $id Personnage Le personnage à ensorceler
     * @return int
     */
    public function lancerUnSort(Personnage $perso)
    {
        if ($perso->id == $this->id)
            return self::CEST_MOI;

        if ($this->atout == 0)
            return self::PAS_DE_MAGIE;

        // Si on est à 3 coups
        if ($this->nombre_coups == 3)
        {
            if ($this->time_coups + 24 * 3600 >= time()) // Si son dernier coup remon
                return self::QUOTA_DEPASSE;
            else // Sinon, il faut bien penser à remettre son nombre de coups à 0
                $this->nombre_coups = 0;
        }

        $this->nombre_coups++;
        $this->time_coups = time();

        $perso->time_endormi = time() + ($this->atout / 2) * 3600;

        return self::PERSONNAGE_ENSORCELE;
    }
}

```

Passons maintenant au manager. Nous allons toujours garder un seul manager, car on gère toujours des personnages ayant la même structure. Voici la nouvelle classe :

Code : PHP - PersonnagesManager.class.php

```

<?php
/**
 * Classe PersonnagesManager utilisée pour le TP du tutoriel sur la POO en PHP, dispo
 * @author Victor T.
 * @version 2.0
 */
class PersonnagesManager
{
    private $db; // Instance de PDO

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

    protected function add(Personnage $perso)
    {
        $q = $this->db->prepare('INSERT INTO personnages_v2 SET nom = :nom, time_con

        $q->bindValue(':nom', $perso->nom());
        $q->bindValue(':time_connexion', time(), PDO::PARAM_INT);
        $q->bindValue(':type', $perso->type());

        $q->execute();

        $perso->hydrate(array(
            'id' => $this->db->lastInsertId(),
            'force_perso' => 5,
            'degats' => 0,
            'niveau' => 1,
            'experience' => 0,
            'nombre_coups' => 0,
            'time_coups' => 0,
            'time_connexion' => time()
        ));
    }

    public function count()
    {
        return $this->db->query('SELECT COUNT(*) FROM personnages_v2')->fetchColumn();
    }

    public function delete(Personnage $perso)
    {
        $this->db->exec('DELETE FROM personnages_v2 WHERE id = '.$perso->id());
    }

    public function exists($info)
    {
        if (is_int($info)) // On veut voir si tel personnage ayant pour id $info exi
            return (bool) $this->db->query('SELECT COUNT(*) FROM personnages_v2 WHE

        // Sinon, c'est qu'on veut vérifier que le nom existe ou pas

        $q = $this->db->prepare('SELECT COUNT(*) FROM personnages_v2 WHERE nom = :no
        $q->execute(array(':nom' => $info));

        return (bool) $q->fetchColumn();
    }

    public function get($info)
    {
        if (is_int($info))
        {
            $perso = $this->db->query('SELECT id, nom, force_perso, degats, niveau,
                                ->fetch(PDO::FETCH_ASSOC);
        }
    }
}

```

Enfin, finissons par la page d'index qui a légèrement changé :

Code : PHP - index.php

```

<?php
function chargerClasse($classe)
{
    require $classe . '.class.php';
}

spl_autoload_register('chargerClasse');

session_start();

if (isset($_GET['deconnexion']))
{
    session_destroy();
    header('Location: .');
    exit();
}

$db = new PDO('mysql:host=localhost;dbname=combats', 'root', '');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);

$manager = new PersonnagesManager($db);

if (isset($_SESSION['perso'])) // Si la session perso existe, on restaure l'objet
    $perso = $_SESSION['perso'];

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un perso
{
    switch ($_POST['type'])
    {
        case 'brute' :
            $perso = new Brute(array('nom' => $_POST['nom'], 'type' => 'brute'));
            break;

        case 'magicien' :
            $perso = new Magicien(array('nom' => $_POST['nom'], 'type' => 'magicien'));
            break;

        case 'guerrier' :
            $perso = new Guerrier(array('nom' => $_POST['nom'], 'type' => 'guerrier'));
            break;

        default :
            $message = 'Le type du personnage est invalide.';
            break;
    }

    if (isset($perso)) // Si le type du personnage est valide, on a créé un personnage
    {
        if (!$perso->nomValide())
        {
            $message = 'Le nom choisi est invalide.';
            unset($perso);
        }
        elseif ($manager->exists($perso->nom()))
        {
            $message = 'Le nom du personnage est déjà pris.';
            unset($perso);
        }
        else
            $manager->save($perso);
    }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser
{
    if ($manager->exists($_POST['nom'])) // Si celui-ci existe
    {
        $perso = $manager->get($_POST['nom']);
    }
}

```

Alors, vous commencez à comprendre toute la puissance de la POO et de l'héritage ? Avec une telle structure, vous pourrez à tout moment décider de créer un nouveau personnage très simplement ! Il vous suffit de créer une nouvelle classe, d'ajouter le type du personnage à l'énumération « type » en BDD et de modifier un petit peu le fichier index.php et votre personnage voit le jour ! 😊

Voici la fin de ce TP. J'espère que vous y êtes parvenus sans trop de mal. 😊

Les méthodes magiques

On va terminer cette partie par un chapitre assez simple. Dans ce chapitre, on va se pencher sur une possibilité que nous offre le langage : il s'agit des méthodes magiques. Ce sont des petites bricoles bien pratiques dans certains cas. 😊

Le principe

Vous devez sans doute vous poser une grosse question en voyant le titre du chapitre : mais qu'est-ce que c'est qu'une méthode magique ? 😊

Une méthode magique est une méthode qui, si elle est présente dans votre classe, sera appelée lors de tel ou tel événement. Si la méthode n'existe pas et que l'événement est exécuté, aucun effet « spécial » ne sera ajouté, l'événement s'exécutera normalement. Le but des méthodes magiques est d'intercepter un événement, dire de faire ça ou ça et retourner une valeur utile pour l'événement si besoin il y a.

Bonne nouvelle : vous connaissez déjà une méthode magique ! 😊

Si si, cherchez bien au fond de votre tête... Eh oui, la méthode `__construct` est magique ! Comme dit plus haut, chaque méthode magique s'exécute au moment où tel événement est lancé. L'événement qui appelle la méthode `__construct` est la création de l'objet.

Dans le même genre que `__construct` on peut citer `__destruct` qui, elle, sera appelée lors de la destruction de l'objet. Assez intuitif, mais voici un exemple au cas où :

Code : PHP

```
<?php
class MaClasse
{
    public function __construct()
    {
        echo 'Construction de MaClasse';
    }

    public function __destruct()
    {
        echo 'Destruction de MaClasse';
    }
}

$obj = new MaClasse;

?>
```

Ainsi, vous verrez les deux messages écrits ci-dessus à la suite.

Surcharger les attributs et méthodes

Parlons maintenant des méthodes magiques liées à la surcharge des attributs et méthodes.

Euh, deux secondes là... c'est quoi la « surcharge des attributs et méthodes » ??

Ah, oui, il serait préférable d'expliquer ceci. La surcharge d'attributs ou méthodes consiste à prévoir le cas où l'on appelle un attribut ou méthode qui n'existe pas ou du moins, auquel on n'a pas accès (par exemple, si un attribut ou une méthode est privé(e)). Dans ce cas-là, on a... voyons... 6 méthodes magiques à notre disposition ! 😊

« __set » et « __get »

Commençons par étudier ces deux méthodes magiques. Leur principe est le même, leur fonctionnement est à peu près semblable, c'est juste l'événement qui change.

Commençons par __set. Cette méthode est appelée lorsqu'on essaye d'assigner une valeur à un attribut auquel on n'a pas accès ou qui n'existe pas. Cette méthode prend deux paramètres : le premier est le nom de l'attribut auquel on a tenté d'assigner une valeur, le second paramètre est la valeur que l'on a tenté d'assigner à l'attribut. Cette méthode ne retourne rien. Vous pouvez juste faire ce que bon vous semble. 😊

Exemple :

Code : PHP

```
<?php
class MaClasse
{
    private $unAttributPrive;

    public function __set ($nom, $valeur)
    {
        echo 'Ah, on a tenté d\'assigner à l\'attribut <strong>', $nom, '</strong> la valeur ' . $valeur . ' mais c\'est pas possible !';
    }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';
```

À la sortie s'affichera :

Citation : Résultat

Ah, on a tenté d'assigner à l'attribut attribut la valeur Simple test mais c'est pas possible !
Ah, on a tenté d'assigner à l'attribut unAttributPrive la valeur Autre simple test mais c'est pas possible !

Tenez, petit exercice, stockez dans un tableau tous les attributs (avec leurs valeurs) qu'on a essayé de modifier ou créer. 😊

Solution :

Code : PHP

```

<?php
class MaClasse
{
    private $attributs = array();
    private $unAttributPrive;

    public function __set ($nom, $valeur)
    {
        $this->attributs[$nom] = $valeur;
    }

    public function afficherAttributs()
    {
        echo '<pre>', print_r ($this->attributs, true), '</pre>';
    }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';

$obj->afficherAttributs();

?>

```

Pas compliqué, mais ça fait pratiquer un peu. 😊

Parlons maintenant de `__get`. Cette méthode est appelée lorsqu'on essaye d'accéder à un attribut qui n'existe pas ou auquel on n'a pas accès. Elle prend un paramètre : le nom de l'attribut auquel on a essayé d'accéder. Cette méthode peut retourner ce qu'elle veut (ce sera, en quelque sorte, la valeur de l'attribut impossible à accéder).

Exemple :

Code : PHP

```

<?php
class MaClasse
{
    private $unAttributPrive;

    public function __get ($nom)
    {
        return 'Impossible d\'accéder à l\'attribut <strong>' . $nom . '</strong>, désolé !';
    }
}

$obj = new MaClasse;

echo $obj->attribut;
echo $obj->unAttributPrive;

?>

```

Ce qui va afficher :

Citation : Résultat

```

Impossible d'accéder à l'attribut attribut, désolé !
Impossible d'accéder à l'attribut unAttributPrive, désolé !

```

Encore un exercice. 😊

Combinez l'exercice précédent en vérifiant si l'attribut auquel on a tenté d'accéder est contenu dans le tableau de stockage d'attributs. Si tel est le cas, on l'affiche, sinon, on ne fait rien. 😊

Solution :

Code : PHP

```
<?php
class MaClasse
{
    private $attributs = array();
    private $unAttributPrive;

    public function __get ($nom)
    {
        if (isset ($this->attributs[$nom]))
            return $this->attributs[$nom];
    }

    public function __set ($nom, $valeur)
    {
        $this->attributs[$nom] = $valeur;
    }

    public function afficherAttributs()
    {
        echo '<pre>', print_r ($this->attributs, true), '</pre>';
    }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';

echo $obj->attribut;
echo $obj->autreAttribut;

?>
```

Étant donné que tous vos attributs doivent être privés, vous pouvez facilement les mettre en « lecture seule » grâce à `__get`. L'utilisateur aura accès aux attributs, mais ne pourra pas les modifier.

« `__isset` » et « `__unset` »

La première méthode `__isset` est appelée lorsque l'on appelle la fonction `isset` sur un attribut qui n'existe pas ou auquel on n'a pas accès. Étant donné que la fonction initiale `isset` renvoie `true` ou `false`, la méthode magique `__isset` doit renvoyer un booléen. Cette méthode prend un paramètre : le nom de l'attribut que l'on a envoyé à la fonction `isset`. Vous pouvez par exemple utiliser la classe précédente en implémentant la méthode `__isset`, ce qui peut nous donner :

Code : PHP

```

<?php
class MaClasse
{
    private $attributs = array();
    private $unAttributPrive;

    public function __set ($nom, $valeur)
    {
        $this->attributs[$nom] = $valeur;
    }

    public function __get ($nom)
    {
        if (isset ($this->attributs[$nom]))
            return $this->attributs[$nom];
    }

    public function __isset ($nom)
    {
        return isset ($this->attributs[$nom]);
    }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';

if (isset ($obj->attribut))
    echo 'L\'attribut <strong>attribut</strong> existe !<br />';
else
    echo 'L\'attribut <strong>attribut</strong> n\'existe pas !<br />';

if (isset ($obj->unAutreAttribut))
    echo 'L\'attribut <strong>unAutreAttribut</strong> existe !';
else
    echo 'L\'attribut <strong>unAutreAttribut</strong> n\'existe pas !';
?>

```

Ce qui affichera :

Citation : Résultat

L'attribut attribut existe !
 L'attribut unAutreAttribut n'existe pas !

Pour `__unset`, le principe est le même. Cette méthode est appelée lorsque l'on tente d'appeler la fonction `unset` sur un attribut inexistant ou auquel on n'a pas accès. On peut facilement implémenter `__unset` à la classe précédente de manière à supprimer l'entrée correspondante dans notre tableau `$attributs`. Cette méthode ne doit rien retourner.

Code : PHP

```

<?php
class MaClasse
{
    private $attributs = array();
    private $unAttributPrive;

    public function __set ($nom, $valeur)
    {
        $this->attributs[$nom] = $valeur;
    }

    public function __get ($nom)
    {
        if (isset ($this->attributs[$nom]))
            return $this->attributs[$nom];
    }

    public function __isset ($nom)
    {
        return isset ($this->attributs[$nom]);
    }

    public function __unset ($nom)
    {
        if (isset ($this->attributs[$nom]))
            unset ($this->attributs[$nom]);
    }
}

$obj = new MaClasse;

$obj->attribut = 'Simple test';
$obj->unAttributPrive = 'Autre simple test';

if (isset ($obj->attribut))
    echo 'L\'attribut <strong>attribut</strong> existe !<br />';
else
    echo 'L\'attribut <strong>attribut</strong> n\'existe pas !<br />';

unset ($obj->attribut);

if (isset ($obj->attribut))
    echo 'L\'attribut <strong>attribut</strong> existe !<br />';
else
    echo 'L\'attribut <strong>attribut</strong> n\'existe pas !<br />';

if (isset ($obj->unAutreAttribut))
    echo 'L\'attribut <strong>unAutreAttribut</strong> existe !';
else
    echo 'L\'attribut <strong>unAutreAttribut</strong> n\'existe pas !';
?>

```

Ce qui donnera :

Citation : Résultat

L'attribut attribut existe !
 L'attribut attribut n'existe pas !
 L'attribut unAutreAttribut n'existe pas !

Finissons par « __call » et « __callStatic »

Bien que la méthode magique __call soit disponible sous PHP 5.1, la méthode __callStatic n'est disponible que sous PHP 5.3 !

Bien. On va laisser de côté les attributs pour le moment et parler cette fois-ci des méthodes que l'on appelle alors qu'on n'y a pas accès (soit elle n'existe pas, soit elle est privée). La méthode `__call` sera appelée lorsque l'on essayera d'appeler une telle méthode. Elle prend deux arguments : le premier est le nom de la méthode qu'on a essayé d'appeler et le second est la liste des arguments qui lui ont été passés (sous forme de tableau).

Exemple :

Code : PHP

```
<?php
class MaClasse
{
    public function __call ($nom, $arguments)
    {
        echo 'La méthode <strong>', $nom, '</strong> a été appelée alors qu\'elle n\'
    }
}

$obj = new MaClasse;

$obj->methode (123, 'test');
```

Résultat :

Citation : Résultat

La méthode methode a été appelée alors qu'elle n'existe pas ! Ses arguments étaient les suivants : 123, test

Et si on essaye d'appeler une méthode qui n'existe pas statiquement ? Eh bien, erreur fatale ! Sauf si vous utilisez `__callStatic`. Cette méthode est appelée lorsque vous appelez une méthode dans un contexte statique alors qu'elle n'existe pas. La méthode magique `__callStatic` doit obligatoirement être `static` !

Code : PHP

```
<?php
class MaClasse
{
    public function __call ($nom, $arguments)
    {
        echo 'La méthode <strong>', $nom, '</strong> a été appelée alors qu\'elle n\'
    }

    public static function __callStatic ($nom, $arguments)
    {
        echo 'La méthode <strong>', $nom, '</strong> a été appelée dans un contexte s
    }
}

$obj = new MaClasse;

$obj->methode (123, 'test');

MaClasse::methodeStatique (456, 'autre test');
```

Résultat :

Citation : Résultat

La méthode `methode` a été appelée alors qu'elle n'existe pas ! Ses arguments étaient les suivants : 123, test.
La méthode `methodeStatique` a été appelée dans un contexte statique alors qu'elle n'existe pas ! Ses arguments étaient les suivants : 426, autre test.

Linéariser ses objets

Voici un point important de ce chapitre sur lequel je voudrais m'arrêter un petit instant : la linéarisation des objets. Pour suivre cette partie, je vous recommande chaudement [ce tutoriel](#) qui vous expliquera de manière générale ce qu'est la linéarisation et vous fera pratiquer sur des exemples divers. Une fois que vous arrivez à la partie Encore plus fort !, arrêtez-vous et revenez ici. Je vais mieux vous expliquer (car ce tutoriel est spécialisé en POO contrairement à celui traitant de la linéarisation).

Posons le problème

Vous avez un système de sessions sur votre site avec une classe `Connexion`. Cette classe, comme son nom l'indique, aura pour rôle d'établir une connexion à la BDD. Vous aimeriez bien stocker l'objet créé dans une variable `$_SESSION` mais vous ne savez pas comment faire.

Ben si ! On fait `$_SESSION['connexion'] = $objetConnexion` et puis voilà !

Oui, ça fonctionne, mais savez-vous vraiment ce qui se passe quand vous effectuez une telle opération ? Ou plutôt, ce qui se passe à la fin du script ? En fait, à la fin du script, le tableau de session est linéarisé automatiquement. Linéariser signifie que l'on transforme une variable en chaîne de caractères selon un format bien précis. Cette chaîne de caractères pourra, quand on le souhaitera, être transformée dans l'autre sens (c'est-à-dire qu'on va restituer son état d'origine). Pour bien comprendre ce principe, on va linéariser nous-mêmes notre objet. Voici ce que nous allons faire :

- Création de l'objet (`$objetConnexion = new Connexion();`);
- transformation de l'objet en chaîne de caractères (`$_SESSION['connexion'] = serialize ($objetConnexion);`);
- changement de page ;
- transformation de la chaîne de caractères en objet (`$objetConnexion = unserialize ($_SESSION['connexion']);`).

Des explications s'imposent. 🤔

Les nouveautés rencontrées ici sont l'apparition de deux nouvelles fonctions : [serialize](#) et [unserialize](#).

La première fonction, `serialize`, retourne l'objet passé en paramètre sous forme de chaîne de caractères. Vous vous demandez sans doute comment on peut transformer un objet en chaîne de caractères : la réponse est toute simple. Quand on y réfléchit, un objet c'est quoi ? C'est un ensemble d'attributs, tout simplement. Les méthodes ne sont pas stockées dans l'objet, c'est la classe qui s'en occupe. Notre chaîne de caractères contiendra donc juste quelque chose comme : « Objet MaClasse contenant les attributs `unAttribut` qui vaut "Hello world !", `autreAttribut` qui vaut "Vive la linéarisation", `dernierAttribut` qui vaut "Et un dernier pour la route !" ». Ainsi, vous pourrez conserver votre objet dans une variable sous forme de chaîne de caractères. Si vous affichez cette chaîne par un echo par exemple, vous n'arriverez sans doute pas à déchiffrer l'objet, c'est normal, ce n'est pas aussi simple que la chaîne que j'ai montrée à titre d'exemple 😊. Cette fonction est automatiquement appelée sur l'array `$_SESSION` à la fin du script, notre objet est donc automatiquement linéarisé à la fin du script. C'est uniquement dans un but didactique que nous linéarisons manuellement. 🤔

La seconde fonction, `unserialize`, retourne la chaîne de caractères passée en paramètre sous forme d'objet. En gros, cette fonction lit la chaîne de caractères, crée une instance de la classe correspondante et assigne à chaque attribut la valeur qu'ils avaient. Ainsi, vous pourrez utiliser l'objet retourné (appel de méthodes, attributs et diverses opérations) comme avant. Cette fonction est automatiquement appelée dès le début du script pour restaurer le tableau de sessions précédemment enregistré dans le fichier. Sachez toutefois que si vous avez linéarisé un objet manuellement, il ne sera jamais restauré automatiquement.

Et c'est quoi le rapport avec tes méthodes magiques ?

En fait, les fonctions citées ci-dessus (`serialize` et `unserialize`) ne se contentent pas de transformer le paramètre qu'on leur

passer en autre chose : elles vérifient si, dans l'objet passé en paramètre (pour `serialize`), il y a une méthode `__sleep`, auquel cas elle est exécutée. Si c'est `unserialize` qui est appelée, la fonction vérifie si l'objet obtenu comporte une méthode `__wakeup`, auquel cas elle est appelée.

« `serialize` » et « `__sleep` »

La méthode magique `__sleep` est utilisée pour nettoyer l'objet ou pour sauver des attributs. Si la méthode magique `__sleep` n'existe pas, tous les attributs seront sauvés. Cette méthode doit renvoyer un tableau avec les noms des attributs à sauver. Par exemple, si vous voulez sauver `$serveur` et `$login`, la fonction devra retourner `array ('serveur', 'login')`.

Voici ce que pourrait donner notre classe `Connexion` :

Code : PHP

```
<?php
class Connexion
{
    private $serveur, $utilisateur, $motDePasse, $dataBase;

    public function __construct ($serveur, $utilisateur, $motDePasse, $dataBase)
    {
        $this->serveur = $serveur;
        $this->utilisateur = $utilisateur;
        $this->motDePasse = $motDePasse;
        $this->dataBase = $dataBase;

        $this->connexionBDD();
    }

    private function connexionBDD()
    {
        mysql_connect ($this->serveur, $this->utilisateur, $this->motDePasse);
        mysql_select_db ($this->dataBase);
    }

    public function __sleep()
    {
        mysql_close();
        return array ( 'serveur', 'utilisateur', 'motDePasse', 'dataBase' );
    }
}
?>
```

Ainsi, vous pourrez faire ceci :

Code : PHP

```
<?php
$connexion = new Connexion ( 'localhost', 'root', '', 'tests' );

$_SESSION[ 'connexion' ] = serialize ( $connexion );
?>
```

« `unserialize` » et « `__wakeup` »

Maintenant nous allons simplement implémenter la fonction `__wakeup`. Qu'est-ce qu'on va mettre dedans ?

Ben rien de compliqué... On va juste appeler la méthode `connexionBDD` qui se chargera de nous connecter à notre base de données puisque les identifiants, serveur et nom de la base ont été sauvegardés et ainsi restaurés à l'appel de la fonction `unserialize` !

Code : PHP

```

<?php
class Connexion
{
    private $serveur, $utilisateur, $motDePasse, $dataBase;

    public function __construct($serveur, $utilisateur, $motDePasse, $dataBase)
    {
        $this->serveur = $serveur;
        $this->utilisateur = $utilisateur;
        $this->motDePasse = $motDePasse;
        $this->dataBase = $dataBase;

        $this->connexionBDD();
    }

    private function connexionBDD()
    {
        mysql_connect ($this->serveur, $this->utilisateur, $this->motDePasse);
        mysql_select_db ($this->dataBase);
    }

    public function __sleep()
    {
        mysql_close();
        return array ('serveur', 'utilisateur', 'motDePasse', 'dataBase');
    }

    public function __wakeup()
    {
        $this->connexionBDD();
    }
}
?>

```

Pratique, hein ? 😊

Maintenant que vous savez ce qui se passe quand vous enregistrez un objet dans une entrée de session, je vous autorise à ne plus appeler serialize et unserialize. 😊

Ainsi, ce code fonctionne parfaitement :

Code : PHP

```

<?php
session_start();


if (!isset ($_SESSION['connexion']))
{
    $connexion = new Connexion ('localhost', 'root', '', 'tests');
    $_SESSION['connexion'] = $connexion;

    echo 'Actualisez la page !';
}

else
{
    echo '<pre>';
    var_dump ($_SESSION['connexion']); // On affiche les infos concernant notre objet
    echo '</pre>';
}
?>

```

Vous voyez donc, en testant ce code, que notre objet a bel et bien été sauvegardé comme il fallait, et que tous les attributs ont été

sauvés. Bref, c'est magique. 

Étant donné que notre objet est restauré automatiquement lors de l'appel de `session_start()`, la classe correspondante doit être déclarée avant, sinon le script plantera. Si vous avez un autoload qui chargera la classe automatiquement, il sera appelé.

Autres méthodes magiques

Voici les dernières méthodes magiques que vous n'avez pas vues. Je parlerai ici de `__toString`, `__set_state` et `__invoke`.

« `__toString` »

La méthode magique `__toString` est appelée lorsque l'objet est amené à être converti en chaîne de caractères. Cette méthode doit retourner la chaîne de caractères souhaitée.

Exemple :

Code : PHP

```
<?php
class MaClasse
{
    private $texte;

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


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

$objj = new MaClasse ('Hello world !');

// Solution 1 : le cast
$texte = (string) $objj;
var_dump ($texte); // Affiche : string(13) "Hello world !".

// Solution 2 : directement dans un echo
echo $objj; // Affiche : Hello world !

?>
```

Pas mal, hein ? 

« `__set_state` »

La méthode magique `__set_state` est appelée lorsque vous appelez la fonction [var_export](#) en passant votre objet à exporter en paramètre. Cette fonction `var_export` a pour rôle d'exporter la variable passée en paramètre sous forme de code PHP (chaîne de caractères). Si vous ne spécifiez pas de méthode `__set_state` dans votre classe, une erreur fatale sera levée.

Notre méthode `__set_state` prend un paramètre, la liste des attributs ainsi que leur valeur dans un tableau associatif (`array ('attribut' => 'valeur')`). Notre méthode magique devra retourner l'objet à exporter. Il faudra donc créer un nouvel objet et lui assigner les valeurs qu'on souhaite, puis le retourner.

Ne jamais retourner `$this`, car cette variable n'existera pas dans cette méthode ! `var_export` reportera donc une valeur nulle.

Puisque la fonction `var_export` retourne du code PHP valide, on peut utiliser la fonction [eval](#) qui exécute du code PHP sous forme de chaîne de caractères qu'on lui passe en paramètre.

Par exemple, pour retourner un objet en sauvant ses attributs, on pourrait faire :

Code : PHP

```
<?php
class Export
{
    private $chaine1, $chaine2;

    public function __construct ($param1, $param2)
    {
        $this->chaine1 = $param1;
        $this->chaine2 = $param2;
    }

    public function __set_state ($valeurs) // Liste des attributs de l'objet en paramètre
    {
        $obj = new Export ($valeurs['chaine1'], $valeurs['chaine2']); // On crée un objet
        return $obj; // on retourne l'objet créé
    }
}

$obj1 = new Export('Hello ', 'world !');

eval ('$obj2 = ' . var_export ($obj1, true) . ';'); // On crée un autre objet, celui-ci

echo '<pre>', print_r ($obj2, true), '</pre>';
```

Le code affichera donc :

Citation : Résultat

```
Export Object
(
    [chaine1:private] => Hello
    [chaine2:private] => world !
)
```

« __invoke »

Disponible depuis PHP 5.3

Que diriez-vous de pouvoir utiliser l'objet comme fonction ? Vous ne voyez pas ce que je veux dire ? Je comprends. 🤔

Voici un code qui illustrera bien le tout :

Code : PHP

```
<?php
$obj = new MaClasse;
$obj ('Petit test'); // Utilisation de l'objet comme fonction.
?>
```

Essayez ce code et... BAM ! Une erreur fatale (c'est bizarre 🤔). Plus sérieusement, pour résoudre ce problème, on va devoir utiliser la méthode magique `__invoke`. Elle est appelée dès qu'on essaye d'utiliser l'objet comme fonction (comme on vient de faire). Cette méthode comprend autant de paramètres que d'arguments passés à la fonction.

Exemple :

Code : PHP

```
<?php
class MaClasse
{
    public function __invoke ($argument)
    {
        echo $argument;
    }
}

$obj = new MaClasse;

$obj (5); // Affiche « 5 ».
```

Ce tutoriel portant sur les méthodes magiques s'arrête ici. Je parlerai de la méthode `__clone` lors du clonage d'objets en deuxième partie. 🤔

Note : comme vous l'avez remarqué, tous les noms de méthodes magiques commencent par deux underscores (`__`). Il est donc fortement conseillé de ne créer aucune méthode commençant par ce préfixe au risque d'implémenter une méthode magique sans le vouloir.

Voilà cette première partie posant les bases terminée. Avec les connaissances que vous avez, vous pouvez déjà créer de belles classes. 🤔

Partie 2 : Techniques avancées

Vous croyez tout savoir sur la POO en PHP ? Vous êtes loin du compte ! Cette partie vous montrera beaucoup de choses possibles avec la POO. 🤔

Les objets en profondeur

Continuons tranquillement notre ascension au sommet de la POO en PHP. Nous verrons dans ce chapitre quelques petites astuces concernant nos objets. Nous verrons également la dernière méthode magique dont je vous avais parlé. 🤔

Un objet, un identifiant

Je vais commencer cette partie en vous faisant une révélation : quand vous instanciez une classe, la variable stockant l'objet ne stocke en fait pas l'objet lui-même, mais un identifiant qui représente cet objet. C'est-à-dire qu'en faisant `$objet = new Classe;`, `$objet` ne contient pas l'objet lui-même, mais son identifiant unique. C'est un peu comme quand vous enregistrez des informations dans une BDD : la plupart du temps, vous avez un champ "id" unique qui représente l'entrée. Quand vous faites une requête SQL, vous sélectionnez l'élément en fonction de son id. Et bien là c'est pareil : quand vous accédez à un attribut ou à une méthode de l'objet, PHP regarde l'identifiant contenu dans la variable, va chercher l'objet correspondant et effectue le traitement nécessaire. Il est très important que vous compreniez ça, sinon vous allez être complètement perdus pour la suite du chapitre.

On a donc vu que la variable `$objet` contenait l'identifiant de l'objet qu'elle a instancié. Vérifions cela :

Code : PHP

```

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

$a = new MaClasse;

$b = $a; // On assigne à $b l'identifiant de $a, donc $a et $b représentent le même objet

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

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

```

Je recommande plus en détails la ligne 10 pour ceux qui sont un peu perdus. On a dit plus haut que \$a ne contenait pas l'objet lui-même mais son identifiant (un identifiant d'objet). \$a contient donc l'identifiant représentant l'objet créé. Ensuite, on assigne à \$b la valeur de \$a. Donc qu'est-ce que \$b vaut maintenant ? Et bien la même chose que \$a, à savoir l'identifiant qui représente l'objet ! \$a et \$b font donc référence à la même instance. 😊

Schématiquement, on peut représenter le code ci-dessus comme ceci :



Comme vous le voyez sur l'image, en réalité, il n'y a qu'un seul objet, qu'un seul identifiant, mais deux variables contenant exactement le même identifiant d'objet. Tout ceci peut sembler abstrait, donc allez à votre rythme pour bien comprendre. 😊

Maintenant que l'on sait que ces variables ne contiennent pas d'objet mais un identifiant d'objet, vous êtes censés savoir que 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 ! Ainsi, vous n'êtes pas obligé de passer l'objet en référence, car vous passerez une référence de l'identifiant de l'objet. Inutile, donc. 😊

Maintenant un problème se pose. Comment faire pour copier un objet ? Comment faire pour pouvoir copier tous ses attributs et valeurs dans un nouvel objet unique ? On a vu qu'on ne pouvait pas faire un simple \$objet1 = \$objet2 pour arriver à cela. Comme vous vous en doutez peut-être, c'est là qu'intervient le clonage d'objet.

Pour cloner un objet, c'est assez simple. Il faut utiliser le mot-clé clone juste avant l'objet à copier. Exemple :

Code : PHP

5

6

4

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

C'est aussi simple que cela. Avec ça, les deux objets contiennent des identifiants différents : par conséquent, si on veut modifier l'un d'eux, on peut le faire sans qu'aucune propriété de l'autre ne soit modifiée. 😊

Dis, t'avais pas parlé d'une méthode magique ?

Si si, j'y viens. 😊

Lorsque vous clonez un objet, la méthode `__clone` de celui-ci sera appelée (du moins, si vous l'avez définie). Vous ne pouvez pas appeler cette méthode directement. C'est la méthode `__clone` de l'objet à cloner qui est appelée, pas la méthode `__clone` du nouvel objet créé. 😊

Vous pouvez utiliser cette méthode pour modifier certains attributs pour l'ancien objet, ou alors incrémenter un compteur d'instances par exemple.

Code : PHP

```
<?php
class MaClasse
{
    private static $instances = 0;

    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();
?>
```

Ce qui affichera :

Citation : Résultat

Nombre d'instances de MaClasse : 2

Comparons nos objets

Nous allons maintenant voir comment comparer deux objets. Comparer deux objets est très simple, il suffit de faire comme vous

avez toujours fait en comparant des chaînes de caractères ou des nombres. Voici un exemple :

Code : PHP

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

Cette partie ne vous expliquera donc pas comment comparer des objets mais vous expliquera la démarche que PHP exécute pour les comparer et les effets que ces comparaisons peuvent produire.

Reprenons le code ci-dessus. Pour que la condition renvoie true, il faut que \$objet1 et \$objet2 aient les mêmes attributs et les mêmes valeurs, mais également que les deux objets soient des instances de la même classe. C'est-à-dire que même s'ils ont les mêmes attributs et valeurs mais que l'un est une instance de la classe A et l'autre une instance de la classe B, la condition renverra false. 😊

Exemple :

Code : 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 == $b';
else
    echo '$a != $b';

echo '<br />';

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

Si vous avez bien suivi, vous savez ce qui va s'afficher, à savoir :

Citation : Résultat

```
$a != $b  
$a == $c
```

Comme on peut le voir, \$a et \$b ont beau avoir les mêmes attributs et les mêmes valeurs, ils ne sont pas identiques car ils ne sont pas des instances de la même classe. Par contre, \$a et \$c sont bien identiques. 😊

Parlons maintenant de l'opérateur === qui permet de vérifier que deux objets sont strictement identiques. Vous n'avez jamais entendu parler de cet opérateur ? [Allez lire ce tutoriel](#) !

Cet opérateur vérifiera si les deux objets font référence vers la même instance. Il vérifiera donc que les deux identifiants d'objets comparés sont les mêmes. Allez relire la première partie de ce chapitre si vous êtes un peu perdu. 😊

Faisons quelques tests pour être sûr que vous avez bien compris :

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

Et à l'écran s'affichera :

Citation : Résultat

```
$a !== $b  
$a === $c
```

On voit donc que cette fois ci, la condition qui renvoyait true avec l'opérateur == renvoie maintenant false. \$a et \$c font référence à la même instance, la condition renvoie donc true. 😊

Parcourons nos objets

Finissons en douceur par voir comment parcourir nos objets, et ce en quoi ça consiste.

Le fait de parcourir un objet consiste à lire tous les attributs visibles de l'objet. Qu'est-ce que ça veut dire ? Ceci veut tout simplement dire que vous ne pourrez pas lire les attributs privés ou protégés en dehors de la classe, mais l'inverse est tout à fait possible. Je ne vous apprend rien de nouveau me direz-vous, mais ce rappel me semblait important pour vous expliquer le parcours d'objets.

Qui dit "parcours" dit "boucle". Quelle boucle devons-nous utiliser pour parcourir un objet ? Et bien la même boucle que pour parcourir un tableau... J'ai nommé foreach !

Son utilisation est d'une simplicité remarquable (du moins, si vous savez parcourir un tableau). Sa syntaxe est la même. Il y en a deux possibles :

- 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.

Vous ne devez sans doute pas être dépaycé, il n'y a presque rien de nouveau. Comme je vous l'ai dit, la boucle foreach parcourt les attributs visibles. Faisons quelques tests. Normalement, vous devez sans doute vous attendre au bon résultat (enfin, j'espère, mais si vous êtes tombé à côté de la plaque ce n'est pas un drame ! 😊).

Code : 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 pour que ce ne soit pas
    {
        foreach ($this as $attribut => $valeur)
            echo '<strong>', $attribut, '</strong> => ', $valeur, '<br />';
    }
}

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

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

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

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

foreach ($classe as $attribut => $valeur)
    echo '<strong>', $attribut, '</strong> => ', $valeur, '<br />';
```

Ce qui affichera :

Citation : Résultat

```
---- Liste les attributs depuis l'intérieur de la classe principale ----
attribut1 => Premier attribut public
attribut2 => Deuxième attribut public
attributProtege1 => Premier attribut protégé
attributProtege2 => Deuxième attribut protégé
attributPrive1 => Premier attribut privé
attributPrive2 => Deuxième attribut privé

---- Liste les attributs depuis l'intérieur de la classe enfant ----
attribut1 => Premier attribut public
attribut2 => Deuxième attribut public
attributProtege1 => Premier attribut protégé
attributProtege2 => Deuxième attribut protégé

---- Liste les attributs depuis le script global ----
attribut1 => Premier attribut public
attribut2 => Deuxième attribut public
```

Voici la fin de ce chapitre. J'espère que l'histoire des identifiants ne vous a pas trop embrouillé. C'est pourtant quelque chose d'assez important à mes yeux de savoir comment PHP fonctionne en interne. 🤔

J'ai volontairement terminé ce chapitre par le parcours d'objets. Pourquoi ? Car dans le prochain chapitre nous verrons comment modifier le comportement de l'objet quand il est parcouru grâce aux interfaces ! Celles-ci permettent de réaliser beaucoup de choses pratiques, mais je ne vous en dis pas plus. 😊

Les interfaces

Nous voici dans le chapitre concernant les interfaces. Nous allons voir comment imposer une structure à nos classes, puis nous nous amuserons (enfin si je peux me permettre d'utiliser ce terme 🤔) un peu avec les interfaces prédéfinies. 😊

Présentation et création d'interfaces

Le rôle d'une interface

Techniquement, une interface est une classe entièrement abstraite. Son rôle est de décrire un comportement à notre objet. Les interfaces ne doivent pas être confondues avec l'héritage : l'héritage représente un sous-ensemble (exemple : un magicien est un sous-ensemble d'un personnage). Ainsi, 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.

Créer une interface

Une interface se déclare avec le mot-clé `interface`, suivi du nom de l'interface, suivi d'une paire d'accolades. C'est entre ces accolades que vous listerez des méthodes. Par exemple, voici une interface pouvant représenter le point commun évoqué ci-dessus :

Code : PHP

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


1. Toutes les méthodes présentes dans une interface doivent être publiques ;
2. Une interface ne peut pas lister de méthodes abstraites ou finales ;
3. Une interface ne peut pas avoir le même nom qu'une classe, et vice-versa

Implémenter une interface

Cette interface étant toute seule, elle est un peu inutile. Il va donc falloir implémenter l'interface à notre classe grâce au mot-clé implements ! La démarche à exécuter est comme quand on faisait hériter une classe d'une autre, à savoir :

Code : PHP

```
<?php
class Personnage implements Movable
{
}
?>
```

Essayez ce code et...

Citation : Résultat

Fatal error: Class Personnage contains 1 abstract method and must therefore be declared abstract or implement the remaining methods (Movable::move)

Et oui, une erreur fatale est générée car notre classe Personnage n'a pas implémenté la méthode présente dans l'interface Movable. Pour que ce code ne génère aucune erreur, il faut qu'il y ait au minimum ce code :

Code : PHP

```
<?php
class Personnage implements iPersonnage
{
    public function move($dest)
    {
    }
}
?>
```

Et là... L'erreur a disparu !

Vous pouvez très bien, dans votre classe, définir une méthode comme étant abstraite ou finale. 😊

Si vous héritez une classe et que vous implémentez une interface, alors vous devez d'abord spécifier la classe à hériter avec le mot-clé extends puis les interfaces à implémenter avec le mot-clé implements.

Une interface vous oblige à écrire toutes ses méthodes, mais vous pouvez en rajouter autant que vous voulez. 😊

Vous pouvez très bien implémenter plus d'une interface par classe, à condition que celles-ci n'aient aucune méthode portant le même nom ! Exemple :

Code : 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 :

Code : 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, vous pouvez hériter vos interfaces grâce à l'opérateur extends. Vous ne pouvez réécrire ni une méthode ni une constante qui a déjà été listée dans l'interface parente. Exemple :

Code : PHP

```

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

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

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. Il vous suffit de séparer leur nom par une virgule. Exemple :

Code : PHP

```

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

interface iB
{
    public function test2();
}

interface iC extends iA, iB
{
    public function test3();
}
?>

```

Dans cet exemple, si on imagine une classe implémentant iC, celle-ci devra implémenter les trois méthodes test1, test2 et test3.

Interfaces prédéfinies

Nous allons maintenant voir les interfaces prédéfinies. Grâce à certaines, on va pouvoir modifier le comportement de nos objets ou réaliser plusieurs choses pratiques. Il y a beaucoup d'interfaces prédéfinies, je ne vous les présenterai pas toutes, juste 4. Déjà avec celles-là on va pouvoir réaliser de belles choses, et puis vous êtes libres de lire la documentation pour découvrir toutes les

interfaces. On va essayer ici de créer un « tableau-objet ».

L'interface Iterator

Commençons d'abord par l'interface Iterator. Si votre classe implémente cette interface, alors vous pourrez modifier le comportement de votre objet lorsqu'il est parcouru. Cette interface comporte 5 méthodes :

- current : renvoie l'élément courant ;
- key : retourne la clé de l'élément courant ;
- next : déplace le pointeur sur l'élément suivant ;
- rewind : remet le pointeur sur le premier élément ;
- valid : vérifie si la position courante est valide.

En écrivant ces méthodes, on pourra renvoyer la valeur qu'on veut, et pas forcément la valeur de l'attribut actuellement lu. Imaginons qu'on ait un attribut qui soit un array. On pourrait très bien créer un petit script qui, au lieu de parcourir l'objet, parcourt le tableau ! Je vous laisse essayer. Vous aurez besoin d'un attribut \$position qui stocke la position actuelle. 🤔

Correction :

Secret ([cliquez pour afficher](#))

Ce qui affichera :

Citation : Résultat

```
0 => Premier élément
1 => Deuxième élément
2 => Troisième élément
3 => Quatrième élément
4 => Cinquième élément
```

Alors, pratique non ? 😊

L'interface SeekableIterator

Cette interface hérite de l'interface Iterator, on n'aura donc pas besoin d'implémenter les deux à notre classe. 😊

SeekableIterator ajoute une méthode à la liste des méthodes d'Iterator : la méthode seek. Cette méthode permet de placer le curseur interne à une position précise. Elle demande donc un argument : la position du curseur à laquelle il faut le placer. Je vous déconseille de modifier directement l'attribut \$position afin d'assigner directement la valeur de l'argument à \$position. En effet, qui vous dit que la valeur de l'argument est une position valide ?

Je vous laisse réfléchir quant à l'implémentation de cette méthode. Voici la correction (j'ai repris la dernière classe) :

Secret ([cliquez pour afficher](#))

Ce qui affichera :

Citation : Résultat

```
0 => Premier élément
1 => Deuxième élément
2 => Troisième élément
3 => Quatrième élément
4 => Cinquième élément
```

Troisième élément

L'interface `ArrayAccess`

Nous allons enfin, grâce à cette interface, pouvoir placer des crochets à la suite de notre objet avec la clé à laquelle accéder, comme sur un vrai tableau ! L'interface `ArrayAccess` liste 4 méthodes :

- `offsetExists` : méthode qui vérifiera l'existence de la clé entre crochets lorsque l'objet est passé à la fonction `isset` ou `empty` (cette valeur entre crochet est passé à la méthode en paramètre) ;
- `offsetGet` : méthode appelée lorsqu'on fait un simple `$obj['clé']`. La valeur 'clé' est donc passée à la méthode `offsetGet` ;
- `offsetSet` : méthode appelée lorsqu'on assigne une valeur à une entrée. Cette méthode reçoit donc deux arguments, la valeur de la clé et la valeur qu'on veut lui assigner.
- `offsetUnset` : méthode appelée lorsqu'on appelle la fonction `unset` sur l'objet avec une valeur entre crochets. Cette méthode reçoit un argument, la valeur qui est mise entre les crochets.

Maintenant, votre mission, c'est d'implémenter cette interface et de gérer l'attribut `$tableau` grâce aux 4 méthodes. C'est parti ! 😊

Correction :

Secret ([cliquez pour afficher](#))

Ce qui affiche :

Citation : Résultat

Parcours de l'objet...

```
0 => Premier élément
1 => Deuxième élément
2 => Troisième élément
3 => Quatrième élément
4 => Cinquième élément
```

Remise du curseur en troisième position...

Élément courant : Troisième élément

Affichage du troisième élément : Troisième élément

Modification du troisième élément... Nouvelle valeur : Hello world !

Destruction du quatrième élément...

Tout se passe bien, `$objet[3]` n'existe plus !

Alors, on se rapproche vraiment du comportement d'un tableau, hein ? On peut faire tout ce qu'on veut, comme sur un tableau ! Enfin, il manque juste un petit truc pour que ce soit parfait quand même...

L'interface `Countable`

Et voici la dernière interface que je vous présenterai. Elle contient une méthode : la méthode `count`. Celle-ci doit obligatoirement renvoyer un entier qui sera la valeur renvoyée par la fonction `count` appelée sur notre objet. Cette méthode n'est pas bien compliquée à implémenter, il suffit juste de retourner le nombre d'entrées de notre tableau. 😊

Correction :

Secret ([cliquez pour afficher](#))

Ce qui affichera :

Citation : Résultat

Parcours de l'objet...

0 => Premier élément

1 => Deuxième élément

2 => Troisième élément

3 => Quatrième élément

4 => Cinquième élément

Remise du curseur en troisième position...

Élément courant : Troisième élément

Affichage du troisième élément : Troisième élément

Modification du troisième élément... Nouvelle valeur : Hello world !

Actuellement, mon tableau comporte 5 entrées

Destruction du quatrième élément...

Tout se passe bien, \$objet[3] n'existe plus !

Maintenant, il n'en comporte plus que 4 !

Bonus : la classe ArrayIterator

Je dois vous avouer quelque chose : la classe qu'on vient de créer pour pouvoir créer des « objets-tableaux » existe déjà. En effet, PHP possède nativement une classe nommée ArrayIterator. Comme notre précédente classe, celle-ci implémente les quatre interfaces qu'on a vues.

Mais pourquoi tu nous as fais faire tout ça ?!

Pour faire pratiquer, tiens. 😊

Sachez que réécrire des classes ou fonctions natives de PHP est un excellent exercice (et c'est valable pour tous les langages de programmations). Je ne vais pas m'attarder sur cette classe, étant donné qu'elle s'utilise exactement comme la notre. Elle possède les mêmes méthodes, à une différence près : cette classe implémente un constructeur qui accepte un tableau en guise d'argument. Comme vous l'aurez deviné, c'est ce tableau qui sera « transformé » en objet. Ainsi, si vous faites un `echo $monInstanceArrayIterator['cle']`, alors à l'écran s'affichera l'entrée qui a pour clé cle du tableau passé en paramètre. 😊

Voici la fin du chapitre. Rendez-vous au prochain sur les exceptions. 😊

Les exceptions

Nous allons découvrir dans ce chapitre une différente façon de gérer les erreurs. Nous allons, en quelques sortes, créer nos propres types d'erreurs. Actuellement, vous connaissez les erreurs fatales, les alertes, les erreurs d'analyse ou encore les notices. Les exceptions sont des erreurs assez différentes qui ne fonctionnent pas de la même façon, nous allons donc tout de suite commencer ce chapitre en abordant cette nouvelle notion ! 😊

Une différente gestion des erreurs

Les exceptions, comme dit dans l'introduction de ce chapitre, sont une façon différente de gérer les erreurs. Celles-ci sont en fait des erreurs lancées par PHP lorsque quelque chose qui ne va pas est survenu. Nous allons commencer par lancer nos propres exceptions. Pour cela, on va devoir s'intéresser à la classe Exception.

Lancer une exception

Une exception peut être lancée depuis n'importe où dans le code. Quand on lance une exception, on doit, en gros, lancer une instance de la classe Exception. Cet objet lancé contiendra le message d'erreur ainsi que son code. Pensez à spécifier au moins le message d'erreur, bien que celui-ci soit facultatif. Je ne vois pas l'intérêt de lancer une exception sans spécifier l'erreur rencontrée. 😊 Pour le code d'erreur, il n'est pas (pour l'instant) très utile. Libre à vous de le spécifier ou pas. Le troisième et dernier argument est l'exception précédente. Là aussi, spécifiez-la si vous le souhaitez, mais ce n'est pas indispensable.

Passons à l'acte. Nous allons créer une simple fonction qui aura pour rôle d'additionner un nombre avec un autre. Si l'un des deux nombres n'est pas numérique, alors on lancera une exception de type Exception à l'aide du mot throw (= lancer). On va donc lancer une nouvelle Exception. Le constructeur de la classe Exception demande en paramètre le message d'erreur, son code et l'exception précédente. Ces trois paramètres sont facultatifs.

Code : PHP

```
<?php
function additionner ($a, $b)
{
    if (!is_numeric ($a) OR !is_numeric ($b))
        // On lance une nouvelle exception grâce à throw et on instancie directement une
        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);
```

Et là, vous avez :

Citation : Résultat

15

Fatal error: Uncaught exception 'Exception' with message 'Les deux paramètres doivent être des nombres' in C:\wamp\www\Tests\pdo.php:5 Stack trace: #0 C:\wamp\www\Tests\exceptions.php(11): additionner('azerty', 54) #1 {main} thrown in C:\wamp\www\Tests\exceptions.php on line 5

Et voilà votre première exception qui apparaît d'une façon assez désagréable devant vos yeux ébahis. Décortiquons ce que PHP veut bien nous dire.

Premièrement, il génère une erreur fatale. Eh oui, une exception non attrapée génère automatiquement une erreur fatale. Nous verrons plus tard ce que veut dire « attraper ».

Deuxièmement, il nous dit Uncaught exception 'Exception' with message 'Les deux paramètres doivent être des nombres' ce qui signifie « Exception 'Exception' non attrapée avec le message 'Les deux paramètres doivent être des nombres' ». Ce passage se passera de commentaire, la traduction parle d'elle-même : on n'a pas attrapé l'exception 'Exception' (= le nom de la classe instanciée par l'objet qui a été lancé) avec tel message (ici, c'est le message qu'on a spécifié dans le constructeur).

Et pour finir, PHP nous dit où a été lancée l'exception, depuis quelle fonction, à quelle ligne etc.

Maintenant, puisque PHP n'a pas l'air content que l'on n'ait pas « attrapé » cette exception, et bien c'est ce qu'on va faire.

Ne lancez jamais d'exception dans un destructeur. Si vous faites une telle chose, vous aurez une erreur fatale : Exception thrown without a stack frame in Unknown on line 0. Cette erreur peut aussi être lancée dans un autre cas évoqué plus tard.

Attraper une exception

Afin d'attraper une exception, encore faut-il qu'elle soit lancée. Le problème, c'est qu'on ne peut pas dire à PHP que toutes les exceptions lancées doivent être attrapées : c'est à nous de lui dire que l'on va essayer d'effectuer telles instructions et, si une exception est lancée, alors on attrapera celle-ci afin qu'aucune erreur fatale ne soit lancée et que de tels messages ne s'affichent plus.

On va dès à présent placer nos instructions dans un bloc try. Celles-ci seront à placer entre une paire d'accolades. Qui dit bloc try dit aussi bloc catch car l'un ne va pas sans l'autre (si vous mettez l'un sans l'autre, une erreur d'analyse sera levée). Ce bloc catch, lui, a une petite particularité. Au lieu de placer catch suivi directement des deux accolades, on va devoir spécifier entre une paire de parenthèses placée entre catch et l'accolade ouvrante le type d'exception à attraper suivi d'une variable qui représentera cette exception. C'est à partir d'elle qu'on pourra récupérer le message ou le code d'erreur grâce aux méthodes de la classe.

Commençons en douceur en attrapant simplement toute exception Exception :

Code : PHP

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

    return $a + $b;
}

try // On va 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) // On va attraper les exceptions "Exception" s'il y en a une qui est lancée
{
    // Ici on pourrait récupérer le message d'erreur avec $e->getMessage()
    // ou le code d'erreur avec $e->getCode()
}
```

Et là, miracle, vous n'avez plus que 15 qui s'affiche, et plus d'erreur ! Par contre, les deux autres résultats ne sont pas affichés, et il serait intéressant de savoir pourquoi. On va afficher le message d'erreur. Pour ce faire, il faut appeler la méthode getMessage(). Si vous souhaitez récupérer le code d'erreur, il faut appeler getCode().

Code : PHP


```

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

    return $a + $b;
}

try // On va 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) // On va attraper les exceptions "Exception" s'il y en a une qu
{
    echo 'Une exception a été lancée. Message d\'erreur : ', $e->getMessage();
}

```

Ce qui affichera :

Citation

```

15
Une exception a été lancée. Message d'erreur : Les deux paramètres doivent être des nombres

```

Comme vous pouvez le constater, la troisième instruction du bloc try n'a pas été exécutée. C'est normal puisque la deuxième instruction a interrompu la lecture du bloc. Si vous interceptez les exceptions comme on a fait, alors le script n'est pas interrompu. En voici la preuve :

Code : PHP

```

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

    return $a + $b;
}

try // On va 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) // On va attraper les exceptions "Exception" s'il y en a une qu
{
    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

```

Vous vous demandez sans doute pourquoi on doit spécifier le nom de l'exception à intercepter puisque c'est toujours une instance de la classe Exception. En fait, la classe Exception est la classe de base pour toute exception qui doit être lancée, ce qui signifie que l'on peut lancer n'importe quelle autre instance d'une classe, du moment qu'elle hérite de la classe Exception !

Des exceptions spécialisées

Hériter la classe Exception

PHP nous offre la possibilité d'hériter la classe Exception afin de personnaliser nos exceptions. Par exemple, nous pouvons créer une classe MonException qui réécrira des méthodes de la classe Exception ou en créera de nouvelles qui lui seront propres.

Avant de foncer tête baissée dans l'écriture de notre classe personnalisée, encore faut-il savoir quelles méthodes et attributs nous seront disponibles. Voici la liste des attributs et méthodes de la classe Exception tirée [de la documentation](#) :

Code : 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 formatée de trace

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

Ainsi, nous voyons que l'on a accès aux attributs protégés de la classe et qu'on peut réécrire les méthodes __construct et __toString. Toutes les autres méthodes sont finales, on n'a donc pas le droit de les réécrire.

Nous allons donc créer notre classe MonException qui, par exemple, réécrira le constructeur en rendant obligatoire le premier argument, ainsi que la méthode __toString pour n'afficher que le message d'erreur (c'est uniquement ça qui nous intéresse).

Code : PHP

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

    public function __toString()
    {
        return $this->message;
    }
}
?>
```

Maintenant, comme vous l'aurez peut-être deviné, nous n'allons pas lancer d'exception Exception mais une exception de type MonException.

Dans notre script, on ne va désormais attraper que les exceptions MonException, ce qui éclaircira le code car c'est une manière de se dire que l'on ne travaille, dans le bloc try, qu'avec des instructions susceptibles de lancer des exceptions de type MonException.

Exemple :

Code : 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) OR !is_numeric ($b))
        throw new MonException ('Les deux paramètres doivent être des nombres'); // c

    return $a + $b;
}

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

catch (MonException $e) // On va attraper les exceptions "MonException" s'il y en a
{
    echo $e; // On affiche le message d'erreur grâce à la méthode __toString que l'on
}

echo '<br />Fin du script'; // Ce message s'affiche, ça prouve bien que le script est
```

Ainsi, on a attrapé uniquement les exceptions de type MonException. Essayez de lancer une exception Exception à la place, et vous verrez qu'elle ne sera pas attrapée. Si vous décidez d'attraper, dans le bloc catch, les exceptions Exception, alors toutes les exceptions seront attrapées car elles héritent toutes de cette classe. En fait, quand vous héritez une classe d'une autre et que vous décidez d'attraper les exceptions de la classe parente, alors celles de la classe enfant le seront aussi.

Emboîter plusieurs blocs catch

Il est possible d'emboîter plusieurs blocs catch. En effet, vous pouvez mettre un premier bloc attrapant les exceptions MonException suivi d'un deuxième attrapant les exceptions Exception. Si vous effectuez une telle opération et qu'une exception est lancée, alors PHP ira dans le premier bloc pour voir si ce type d'exception doit être attrapé, si tel n'est pas le cas il va dans le deuxième etc. jusqu'à ce qu'il tombe sur un bloc qui l'attrape. Si aucun ne l'attrape, alors une erreur fatale est levée. Exemple :

Code : 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) OR !is_numeric ($b))
        throw new MonException ('Les deux paramètres doivent être des nombres'); // C

    if (func_num_args() > 2)
        throw new Exception ('Pas plus de deux arguments ne doivent être passés à la

    return $a + $b;
}

try // On va essayer d'effectuer les instructions situées dans ce bloc
{
    echo additionner (12, 3), '<br />';
    echo additionner (15, 54, 45), '<br />';
}

catch (MonException $e) // On va attraper les exceptions "MonException" s'il y en a
{
    echo '[MonException] : ', $e; // On affiche le message d'erreur grâce à la méthode
}

catch (Exception $e) // Si l'exception n'est toujours pas attrapée, alors on va essay
{
    echo '[Exception] : ', $e->getMessage(); // La méthode __toString() nous affiche
}

echo '<br />Fin du script'; // Ce message s'affiche, ça prouve bien que le script est

```

Cette fois-ci, aucune exception MonException n'est lancée, mais une exception Exception l'a été. PHP va donc effectuer les opérations demandées dans le deuxième bloc catch.

Exemple concret : la classe PDOException

Vous connaissez sans doute la bibliothèque PDO ([lire le tutoriel à ce sujet](#)). Dans ce tutoriel, il vous est donné une technique pour capturer les erreurs générées par PDO : comme vous le savez maintenant, ce sont des exceptions ! Et PDO a sa classe d'exception : PDOException. Celle-ci n'hérite pas directement de la classe Exception mais de RuntimeException. Cette classe n'a rien de plus que sa classe mère, il s'agit juste d'une classe qui est instanciée pour émettre une exception lors de l'exécution du script. Il existe une classe pour chaque circonstance dans laquelle l'exception est lancée : vous trouverez [la liste des exceptions ici](#).

Cette classe PDOException est donc la classe personnalisée pour émettre une exception par la classe PDO ou PDOStatement. Sachez d'ailleurs que si une extension orientée objet doit émettre une erreur, elle émettra une exception.

Bref, voici un exemple d'utilisation de PDOException :

Code : PHP

```

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

}

catch (PDOException $e) // On attrape les exceptions PDOException
{
    echo 'La connexion a échoué.<br />';
    echo 'Informations : [' , $e->getCode(), ']' , $e->getMessage(); // On affiche le
}
}

```

Exceptions pré-définies

Il existe toute une quantité d'exceptions pré-définies. Vous pouvez obtenir cette liste sur [la documentation](#). Au lieu de lancer tout le temps une exception en instanciant Exception, il est préférable d'instancier la classe adaptée à la situation. Par exemple, reprenons le code proposé en début de chapitre :

Code : PHP

```

<?php
function additionner ($a, $b)
{
    if (!is_numeric ($a) OR !is_numeric ($b))
        // On lance une nouvelle exception grâce à throw et on instancie directement un
        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);

```

La classe à instancier ici est celle qui doit l'être lorsqu'un paramètre est invalide. On regarde la documentation, et on tombe sur `InvalidArgumentException`. Le code donnerait donc :

Code : PHP

```

<?php
function additionner ($a, $b)
{
    if (!is_numeric ($a) OR !is_numeric ($b))
        throw new InvalidArgumentException ('Les deux paramètres doivent être des nom

    return $a + $b;
}

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

```

Cela permet de mieux se repérer dans le code et surtout de mieux cibler les erreurs grâce aux multiples blocs catch. 😊

Gérer les erreurs facilement

Convertir les erreurs en exceptions

Nous allons voir une dernière chose avant de passer au prochain chapitre : nous allons voir comment convertir les erreurs fatales, les alertes et les notices en exceptions. Pour cela, nous allons avoir besoin de la fonction [set_error_handler](#). Celle-ci permet d'enregistrer une fonction en callback qui sera appelée à chaque fois qu'une de ces trois erreurs sera lancée. Il n'y a pas de rapport direct avec les exceptions : c'est à nous de l'établir.

Notre fonction, que l'on nommera `error2exception` par exemple, doit demander entre deux et cinq paramètres.

- Le numéro de l'erreur (obligatoire) ;
- Le message d'erreur (obligatoire) ;
- Le nom du fichier dans lequel l'erreur a été lancée ;
- Le numéro de la ligne à laquelle l'erreur a été identifiée ;
- Un tableau avec toutes les variables qui existaient jusqu'à ce que l'erreur soit rencontrée.

Nous n'allons pas prêter attention au dernier paramètre, juste aux quatre premiers. Nous allons créer notre propre classe `MonException` qui hérite non pas de `Exception` mais de `ErrorException`. Bien sûr, comme je l'ai déjà dit plus haut, toutes les exceptions héritent de la classe `Exception` : `ErrorException` n'échappe pas à la règle et hérite de celle-ci.

La fonction `set_error_handler` demande deux paramètres. Le premier s'agit de la fonction à appeler, et le deuxième s'agit des erreurs à intercepter. Par défaut, ce paramètre intercepte toutes les erreurs, y compris les erreurs strictes.

Le constructeur de la classe `ErrorException` demande cinq paramètres, tous facultatifs.

- Le message d'erreur ;
- Le code de l'erreur ;
- La sévérité de l'erreur (erreur fatale, alerte, notice, etc.) représentées par des [constantes pré-définies](#) ;
- Le fichier où l'erreur a été rencontrée ;
- La ligne à laquelle l'erreur a été rencontrée.

Voici à quoi pourrait ressembler le code de base :

Code : PHP

```

<?php
class MonException extends Exception
{
    public function __toString()
    {
        switch ($this->severity)
        {
            case E_USER_ERROR : // Si l'utilisateur émet une erreur fatale
                $type = 'Erreur fatale';
                break;

            case E_WARNING : // Si PHP émet une alerte
            case E_USER_WARNING : // Si l'utilisateur émet une alerte
                $type = 'Attention';
                break;

            case E_NOTICE : // Si PHP émet une notice
            case E_USER_NOTICE : // Si l'utilisateur émet une notice
                $type = 'Note';
                break;

            default : // Erreur inconnue
                $type = 'Erreur inconnue';
                break;
        }

        return '<strong>' . $type . '</strong> : [' . $this->code . ']' . $this->message;
    }
}

function error2exception ($code, $message, $fichier, $ligne)
{
    // Le code fait office de sévérité
    // Reportez-vous aux constantes prédéfinies pour en savoir plus
    // http://fr2.php.net/manual/fr/errorfunc.constants.php
    throw new MonException ($message, 0, $code, $fichier, $ligne);
}

set_error_handler ('error2exception');

```

Vous voyez que dans la méthode `__toString` je mettais à chaque fois `E_X` et `E_USER_X`. Les erreurs du type `E_X` sont générées par PHP et les erreurs `E_USER_X` sont générées par l'utilisateur grâce à [trigger_error](#). Les erreurs `E_ERROR` (donc les erreurs fatales générées par PHP) ne peuvent être interceptées, c'est la raison pour laquelle je ne l'ai pas placé dans le switch.

Ensuite, à vous de faire des tests, vous verrez bien que ça fonctionne à merveille. Mais gardez bien ça en tête : avec ce code, toutes les erreurs (même les notices) qui ne sont pas dans un bloc try interrompent le script car elles émettront une exception !

On aurait très bien pu utiliser la classe `Exception` mais `Exception` a été conçu exactement pour ce genre de chose. Nous n'avons pas besoin de créer d'attribut stockant la sévérité de l'erreur ou de réécrire le constructeur pour y stocker le nom du fichier et la ligne à laquelle s'est produite l'erreur.

Personnaliser les exceptions non attrapées

Nous avons réussi à transformer toutes nos erreurs en exceptions en les interceptant grâce à `set_error_handler`. Étant donné que la moindre erreur lèvera une exception, il serait intéressant de personnaliser l'erreur générée par PHP. Ce que je veux dire par là, c'est qu'une exception non attrapée génère une longue et laide erreur fatale. On va donc, comme pour les erreurs, intercepter les exceptions grâce à `set_exception_handler`. Cette fonction demande un seul argument : le nom de la fonction à appeler lorsqu'une exception est lancée. La fonction de callback doit accepter un argument : c'est un objet représentant l'exception.

Voici un exemple d'utilisation en reprenant le précédent code :

Code : PHP

```

<?php
class MonException extends ErrorException
{
    public function __toString()
    {
        switch ($this->severity)
        {
            case E_USER_ERROR : // Si l'utilisateur émet une erreur fatale
                $type = 'Erreur fatale';
                break;

            case E_WARNING : // Si PHP émet une alerte
            case E_USER_WARNING : // Si l'utilisateur émet une alerte
                $type = 'Attention';
                break;

            case E_NOTICE : // Si PHP émet une notice
            case E_USER_NOTICE : // Si l'utilisateur émet une notice
                $type = 'Note';
                break;

            default : // Erreur inconnue
                $type = 'Erreur inconnue';
                break;
        }

        return '<strong>' . $type . '</strong> : [' . $this->code . ']' . $this->message;
    }
}

function error2exception ($code, $message, $fichier, $ligne)
{
    // Le code fait office de sévérité
    // Reportez-vous aux constantes prédéfinies pour en savoir plus
    // http://fr2.php.net/manual/fr/errorfunc.constants.php
    throw new MonException ($message, 0, $code, $fichier, $ligne);
}

function customException ($e)
{
    echo 'Ligne ', $e->getLine(), ' dans ', $e->getFile(), '<br /><strong>Exception lancée</strong>';
}

set_error_handler ('error2exception');
set_exception_handler ('customException');

```

e l'exception est

l'exception est

Je dis bien que l'exception est interceptée et non attrapée ! Cela signifie que l'on attrape l'exception, qu'on effectue des opérations puis qu'on la relâche. On attrape l'exception, on effectue des opérations puis qu'on la relâche.

exception est attrapée et non attrapée !

ais d'exception dans votre gestionnaire

s d'exception dans votre gestionnaire

Ne lancez jamais d'exception dans votre gestionnaire d'exception (ici customException). En effet, cela créerait une boucle infinie puisque votre gestionnaire lance lui-même une exception. L'erreur lancée est la même que celle vue précédemment : il s'agit d'une exception fatale.

de l'exception fatale « Exception d'handler d'exception » (ici customException). En effet, cela créerait une boucle infinie puisque votre gestionnaire lance lui-même une exception.

terminé. La notion d'exception est assez importante car elle intervient dans la plupart des langages orientés objet (comme Java par exemple).

vité

té

rtie sur les notions avancées de la POO en découvrant ce qu'est l'API de réflexion en PHP. Cette API contient une multitude de méthodes afin d'obtenir des informations sur vos classes, vos interfaces, vos fonctions ou méthodes, vos attributs de

classes et sur les extensions. Bien entendu, nous ne les verrons pas toutes, nous nous intéresserons uniquement aux classes en rapport avec la POO. Bref, ne tardons pas, commençons de suite ! 😊

Obtenir des informations sur ses classes

Qui dit « Réflexivité » dit « Instanciation de classe ». Nous allons donc instancier une classe qui nous fournira des informations sur telle classe. Dans cette sous-partie, il s'agit de la classe `ReflectionClass`. Quand nous l'instancierons, nous devrons spécifier le nom de la classe sur laquelle on veut obtenir des informations. Nous allons prendre pour exemple la classe `Magicien` de [notre précédent TP](#).

Pour obtenir des informations la concernant, nous allons procéder comme suit :

Code : PHP

```
<?php
$classeMagicien = new ReflectionClass('Magicien'); // Le nom de la classe doit être en majuscule
?>
```

La classe `ReflectionClass` possède beaucoup de méthodes et je ne pourrai par conséquent toutes vous les présenter.

Il est aussi possible d'obtenir des informations sur une classe grâce à un objet. Nous allons pour cela instancier la classe `ReflectionObject` en fournissant l'instance en guise d'argument. Cette classe hérite de toutes les méthodes de `ReflectionClass` : elle ne réécrit que deux méthodes (dont le constructeur). La seconde méthode réécrite ne nous intéresse pas. Cette classe n'implémente pas de nouvelles méthodes.

Exemple d'utilisation très simple :

Code : PHP

```
<?php
$magicien = new Magicien (Personnage::EXISTANT, 'Vykl2');
$classeMagicien = new ReflectionObject ($magicien);
?>
```

Informations propres à la classe

Les attributs

Pour savoir si la classe possède tel attribut, tournons-nous vers `ReflectionClass::hasProperty($attributeName)`. Cette méthode retourne vrai si l'attribut passé en paramètre existe, et faux s'il n'existe pas. Exemple :

Code : PHP

```
<?php
if ($classeMagicien->hasProperty ('magie'))
    echo 'La classe Magicien possède un attribut $magie';
else
    echo 'La classe Magicien ne possède pas d\'attribut $magie';
?>
```

Vous pouvez aussi récupérer cet attribut afin d'obtenir des informations le concernant, comme on vient de faire jusqu'à maintenant avec notre classe. On verra ça plus tard, une sous-partie sera dédiée à ce sujet. 😊

Les méthodes

Si vous voulez savoir si la classe implémente telle méthode, alors il va falloir regarder du côté de `ReflectionClass::hasMethod($methodName)`. Celle-ci retourne vrai si la méthode est implémentée, ou faux si elle ne l'est pas. Exemple :

Code : PHP

```
<?php
    if ($classeMagicien->hasMethod ('lancerUnSort'))
        echo 'La classe Magicien implémente une méthode lancerUnSort()';
    else
        echo 'La classe Magicien n\'implémente pas de méthode lancerUnSort()';
?>
```

Vous pouvez, comme pour les attributs, récupérer une méthode, et une sous-partie sera dédiée à la classe permettant d'obtenir des informations sur telle méthode.

Les constantes

Là aussi il est possible de savoir si telle classe possède telle constante. Ceci est possible grâce à la méthode `ReflectionClass::hasConstant($constName)`. Exemple :

Code : PHP

```
<?php
    if ($classePersonnage->hasConstant('NOUVEAU'))
        echo 'La classe Personnage possède une constante NOUVEAU';
    else
        echo 'La classe Personnage ne possède pas de constante NOUVEAU';
?>
```

Il est aussi possible de récupérer la valeur de la constante grâce à `ReflectionClass::getConstant($constName)` :

Code : PHP

```
<?php
    if ($classePersonnage->hasConstant('NOUVEAU'))
        echo 'La classe Personnage possède une constante NOUVEAU (celle ci vaut ', $classe
    else
        echo 'La classe Personnage ne possède pas de constante NOUVEAU';
?>
```

Vous pouvez aussi récupérer la liste complète des constantes d'une classe sous forme de tableau grâce à `ReflectionClass::getConstants()` :

Code : PHP

```
<?php
    echo '<pre>', print_r ($classePersonnage->getConstants(), true), '</pre>';
?>
```

Les relations entre classes

L'héritage

Il est possible de récupérer la classe parente de notre classe. Pour cela, nous allons regarder du côté de `ReflectionClass::getParentClass()`. Cette méthode nous renvoie la classe parente s'il y en a une : la valeur de retour sera une instance de la classe `ReflectionClass` qui représentera la classe parente ! Si la classe ne possède pas de parent, alors la valeur de retour sera `false`.

Exemple :

Code : 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';
?>

```

Voici une belle occasion de vous présenter la méthode `ReflectionClass::getName()`. Cette méthode se contente de renvoyer le nom de la classe. Pour l'exemple avec le magicien ça aurait été inutile puisqu'on savait déjà le nom de la classe, mais ici on ne sait pas (quand je dis qu'on ne sait pas c'est que ce n'est pas déclaré explicitement dans les dernières lignes de code).

Dans le domaine de l'héritage, on peut aussi citer `ReflectionClass::isSubclassOf($className)`. Cette méthode nous renvoie vrai si la classe spécifiée en paramètre est le parent de notre classe. Exemple :

Code : PHP

```

<?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';
?>

```

Les deux prochaines méthodes que je vais vous présenter ne sont pas en rapport direct avec l'héritage mais sont cependant utilisées lorsque cette relation existe : il s'agit de savoir si la classe est abstraite ou finale. Nous avons pour cela les méthodes `ReflectionClass::isAbstract()` et `ReflectionClass::isFinal()`. Notre classe `Personnage` est abstraite, vérifions donc cela :

Code : 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';
?>

```

Dans le même genre, on peut citer `ReflectionClass::isInstantiable()` qui permet de savoir si notre classe est instanciable. Comme la classe `Personnage` est abstraite, elle ne l'est pas. Vérifions cela :

Code : PHP

```

<?php
    if ($classePersonnage->isInstantiable())
        echo 'La classe Personnage est instanciable';
    else
        echo 'La classe personnage n\'est pas instanciable';
?>

```

Bref, pas de grosse surprise. 😊

Les interfaces

Voyons maintenant les méthodes en rapport avec les interfaces. Comme je vous l'ai déjà dit, une interface n'est autre qu'une classe entièrement abstraite : nous pouvons donc instancier la classe `ReflectionClass` en spécifiant une interface en paramètre et vérifier si celle-ci est bien une interface grâce à la méthode `ReflectionClass::isInterface()`.

Dans les exemples qui vont suivre, on va admettre que la classe `Magicien` implémente une interface `iMagicien`.

Code : 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';
?>
```

Vous pouvez aussi savoir si telle classe implémente telle interface grâce à la méthode `ReflectionClass::implementsInterface($interfaceName)`. Exemple :

Code : PHP

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

Il est aussi possible de récupérer toutes les interfaces implémentées, interfaces contenues dans un tableau. Pour cela, deux méthodes sont à votre disposition : `ReflectionClass::getInterfaces()` et `ReflectionClass::getInterfaceNames()`. La première renvoie autant d'instances de la classe `ReflectionClass` qu'il y a d'interfaces, chacune représentant une interface. La seconde méthode se contente uniquement de renvoyer un tableau contenant le nom de toutes les interfaces implémentées. Je pense qu'il est inutile de donner un exemple sur ce point-là. 😊

Obtenir des informations sur les attributs de ses classes

Nous avons assez parlé de la classe, intéressons-nous maintenant à ses attributs. La classe qui va nous permettre d'en savoir plus à leur sujet est `ReflectionProperty`. Il y a deux moyens d'utiliser cette classe : l'instancier directement ou utiliser une méthode de `ReflectionClass` qui nous renverra une instance de `ReflectionProperty`.

Instanciation directe

L'appel du constructeur se fait en lui passant deux arguments. Le premier est le nom de la classe, et le second est le nom de l'attribut. Exemple :

Code : PHP

```
<?php
$attributMagie = new ReflectionProperty('Magicien', 'magie');
?>
```

Tout simplement. 😊

Récupération d'attribut d'une classe

Récupérer un attribut

Afin de récupérer un attribut d'une classe, on aura besoin de la méthode `ReflectionClass::getProperty($attrName)` :

Code : PHP

```
<?php
$classeMagicien = new ReflectionClass('Magicien');
$attributMagie = $classeMagicien->getProperty('magie');
?>
```

Récupérer tous les attributs

Si vous souhaitez récupérer tous les attributs d'une classe, il va falloir se servir de `ReflectionClass::getProperties()`. Le résultat retourné est un tableau contenant autant d'instance de `ReflectionProperty` que d'attributs.

Code : PHP

```
<?php
$classePersonnage = new ReflectionClass('Personnage');
$attributsPersonnage = $classeMagicien->getProperties();
?>
```

Après avoir vu comment récupérer un attribut, nous allons voir ce qu'on peut faire avec. 😊

Le nom et la valeur des attributs

Afin de récupérer le nom de l'attribut, nous avons toujours la méthode `ReflectionProperty::getName()`. Pour obtenir la valeur de celui-ci, nous utiliserons la méthode `ReflectionProperty::getValue($object)`. Nous devons spécifier à cette dernière méthode l'instance dans laquelle nous voulons obtenir la valeur de l'attribut : chaque attribut est propre à chaque instance, ça n'aurait pas de sens de demander la valeur de l'attribut d'une classe. 😞

Pour nous exercer, nous allons lister tous les attributs de la classe `Magicien`.

Code : PHP

```
<?php
$classeMagicien = new ReflectionClass('Magicien');
$magicien = new Magicien(Personnage::EXISTANT, 'Vykl2');

foreach ($classeMagicien->getProperties() as $attribut)
    echo $attribut->getName(), ' => ', $attribut->getValue($magicien);
?>
```

Il fonctionne pas ton code, j'ai une vieille erreur fatale qui me bloque tout...

En effet. Cette erreur fatale est levée car vous avez appelé la méthode `ReflectionProperty::getValue()` sur un attribut non public. Il faut donc rendre l'attribut accessible grâce à `ReflectionProperty::setAccessible($accessible)`, où `$accessible` vaut vrai ou faux suivant si vous voulez rendre l'attribut accessible ou non.

Code : PHP

```
<?php
$classeMagicien = new ReflectionClass('Magicien');
$magicien = new Magicien(Personnage::EXISTANT, 'Vykl2');

foreach ($classeMagicien->getProperties() as $attribut)
{
    $attribut->setAccessible(true);
    echo $attribut->getName(), ' => ', $attribut->getValue($magicien);
}

?>
```

Quand vous rendez un attribut accessible, vous pouvez modifier sa valeur grâce à `ReflectionProperty::setValue($objet, $valeur)`, ce qui est contre le principe d'encapsulation. Pensez donc bien à rendre l'attribut inaccessible après sa lecture en faisant `$attribut->setAccessible(false)`;

Portée de l'attribut

Il est tout à fait possible de savoir si un attribut est privé, protégé ou public grâce aux méthodes `ReflectionProperty::isPrivate()`, `ReflectionProperty::isProtected()` et `ReflectionProperty::isPublic()`.

Code : PHP

```
<?php
$uneClasse = new ReflectionClass('MaClasse');

foreach ($uneClasse->getProperties() as $attribut)
{
    echo $attribut->getName(), ' => attribut ';

    if ($attribut->isPublic())
        echo 'public';
    elseif ($attribut->isProtected())
        echo 'protégé';
    else
        echo 'privé';
}

?>
```

Il existe aussi une méthode permettant de savoir si l'attribut est statique ou non grâce à `ReflectionProperty::isStatic()`.

Code : PHP

```
<?php
$uneClasse = new ReflectionClass('MaClasse');

foreach ($uneClasse->getProperties() as $attribut)
{
    echo $attribut->getName(), ' => attribut ';

    if ($attribut->isPublic())
        echo 'public';
    elseif ($attribut->isProtected())
        echo 'protégé';
    else
        echo 'privé';

    if ($attribut->isStatic())
        echo ' (attribut statique)';
}

?>
```

Les attributs statiques

Le traitement d'attributs statiques diffère un peu dans le sens où ce n'est pas un attribut d'une instance mais un attribut de la classe. Ainsi, vous n'êtes pas obligé de spécifier d'instance lors de l'appel de `ReflectionProperty::getValue()` car un attribut statique n'appartient à aucune instance.

Code : PHP

```
<?php
class A
{
    public static $attr = 'Hello world !';
}

$classeA = new ReflectionClass('A');
echo $classeA->getProperty('attr')->getValue();
?>
```

Au lieu d'utiliser cette façon de faire, vous pouvez directement appeler `ReflectionClass::getStaticPropertyValue($attr)`, où `$attr` est le nom de l'attribut. Dans le même genre, on peut citer `ReflectionClass::setStaticPropertyValue($attr, $value)` où `$value` est la nouvelle valeur de l'attribut.

Code : PHP

```
<?php
class A
{
    public static $attr = 'Hello world !';
}

$classeA = new ReflectionClass('A');
echo $classeA->getStaticPropertyValue('attr'); // Affiche Hello world !
$classeA->setStaticPropertyValue('attr', 'Bonjour le monde !');
echo $classeA->getStaticPropertyValue('attr'); // Affiche Bonjour le monde !
?>
```

Vous avez aussi la possibilité d'obtenir tous les attributs statiques grâce à `ReflectionClass::getStaticProperties()`. Le tableau retourné ne contient pas des instances de `ReflectionProperty` mais uniquement les valeurs de chaque attribut.

Code : PHP

```
<?php
class A
{
    public static $attr1 = 'Hello world !';
    public static $attr2 = 'Bonjour le monde !';
}

$classeA = new ReflectionClass('A');

foreach ($classeA->getStaticProperties() as $attr)
    echo $attr;

// A l'écran s'affichera Hello world !Bonjour le monde !
?>
```

Obtenir des informations sur les méthodes de ses classes

Voici la dernière classe faisant partie de l'API de réflexivité que je vais vous présenter : il s'agit de `ReflectionMethod`. Comme vous l'aurez deviné, c'est grâce à celle-ci que l'on pourra obtenir des informations concernant telle ou telle méthode. Nous pourrions connaître la portée de la méthode (publique, protégée ou privée), si elle est statique ou non, abstraite ou finale, s'il s'agit du

constructeur ou du destructeur et on pourra même l'appeler sur un objet. 😊

Création d'une instance de ReflectionMethod

Instanciation directe

Le constructeur de ReflectionMethod demande 2 arguments : le nom de la classe et le nom de la méthode. Exemple :

Code : PHP

```
<?php
class A
{
    public function hello ($arg1, $arg2, $arg3 = 1, $arg4 = 'Hello world !')
    {
        echo 'Hello world !';
    }
}

$methode = new ReflectionMethod('A', 'hello');

?>
```

Récupération d'une méthode d'une classe

La seconde façon de faire est de récupérer la méthode de la classe grâce à ReflectionClass::getMethod(\$name). Celle-ci renvoie une instance de ReflectionClass représentant la méthode.

Code : PHP

```
<?php
class A
{
    public function hello ($arg1, $arg2, $arg3 = 1, $arg4 = 'Hello world !')
    {
        echo 'Hello world !';
    }
}

$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');

?>
```

Publique, protégée ou privée ?

Comme pour les attributs, nous avons des méthodes pour le savoir : j'ai nommé ReflectionMethod::isPublic(), ReflectionMethod::isProtected() et ReflectionMethod::isPrivate(). Je ne vais pas m'étendre sur le sujet, vous savez déjà vous en servir. 😊

Code : PHP

```
<?php
$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');

echo 'La méthode ', $methode->getName(), ' est ';

if ($methode->isPublic())
    echo 'publique';
elseif ($methode->isProtected())
    echo 'protégée';
else
    echo 'privée';

?>
```


Je suis sûr que vous savez quelle méthode permet de savoir si elle est statique ou non. 😊

Code : PHP

```
<?php
$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');

echo 'La méthode ', $methode->getName(), ' est ';

if ($methode->isPublic())
    echo 'publique';
elseif ($methode->isProtected())
    echo 'protégée';
else
    echo 'privée';

if ($methode->isStatic())
    echo ' (en plus elle est statique)';

?>
```

Abstraite ? Finale ?

Les méthodes permettant de savoir si une méthode est abstraite ou finale sont très simples à retenir : il s'agit de `ReflectionMethod::isAbstract()` et `ReflectionMethod::isFinal()`.

Code : PHP

```
<?php
$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');

echo 'La méthode ', $methode->getName(), ' est ';

if ($methode->isAbstract())
    echo 'abstraite';
elseif ($methode->isFinal())
    echo 'finale';
else
    echo '« normale »';

?>
```

Constructeur ? Destructeur ?

Dans le même genre de méthodes, on peut citer `ReflectionMethod::isConstructor()` et `ReflectionMethod::isDestructor()` qui permettent de savoir si la méthode est le constructeur ou le destructeur de la classe.

Code : PHP

```
<?php
$classeA = new ReflectionClass('A');
$methode = $classeA->getMethod('hello');

if ($methode->isConstructor())
    echo 'La méthode ', $methode->getName(), ' est le constructeur';
elseif ($methode->isDestructor())
    echo 'La méthode ', $methode->getName(), ' est le destructeur';

?>
```

Pour que la première condition renvoie vrai, il ne faut pas obligatoirement que la méthode soit nommée `__construct`. En effet, si la méthode a le même nom que la classe, celle-ci est considérée comme le constructeur de la classe car sous PHP 4 c'était comme ça

qu'on implémentait le constructeur : il n'y avait jamais de `__construct`. Pour que les scripts développés sous PHP 4 soient aussi compatibles sous PHP 5, le constructeur a aussi la possibilité d'être implémenté de cette façon, mais il est clairement préférable d'utiliser la méthode magique créée pour cet effet. 😊

Appeler la méthode sur un objet

Pour réaliser ce genre de chose, nous allons avoir besoin de `ReflectionMethod::invoke($object, $args)`. Le premier argument est l'objet sur lequel on veut appeler la méthode. Viennent ensuite tous les arguments que vous voulez passer à la méthode : vous devrez donc passer autant d'argument que la méthode appelée en exige. Prenons un exemple tout simple, vous comprendrez mieux :

Code : PHP

```
<?php
class A
{
    public function hello ($arg1, $arg2, $arg3 = 1, $arg4 = 'Hello world !')
    {
        var_dump ($arg1, $arg2, $arg3, $arg4);
    }
}

$a = new A;
$hello = new ReflectionMethod('A', 'hello');

$hello->invoke($a, 'test', 'autre test'); // On ne va passer que deux arguments à notre méthode

// A l'écran s'affichera donc :
// string(4) "test" string(10) "autre test" int(1) string(13) "Hello world !"
```

Une méthode semblable à `ReflectionMethod::invoke($object, $args)` existe : il s'agit de `ReflectionMethod::invokeArgs($object, $args)`. La différence entre ces deux méthodes est que la seconde demandera les arguments listés dans un tableau au lieu de les lister en paramètres. L'équivalent du code précédent avec `Reflection::invokeArgs()` serait donc le suivant :

Code : PHP

```
<?php
class A
{
    public function hello ($arg1, $arg2, $arg3 = 1, $arg4 = 'Hello world !')
    {
        var_dump ($arg1, $arg2, $arg3, $arg4);
    }
}

$a = new A;
$hello = new ReflectionMethod('A', 'hello');

$hello->invokeArgs($a, array ('test', 'autre test')); // Les deux arguments sont ceux de la méthode hello

// Le résultat affiché est exactement le même
```

Si vous n'avez pas accès à la méthode à cause de sa portée restreinte, vous pouvez la rendre accessible comme on l'a fait avec les attributs, grâce à la méthode `ReflectionMethod::setAccessible($bool)`. Si `$bool` vaut `true`, alors la méthode sera accessible, sinon elle ne le sera pas. Je me passe d'exemple, je suis sûr que vous trouverez tous seuls. 😊

Allez, un petit QCM pour être sûr que vous avez un minimum suivi et c'est fini !

Je ne pense pas que vous vous servirez de ce chapitre dans l'immédiat, mais il est toujours bon d'avoir entendu parler de l'API de réflexivité en PHP si l'on décide de programmer OO. 😊

Je vous ai aussi volontairement caché une classe : la classe [ReflectionParameter](#) qui vous permet d'obtenir des informations sur les

paramètres de vos méthodes. Je vous laisse vous documenter à ce sujet, son utilisation est très simple. 😊

Voici cette deuxième partie terminée. Je suis heureux de vous annoncer que vous avez toutes les connaissances requises pour faire ce que vous voulez ! 😊

Aucune syntaxe ou notion n'a été laissée de côté, je vous ai parlé de tout (ou du moins, presque tout). Cependant, il vous reste quelques choses à voir en relation avec la POO. Rendez-vous en troisième partie. 😊

Partie 3 : Bonnes pratiques

Cette troisième partie a pour but de vous aider à vous organiser et vous présenter de bonnes pratiques. Vous n'apprendrez donc pas de nouvelles notions ou de nouvelle syntaxe en PHP : vous allez apprendre à organiser votre code. Vous découvrirez quelques trucs et astuces et enfin, après ça, la super récompense : un TP qui aura pour objectif de créer... un système de news orienté objet ! Eh oui, je pense que maintenant, vous êtes fin prêts. 😊

Alors, on y va ? 😊

UML : présentation (1/2)

Commençons en douceur cette partie en vous présentant l'UML. L'UML est un langage de modélisation objet. Non, ne vous inquiétez pas, ce n'est pas si compliqué que ça. La chose la plus compliquée à comprendre c'est à quoi ça sert (question qui, je pense, a déjà traversé votre esprit au tout début de ce tutoriel sur la POO, non ? 😊). Le but de ces deux chapitres est de vous aider à penser objet en modélisant votre application. Let's go. 😊

UML, kézako ?

L'UML est un langage de modélisation objet. Il permet donc de modéliser vos objets et ainsi représenter votre application sous forme de diagramme.

Avant d'aller plus loin, je vais quand même vous dire ce que signifie UML :

Unified Modeling Language

Ce qui veut dire « langage de modélisation unifié ». UML n'est pas un langage à proprement parler, plutôt une sorte de méthodologie.

Mais qu'est-ce que c'est concrètement ? À quoi ça pourra bien me servir ?

Grâce à UML, vous pourrez modéliser toute votre application. Quand je dis toute, j'entends par là la plupart de votre application car PHP n'est pas un langage orienté objet : le modèle objet lui a été implémenté au cours des versions (dès la version 4). Grâce à ces diagrammes, vous pourrez donc représenter votre application : son fonctionnement, sa mise en route, les actions susceptibles d'être effectuées par l'application, etc. Ces diagrammes ont été conçus pour que quelqu'un n'ayant aucune connaissance en informatique puisse comprendre le fonctionnement de votre application. Certes, certains diagrammes sont réservés aux développeurs car assez difficiles à expliquer si on ne sait pas programmer : ce sont ces diagrammes que l'on va étudier.

Ok, je peux modéliser mon application mais en quoi cela va-t-il m'aider ?

Si vous avez un gros projet (et là je ne parle pas forcément de PHP mais de tout langage implémentant la POO), il peut être utile de le modéliser afin d'y voir plus clair. Si vous vous focalisez sur la question « Par où commencer ? » au début du projet, c'est qu'il vous faut un regard plus objectif sur le projet. Les diagrammes vous apporteront ce regard différent sur votre application.

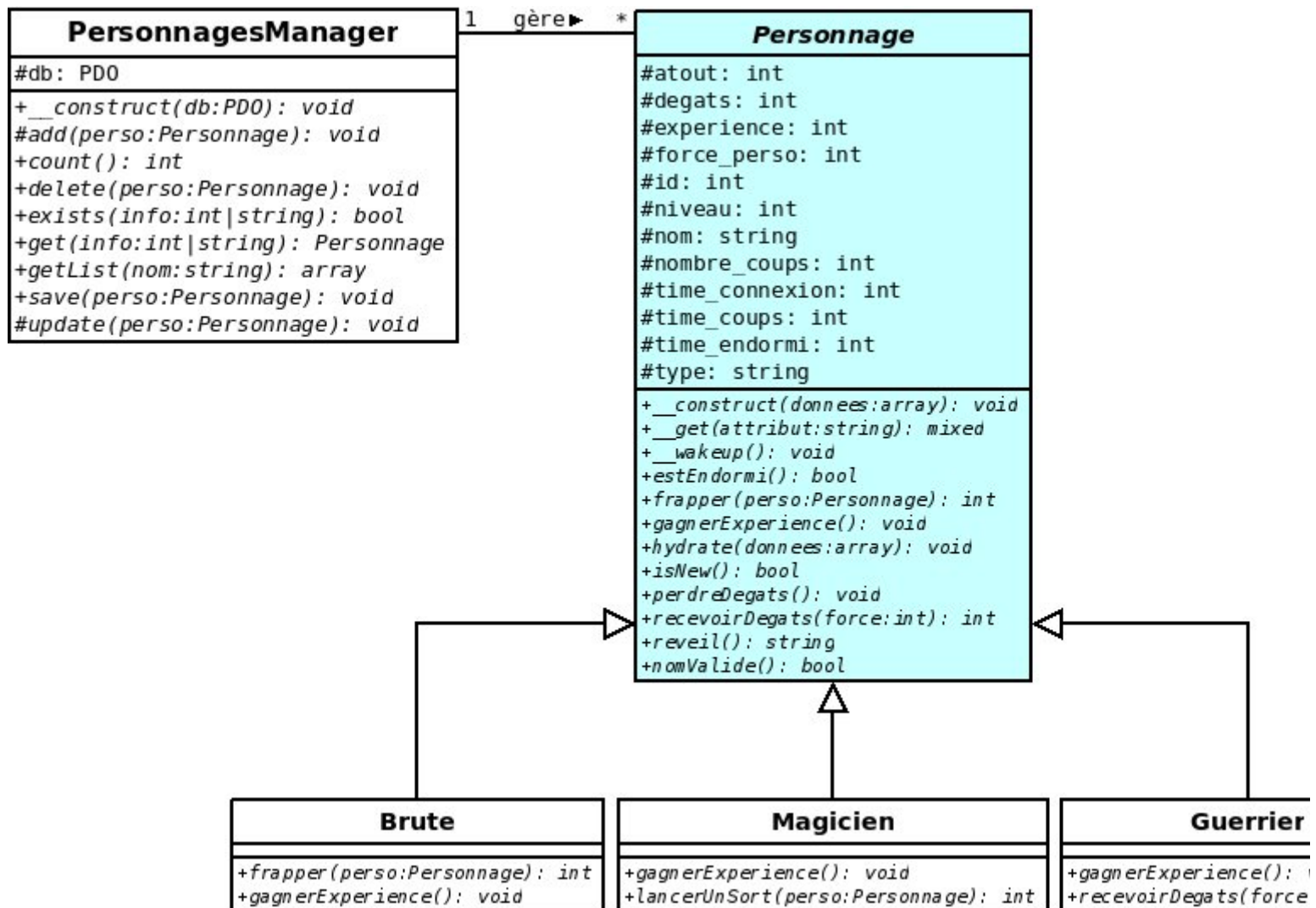
L'UML peut aussi être utile quand on commence à programmer OO mais qu'on ne sait pas trop comment s'y prendre. Par exemple,

vous, vous ne savez pas trop encore comment programmer orienté objet de façon concrète (vous avez vu comment créer un mini-jeu, c'est cool, mais un livre d'or ou un système de news, vous savez ?). Sachez donc que l'UML va vous être d'une grande aide au début. Ça vous aidera à mieux penser objet car les diagrammes représentant vos classes, vous vous rendrez mieux compte de ce qu'il est intelligent de faire ou de ne pas faire. Cependant, l'UML n'est pas un remède miracle et vous ne saurez toujours pas comment vous y prendre pour réaliser votre toute première application, mais une fois que je vous aurai montré et que vous saurez un minimum comment faire, là, l'UML pourra sans doutes vous aider.

Enfin, l'UML a aussi un rôle de documentation. En effet, quand vous représenterez votre projet sous forme de diagramme, quiconque sachant le déchiffrer pourra (du moins, si vous l'avez bien construit) comprendre le déroulement de votre application et éventuellement reprendre votre projet (c'est toujours mieux que la [phpdoc](#) que nous utilisons jusqu'à présent pour commenter nos classes (regardez le dernier TP si vous avez un trou de mémoire)).

Bref, que ce soient pour les gros projets personnels ou professionnels, l'UML vous suivra partout. Cependant, tout le monde ne trouve pas l'UML très utile et certains préfèrent modéliser le projet « dans leur tête ». Vous verrez avec le temps si vous avez réellement besoin de l'UML ou si vous pouvez vous en passer, mais pour l'instant, je vous conseille de modéliser (enfin, dès le prochain chapitre). 😊

Il existe plusieurs types de diagrammes. Nous, nous allons étudier les diagrammes de classe. Ce sont des diagrammes modélisant vos classes et montrant les différentes interactions entre elles. Un exemple ?



Vous aurez sans doutes reconnu notre dernier TP représenté sur un diagramme.

Ce diagramme a l'avantage d'être très simple à étudier. Cependant, il y a des cas complexes et des conventions à connaître car, par exemple, des méthodes abstraites n'ont pas le même style d'écriture qu'une méthode finale. On va étudier les bases, petit à petit, puis à la fin de ce chapitre vous serez capable d'analyser complètement un diagramme de classe. 😊

Modéliser une classe

Première approche

On va commencer en douceur par analyser une simple classe modélisée. On va prendre notre classe News simplifiée.

News
#id: int
+__construct(donnees:array): void +hydrate(donnees:array): void

Nous avons donc ici une simple classe. Commençons par analyser celle-ci.

- En haut, en gras et gros, c'est le nom de la classe (ici, News) ;
- Ensuite, séparé du nom de la classe, vient un attribut de cette classe : il s'agit de id précédé d'un #, nous verrons ce dont il s'agit juste après ;
- Enfin, séparée de la liste d'attributs, vient la liste des méthodes de la classe. Toutes sont précédées d'un +, qui a, comme le #, une signification.

On va commencer par analyser la signification du # et des +. Je ne vous fais pas attendre : ces signes symbolisent la portée de l'attribut ou de la méthode. Voici la liste des trois symboles :

- Le signe + : l'élément suivi de ce signe est public ;
- Le signe # : l'élément suivi de ce signe est protégé ;
- Le signe - : l'élément suivi de ce signe est privé.

Maintenant que vous savez à quoi correspondent ces signes, regardez à droite de l'attribut : suivi de deux points, on peut lire int. Cela veut dire que cet attribut est de type int, tout simplement. int est le diminutif de integer qui veut dire entier : notre attribut est donc un nombre entier.

Vous voyez qu'à droite des méthodes on peut apercevoir la même chose. Ceci indique le type de la valeur de retour de celle-ci. Si elle ne renvoie rien, alors on dit qu'elle est vide (= void). Si elle peut renvoyer plusieurs types de résultats différents, alors on dit qu'elle est mixte (= mixed). Par exemple, une méthode __get peut renvoyer une chaîne de caractères ou un entier (regardez la classe Personnage sur le premier diagramme).

La dernière chose sur ce diagramme à expliquer, c'est ce qu'il y a entre les parenthèses suivant le nom des méthodes. Comme vous vous en doutez, elles contiennent tous les attributs ainsi que leur type (entier, tableau, chaîne de caractères, etc.). Si un paramètre peut être de plusieurs types, alors, comme pour les valeurs de retour des méthodes, on dit qu'il est mixte.

Si un attribut, une méthode ou un paramètre est une instance ou en renvoie une, il ne faut pas dire que son type est object. Son type est le nom de la classe, à savoir Personnage, Brute, Magicien, Guerrier, etc.

Ensuite, il y a des conventions concernant le style d'écriture des attributs et méthodes :

- Si la méthode est en gras, alors elle est abstraite. Si elle n'est pas en italique, alors elle est finale. Si elle n'est pas en gras et est en italique, alors elle est « normale » (ni abstraite, ni finale quoi 🤖) ;
- Si l'attribut ou la méthode est souligné, alors l'élément est statique.

Et enfin, une dernière chose, si le nom de la classe est en italique, alors elle est abstraite. 🤖

Tous ces « codes » (textes en gras, soulignés ou en italiques) sont un choix personnel (choix largement influencé par la configuration par défaut de mon logiciel de création de diagramme, je dois l'avouer 🤖).

Exercices

Maintenant que je vous ai dit tout ça, je vais vous donner quelques attributs et méthodes et vous allez essayer de deviner quels sont ses particularités (portée, statique ou non, abstraite, finale, ou ni l'un ni l'autre, les arguments et leur type...). Je commence avec ceci :

Citation : Méthode

```
-maMethode (param1:int) : void
```

Correction :

Secret ([cliquez pour afficher](#))

Citation : Méthode

```
#maMethode (param1:mixed) : array
```

Correction :

Secret ([cliquez pour afficher](#))

Les différents codes (gras, italique, soulignements, etc.) rentreront dans votre tête au fil du temps, je ne vous impose pas de tous les apprendre sur-le-champs. Référez-vous autant que nécessaire à cette partie de ce chapitre en cas de petits trous de mémoire.



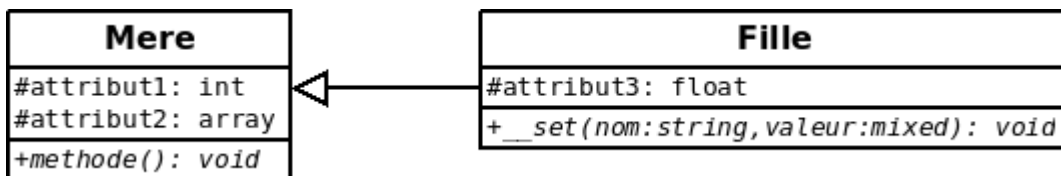
Maintenant que vous savez analyser un objet, il serait bien de savoir analyser les interactions entre ceux-ci, non ? 😊

Modéliser les interactions

Nous attaquons la dernière partie de ce chapitre. Nous allons voir comment modéliser les interactions entre les objets que l'on a modélisés. Jusqu'à présent, on sait comment les reconnaître, mais l'avantage de la POO est de créer un ensemble d'objets qui interagissent entre eux afin de créer une vraie application.

L'héritage

Parmi les interactions, on peut citer l'héritage. Comme montré dans le premier diagramme que vous avez vu, on symbolise l'héritage par une simple flèche, comme ceci :



Ce diagramme équivaut au code suivant :

Code : PHP

```

<?php
class Mere
{
    protected $attribut1, $attribut2;

    public function methode1()
    {

    }
}

class Fille extends Mere
{
    protected $attribut3;

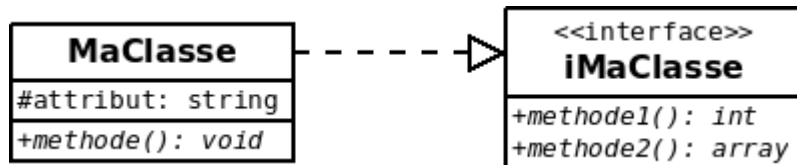
    public function __set ($nom, $valeur)
    {

    }
}
?>

```

Les interfaces

Vous connaissez aussi les interactions avec les interfaces. En effet, comme je l'ai déjà dit, une interface n'est rien d'autre qu'une classe entièrement abstraite, elle est donc considérée comme tel. Si une classe doit implémenter une interface, alors on utilisera la flèche en pointillés, comme ceci :



Traduit en PHP, ça donne :

Code : PHP

```

<?php
interface iMaClasse
{
    public function methode1();
    public function methode2();
}

class MaClasse implements iMaClasse
{
    protected $attribut;

    public function methode()
    {

    }

    // Ne pas oublier d'implémenter les méthodes de l'interface !

    public function methode1()
    {

    }

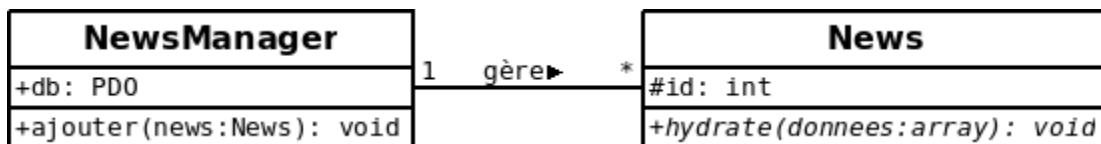
    public function methode2()
    {

    }
}
?>

```

L'association

Nous allons voir encore trois interactions qui se ressemblent. La première est l'association. On dit que deux classes sont associées lorsqu'une instance des deux classes est amenée à interagir avec l'autre instance. L'association entre deux classes est modélisée comme ceci :



L'association est ici caractérisée par le fait qu'une méthode de la classe NewsManager entre en relation avec une instance de la classe News.

Attends deux secondes... Je vois des trucs écrits sur la ligne ainsi qu'aux extrémités, qu'est-ce que c'est ?

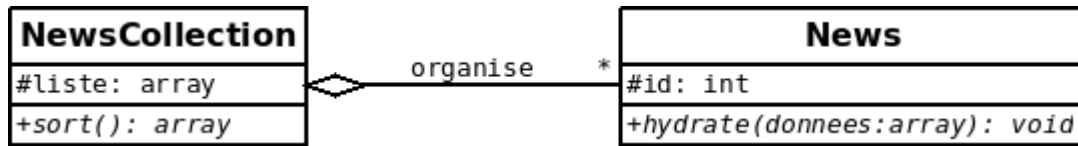
Le mot écrit au centre, au-dessus de la ligne est la définition de la relation. Il est suivi d'un petit symbole indiquant le sens de l'association. Ainsi, on peut lire facilement « NewsManager gère News ».

Ensuite, vous voyez le chiffre 1 écrit à gauche et une astérisque à droite. Ce sont les cardinalités. Ces cardinalités présentent le nombre d'instances qui participent à l'interaction. Nous voyons donc qu'il y a 1 manager pour une infinité de news. On peut désormais lire facilement « 1 NewsManager gère une infinité de News ». Les cardinalités peuvent être écrites sous différentes formes :

- x (nombre entier) : tout simplement la valeur exacte de x ;
- x..y : de x à y (exemple : 1..5) ;
- * : une infinité ;
- x..* : x ou plus (exemple : 5..*).

L'agrégation

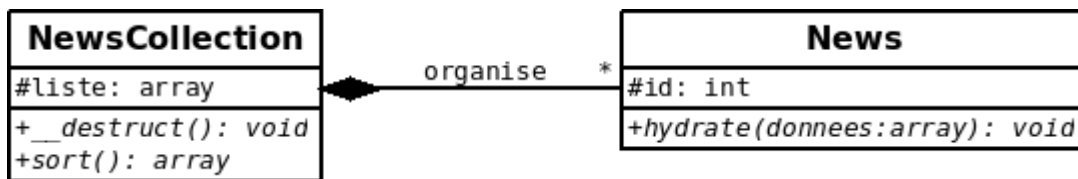
La deuxième flèche que je vais vous présenter est l'agrégation. Il s'agit d'une forme d'association un peu particulière : on parlera d'agrégation entre deux classes lorsque l'une d'entre elles contiendra au moins une instance de l'autre classe. Une agrégation est caractérisée de la sorte :



Vous pouvez remarquer qu'il n'y a pas de cardinalité du côté du losange. En effet, le côté ayant le losange signifie qu'il y a obligatoirement une et une seule instance de la classe par relation (ici la classe est NewsCollection).

La composition

La dernière interaction que je vais vous présenter est la composition. La composition est une agrégation particulière. Imaginons que nous avons une classe A qui contient une ou plusieurs instance(s) de B. On parlera de composition si, lorsque l'instance de A sera supprimée, toutes les instances de B contenues dans l'instance de A seront elles aussi supprimées (ce qui n'était pas le cas avec l'agrégation). Une composition est représentée de la sorte :



Notez la présence de la méthode `__destruct()` qui sera chargée de détruire les instances de la classe News. Cependant, celle-ci n'est pas obligatoire : vous pouvez très bien, dans l'une de vos méthodes, placer un `$this->liste[] = new News;`, et l'instance créée puis stockée dans l'attribut `$liste` sera donc automatiquement détruit. 😊

Je ne vous ai pas préparé de QCM car aucune notion essentielle n'est exprimée ici, et le niveau de ce chapitre n'est pas très haut. Il suffit juste de se rappeler de ce qu'est l'UML, savoir modéliser un objet, à quoi ça sert, etc.

Vous savez maintenant ce qu'est l'UML et savez déchiffrer un diagramme de classe. Que diriez-vous maintenant de les créer vous-même ? 😊

Ce chapitre ne se veut pas complet pour la simple et bonne raison qu'il serait beaucoup trop long de faire le tour de l'UML. En effet, un cours complet sur l'UML s'étalerait sur beaucoup de chapitres, et ce n'est clairement pas le but du tutoriel. Si ce sujet vous intéresse, je peux vous renvoyer sur le site uml.free.fr.

UML : modélisons nos classes (2/2)

Voici la suite directe du précédent chapitre ayant pour objectif de vous introduire à l'UML. Nous savons actuellement analyser des diagrammes de classes et les interactions entre elles, mais que diriez-vous de les créer vous-même ces diagrammes ? 😊

Pour pouvoir créer ces diagrammes, nous allons avoir besoin d'un programme : je vais vous présenter [Dia](#).

Ayons les bons outils

Avant de se lancer tête baissée dans la modélisation, il serait bien d'avoir les logiciels qui le permettront (ça pourrait aider, sait-on jamais 😊). Pour cela, nous allons avoir besoin d'un logiciel de modélisation de diagrammes UML. Il y en a plusieurs, et pour ma part j'ai choisi Dia.

La raison de ce choix est simple : une extension qui permet de convertir les diagrammes UML en code PHP a été développée pour ce logiciel ! Ainsi, en quelques clics, toutes vos classes contenant les attributs et méthodes ainsi que les interactions seront générées, le tout commenté avec une phpdoc. Bref, ça nous simplifiera grandement la tâche ! 😊

Installation

Téléchargement et installation sous Windows

Si vous êtes sous Windows, la marche à suivre est très simple. Il vous suffit d'aller sur le site dia-installer.de et de télécharger Dia v0.97. Une fois téléchargé, installez-le comme n'importe quel autre logiciel. Facile, non ? 😊

Téléchargement et installation sous Linux

L'installation sous Linux se fait en compilant les sources du logiciel. Pour cela, téléchargez l'archive disponible [sur le site du projet Dia](#). Téléchargez la version 0.97 ou supérieure si elle est sortie (ce tutoriel est écrit sous la version 0.97). Une fois l'archive téléchargée, décompressez-la soit en faisant clic droit sur l'archive afin de sélectionner l'option Extraire ici, soit en ligne de commande (`tar xjf dia-0.97.tar.bz2`).

Ensuite la compilation se fait en trois étapes qui sont chacune effectuées en tapant une commande, à savoir `./configure` , `make` et `make install` .

Téléchargement et installation sous Mac OS

Sous Mac OS, vous aurez besoin de [MacPorts](#). Pour le télécharger, rendez-vous sur [le page d'installation du site de MacPorts](#). Sachez aussi qu'il vous faut la version 3.1 (ou supérieure) de Xcode, [téléchargeable ici](#).

Une fois tout ceci terminé, ouvrez votre terminal (application située dans le dossier Utilitaires dans Applications) puis mettez à jour la liste des paquets avec la commande `sudo port selfupdate` . Ensuite, pour installer Dia, exécutez la commande `sudo port install dia` . Enfin, pour lancer Dia plus facilement, créez un alias menant à l'application dans le dossier `/usr/bin/` grâce à la commande `sudo ln -s /opt/local/bin/dia /usr/bin/dia` .

Installation de l'extension uml2php5

Vous avez correctement installé Dia sur votre ordinateur. Il faut maintenant installer l'extension uml2php5 qui nous permettra de générer automatiquement le code de nos classes à partir de nos diagrammes. Pour cela, il vous faut télécharger 5 fichiers. Ces fichiers sont disponibles sur [le site de l'extension](#), en ayant cliqué sur Download à gauche. Téléchargez l'archive .zip ou .tar.gz suivant vos préférences. Décompressez cette archive et copiez / collez les 5 fichiers dans le dossier xslt présent dans le répertoire d'installation de Dia.

Lancer Dia

Histoire qu'on soit tous au point, je vais vous demander de lancer votre logiciel afin qu'on soit sûr qu'on a bien tous le même. Si l'interface est différente, veuillez bien à avoir téléchargé la version 0.97 du logiciel (actuellement la dernière). Si une version plus récente est sortie, vous pouvez me contacter. 😊



[Cliquez pour agrandir](#)

Deux zones principales

L'interface étant assez intuitive, je ne vais pas m'étaler sur la présentation de tous les recoins de celle-ci. Je vais me contenter de vous donner le rôle des deux parties de cette interface.

La première partie est celle de gauche. Elle contient tous les outils (créer une classe, une interaction, ou bien un trait, un cercle, etc.). La seconde est beaucoup plus grande : il s'agit de la zone de droite qui contiendra notre diagramme. C'est à l'intérieur d'elle que l'on effectuera nos opérations pour tout mettre en œuvre.

Unir les fenêtres

Il se peut que vous ayez deux fenêtres séparées. Si c'est le cas, c'est que vous n'avez pas passé l'argument `--integrated` au programme. Vous pouvez passer cet argument en lançant le logiciel en ligne de commande. Il y a cependant plus pratique, voici

donc des petites astuces suivant votre système d'exploitation.

Sous Windows

Sous Windows, vous pouvez créer un raccourci. Pour cela, allez dans le dossier d'installation de Dia puis dans le dossier bin. Faites un clic droit sur le fichier diaw.exe et cliquez sur Copier. Allez dans le répertoire où vous voulez créer le raccourci, cliquez droit dans ce répertoire puis choisissez l'option Coller le raccourci. Faites un clic droit sur le fichier créé, puis cliquez sur Propriétés. Dans le champ texte Cible, rajoutez à la fin (en dehors des guillemets) --integrated. Ainsi, quand vous lancerez Dia depuis ce raccourci, les deux fenêtres seront fusionnées.

Par défaut, le raccourci présent dans le menu démarrer lance Dia avec l'option --intergrated. Si ce n'est pas le cas, modifiez la cible de celui-ci comme on vient de le faire (Clic droit > Propriétés > ...).

Sous Linux

Je vais ici vous donner une manipulation simple sous Ubuntu afin de modifier la cible du lien pointant sur le logiciel dans le menu Applications. Dans le menu Système, allez dans Préférences puis Menu principal. À gauche de la fenêtre qui est apparue, sous le menu Applications, cliquez sur Graphisme. Au milieu de la fenêtre, cliquez sur Éditeur de diagrammes Dia. À droite, cliquez sur Propriétés. Dans le champ texte Commande, placez-y `dia --integrated %F`. Fermez le tout, relancez votre programme via le menu et vous avez vos deux fenêtres fusionnées.

Sous les autres distributions, l'idée est la même : il faut vous arranger pour modifier le raccourci afin d'ajouter l'option --integrated.

Sous Mac OS

Hélas sous Mac OS, vous ne pouvez créer de raccourci. La solution consiste donc à lancer le logiciel en ligne de commande. Heureusement, celle-ci est assez courte et simple à retenir. En effet, un simple `dia --integrated` suffit. 🤖

Cependant, vous pouvez télécharger [un lanceur](#) qui vous facilitera la tâche pour lancer l'application.

Modéliser une classe

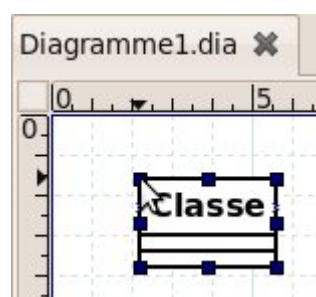
Créer une classe

Lançons-nous maintenant dans la création d'un diagramme. Pour commencer, modélisons notre première classe. Pour cela, rien de plus simple. Je vais vous demander de cliquer sur cet icône :



Cet icône, comme l'infobulle nous l'indique, représente l'outil permettant de créer une classe. Ainsi, dans la zone contenant notre diagramme, il suffira de cliquer n'importe où pour qu'une classe soit créée à l'endroit où on a cliqué. Essayez donc. 🤖

Normalement, vous devriez avoir obtenu quelque chose ressemblant à ceci :



Vous voyez 8 carrés ainsi que 2 croix bleues autour de la classe. Si les carrés sont présents, ça veut dire que la classe est sélectionnée (par défaut sélectionnée quand vous en créez une). Si vous voulez enlever cette sélection, il vous suffit de cliquer à côté. Vous verrez ainsi les 8 carrés remplacés par de nouvelles petites croix bleues qui ont une signification bien particulière. Je vous expliquerai ceci plus tard. 😊

Modifier notre classe

Nous allons accéder aux propriétés de notre classe afin de pouvoir la modifier. Pour cela, je vais vous demander de double-cliquer dessus afin d'obtenir cette fenêtre :



[Cliquez pour agrandir](#)

Décortiquons un peu cette grosse fenêtre. Tout ceci est en fait très simple, il suffit juste de quelques explications.

Tout en haut, vous voyez une liste de 5 onglets :

- Classe : permet de gérer les options concernant cette classe ;
- Attributs : permet de gérer la liste des attributs de cette classe ;
- Opérations : permet de gérer la liste des méthodes de cette classe ;
- Modèles : inutile en PHP étant donné qu'il n'y a pas de templates. Si ça vous intrigue, allez voir en C++ par exemple 😊 ;
- Style : permet de modifier le style de la classe (couleurs et effets de texte).

Gestion des options de la classe

En haut, vous pouvez apercevoir 3 champs texte. Le premier est le nom de la classe. Vous pouvez par exemple y indiquer le nom Personnage. Ensuite vient le stéréotype. Ce champ est à utiliser pour spécifier que la classe est particulière. Par exemple, s'il s'agit d'une interface, on y écrira simplement « interface ». Enfin, dans le dernier champ texte, nous y placerons des commentaires relatifs à la classe.

Ensuite viennent une série de cases à cocher. La première signifie (comme vous vous en doutez) si la classe est abstraite ou non. Par la suite nous pouvons apercevoir 3 cases à cocher permettant d'afficher ou non des éléments (afficher ou masquer les attributs, méthodes ou commentaires). Si vous décidez de masquer les attributs ou méthodes, leur bloc disparaîtra de la classe (les bordures entourant le bloc comprises), tandis que si vous décidez de supprimer les attributs ou méthodes (via les cases à cocher de droite), le bloc restera visible mais ne contiendra plus leur liste d'attributs ou méthodes. Le masquage des attributs est utile dans le cas où la classe est une interface par exemple. 😊

Pour que vous vous rendiez compte des modifications que vous effectuez en temps réel, cliquez sur Appliquer. Vous verrez ainsi votre classe se modifier avec les nouvelles options. Si ça ne vous plaît pas et que vous voulez annuler tout ce que vous avez fait, il vous suffit de cliquer sur Fermer. Par contre, si vous voulez enregistrer vos informations, cliquez sur Valider.

Gestion des attributs

Commençons par ajouter des attributs à notre classe. Pour cela, cliquez sur l'onglet Attributs :



[Cliquez pour agrandir](#)

Analysons le contenu de cette fenêtre. La partie la plus évidente que vous voyez est le gros carré blanc en haut à gauche : c'est à l'intérieur de ce bloc que seront listés tous les attributs. Vous pourrez à partir de là en sélectionner afin d'en modifier, d'en supprimer, ou d'en monter ou descendre d'un cran dans la liste. Afin d'effectuer ces actions, il vous suffira d'utiliser les boutons situés à droite de ce bloc :

- Le bouton Nouveau créera un nouvel attribut ;
- Le bouton Supprimer supprimera l'attribut sélectionné ;
- Le bouton Monter montera l'attribut d'un cran dans la liste (s'il n'est pas déjà tout en haut) ;
- Le bouton Descendre descendra l'attribut d'un cran dans la liste (s'il n'est pas déjà tout en bas).

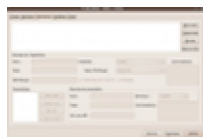
Créez donc un nouvel attribut en cliquant sur le bouton adapté. Vous voyez maintenant tous les champs du bas qui se dégrisent. Regardons de plus près ce dont il s'agit.

- Un premier champ texte est présent afin d'y spécifier le nom de l'attribut (par exemple, vous pouvez y inscrire force) ;
- En dessous est présent un champ texte demandant le type de l'attribut (s'il s'agit d'une chaîne de caractères, d'un nombre entier, d'un tableau, d'un objet, etc.). Par exemple, pour l'attribut créé, vous pouvez entrer int ;
- Ensuite vient un champ texte demandant la valeur de l'attribut. Ce n'est pas obligatoire, il faut juste y spécifier quelque chose quand on souhaite que notre attribut ait une valeur par défaut (par exemple, vous pouvez entrer 0) ;
- Le dernier champ texte est utile si vous souhaitez laisser un commentaire sur l'attribut (ce que je vous conseille de faire) ;
- Vient ensuite une liste déroulante permettant d'indiquer la visibilité de l'attribut. Il y a 4 options. Vous reconnaîtrez parmi elles les 3 types de visibilité. En plus de ces 3 types, vous pouvez apercevoir un quatrième type nommé Implémentation. Ne vous en préoccupez pas, vous n'en aurez pas besoin. 😊
- La dernière option est une case à cocher, suivie de Visibilité de la classe. Si vous cochez cette case, ceci voudra dire que votre attribut sera statique.

Entraînez-vous à créer quelques attributs bidons (ou bien ceux de la classe Personnage si vous êtes à cours d'idée). Vous pourrez ainsi tester les 4 boutons situés à droite du bloc listant les attributs afin de bien comprendre leur utilisation (bien que ceci ne doit pas être bien difficile, mais sait-on jamais 😊).

Gestion des méthodes

Après avoir ajouté des attributs à notre classe, voyons comment lui ajouter des méthodes. Pour cela, cliquez sur l'onglet Opérations :



[Cliquez pour agrandir](#)

Il y a déjà un peu plus de bazarre. 😊

Cependant, si vous observez bien et que vous avez bien suivi la précédente partie sur la gestion des attributs, vous ne devriez pas être entièrement dépaysé. En effet, en haut, nous avons toujours notre bloc blanc qui listera, cette fois-ci, nos méthodes, ainsi que les 4 boutons à droite effectuant les mêmes opérations. Créez donc une nouvelle méthode en cliquant simplement sur le bouton Nouveau, comme pour les attributs, et bidouillons-la un peu. 😊

Pour commencer, je vais parler de cette partie de la fenêtre :



[Cliquez pour agrandir](#)

La partie encadrée de rouge concerne la méthode en elle-même. Les champs qui restent concernent ses paramètres, mais on verra ça juste après.

Les deux premiers champs textes (Nom et Type) ont le même rôle que pour les attributs (je vous rappelle que le type de la méthode est le type de la valeur renvoyée par celle-ci). Ensuite est présent un champ texte Stéréotype. Inutile de vous encombrer l'esprit avec ça, ça ne vous sera pas utile. Nous avons, comme pour les attributs, deux autres options qui refont leur apparition : la visibilité et la case à cocher Visibilité de la classe. Inutile de vous rappeler ce que c'est je pense (si toutefois vous avez un trou de mémoire, il vous suffit de remonter un tout petit peu). Et pour finir avec les points communs avec les attributs, vous pouvez apercevoir tout à droite le champ texte Commentaire qui a aussi pour rôle de spécifier les commentaires relatifs à la méthode.

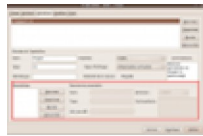
Par contre, contrairement aux attributs, nous avons ici deux nouvelles options. La première est Type d'héritage qui est une liste déroulante présentant 3 options :

- Abstraite : à sélectionner si la méthode est abstraite ;
- Polymorphe (virtuelle) : à sélectionner si la méthode n'est ni abstraite, ni finale ;
- Feuille (finale) : à sélectionner si la méthode est finale.

La deuxième option est une case à cocher. À côté de celle-ci est écrit Requête. Vous vous demandez sans doute ce qu'une requête vient faire ici, non ? En fait, si vous cochez cette case, cela veut dire que la méthode ne modifiera aucun attribut de la classe. Par exemple, vos accesseurs (les méthodes étant chargées de renvoyer la valeur d'attributs privés ou protégés) sont considérés comme de simples requêtes afin d'accéder à ces valeurs. Elles pourront donc avoir cette case de cochée.

Entraînez-vous à créer des méthodes, c'est toujours bien de pratiquer. 😊

Maintenant que vous êtes prêts (enfin j'espère), nous allons leur ajouter des arguments. Pour cela, nous allons nous intéresser à cette partie de la fenêtre :



[Cliquez pour agrandir](#)

Encore une fois, nous remarquons des ressemblances avec tout ce que nous avons vu. En effet, nous retrouvons notre bloc blanc ainsi que ses 4 boutons, toujours fidèles au rendez-vous. Bref, inutile de vous expliquer le rôle de chacun hein 😊. Au cas où vous n'avez pas deviné, c'est ici que se fait la gestion des paramètres de la méthode. 😊

Créez donc un nouveau paramètre. Les champs de droite, comme à leur habitude, se dégrisent. Je fais une brève description des fonctionnalités des champs, étant donné que c'est du déjà vu.

- Le champ Nom indique le nom de l'argument ;
- Le champ Type spécifie le type de l'argument (entier, chaîne de caractères, tableau, etc.) ;
- Le champ Val. par déf. révèle la valeur par défaut de l'argument ;
- Le champ Commentaire permet de laisser un petit commentaire concernant l'argument ;
- L'option Direction est inutile.

Gestion du style de la classe

Si vous êtes sous Windows, vous avez peut-être remarqué que tous vos attributs et méthodes ont le même style, quels que soient leurs spécificités. Pour remédier à ce problème, il faut modifier le style de la classe en cliquant sur l'onglet Style.



[Cliquez pour agrandir](#)

Le premier genre, nommé Normal, ne veut pas dire qu'il s'agit du style de chaque méthode normale, mais de chaque méthode finale ! Le style des méthodes normales est à modifier sur la deuxième ligne, ce qui correspond au genre Polymorphe.

Mis à part cette petite alerte, je ne pense pas que de plus amples explications s'imposent. Je ne ferais que répéter ce que la fenêtre vous affiche. 😊

Pour modifier le style de toutes les classes, sélectionnez-les toutes (en cliquant sur chacune tout en maintenant la touche Shift) puis double-cliquez sur l'une d'elle. Modifiez le style dans la fenêtre ouverte, validez et admirez. 😊

Je crois avoir fait le tour de toutes les fonctionnalités. Pour vous entraîner, vous pouvez essayer de reconstituer la classe

Modéliser les interactions

Vous avez réussi à créer une belle classe avec tout plein d'attributs et de méthodes. Je vais vous demander d'en créer une autre afin de les faire interagir entre elles. Nous allons voir les 4 interactions qu'on a vues dans le précédent chapitre.

Création des liaisons

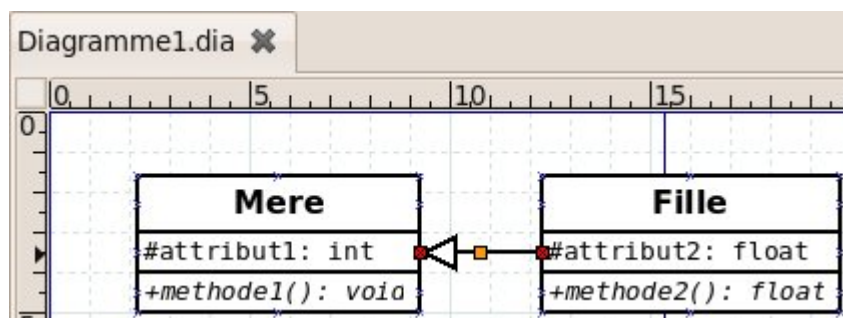
L'héritage

Nous allons créer notre première interaction : l'héritage entre deux classes. Pour cela, il faut cliquer sur l'icône représentant l'interaction de l'héritage, à savoir :



Ensuite, dirigez-vous sur la partie de droite contenant vos deux classes. Cliquez sur une croix bleue de la classe mère et, tout en maintenant le clic de la souris enfoncé, glissez jusqu'à une croix bleue de votre classe fille. Lorsque vous déplacez votre flèche, le curseur prend la forme d'une croix. Une fois qu'il survole une croix bleue, il est transformé en une autre croix représentant trois chemins qui se croisent, et la classe reliée se voit entourée d'un épais trait rouge : c'est à ce moment-là que vous pouvez lâcher le clic.

Normalement, vous devriez avoir obtenu ceci :



Notez les deux points rouges sur la flèche ainsi que le point orange au milieu. Ceux-ci indiquent que la flèche est sélectionnée. Il se peut que l'un des points des extrémités de la flèche soit vert, auquel cas cela signifie que ce point n'est pas situé sur une croix bleue ! Si elle n'est pas sur une croix bleue, alors elle n'est reliée à aucune classe.

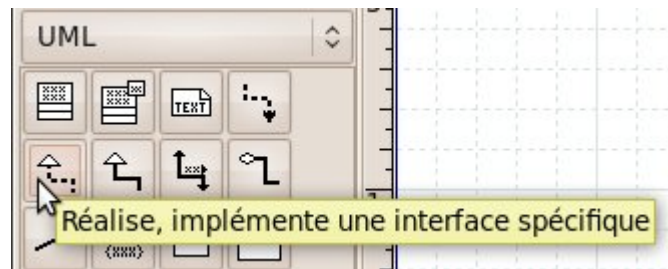
Le point orange du milieu ne bougera jamais (sauf si vous le déplacez manuellement). La flèche passera toujours par ce point. Ainsi, si vous voulez déplacer vos deux classes sans toucher à ce point, votre flèche fera une trajectoire assez étrange. 😊

Alors, premières impressions 😊 ? Si vous êtes parvenus sans encombre à réaliser ce qu'on vient de faire, les quatre prochaines interactions seront toutes aussi simples. 😊

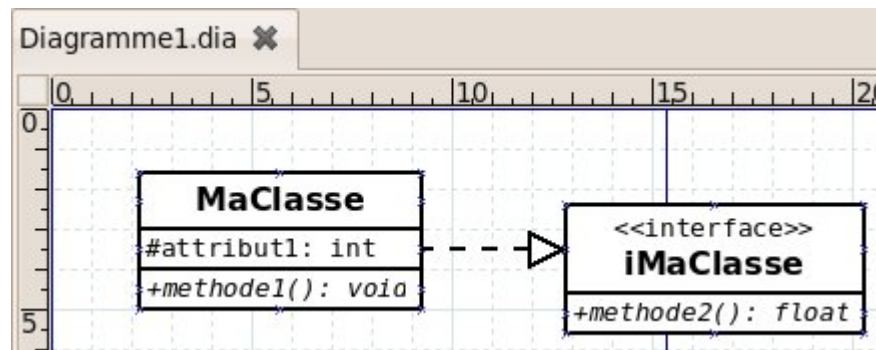
Les interfaces

Passons maintenant à l'implémentation d'interfaces. Créez une classe banale et, histoire que les classes coordonnent avec l'interaction, écrivez interface dans le champ texte Stéréotype de la classe. Pensez aussi à décocher la case Attributs visibles.

L'icône représentant l'interaction de l'implémentation d'interface est situé juste à gauche de celui de l'héritage :



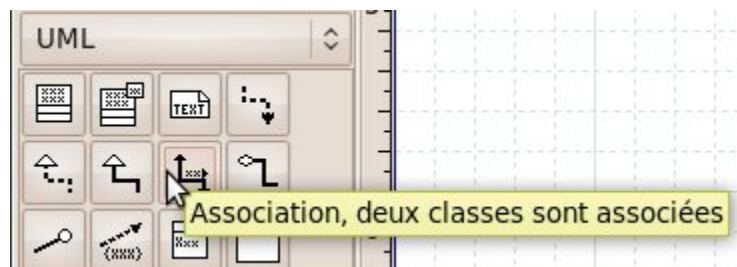
Comme pour la précédente interaction, cliquez sur une des croix bleues de l'interface, puis glissez la souris sur une croix bleue de la classe l'implémentant et relâchez la souris. Si tout s'est bien déroulé, vous devriez avoir obtenu ceci :



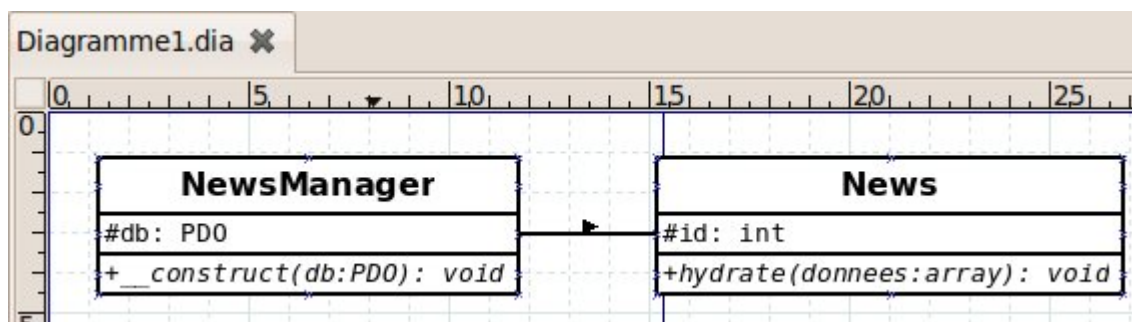
Vous pouvez ici remarquer l'absence des points rouges et du point orange sur la flèche. Je vous rappelle, pour ceux qui ne comprennent toujours pas très bien, que cela signifie que la flèche n'est plus sélectionnée. 😊

L'association

Voyons maintenant comment modéliser l'association. Cet icône est placé juste à droite de l'icône représentant l'héritage :



Cliquez donc sur la croix bleue de la classe ayant un attribut stockant une instance de l'autre classe, puis glissez sur cette autre classe. Vous devriez avoir ceci sous vos yeux :



Voyons maintenant comment définir l'association et placer les cardinalités. En fait, ce sont des options de l'association que l'on peut modifier en accédant à la configuration de cette liaison. Pour cela, double-cliquez sur la ligne. Vous devriez avoir obtenu cette fenêtre :

Propriétés : UML - Association

Nom:

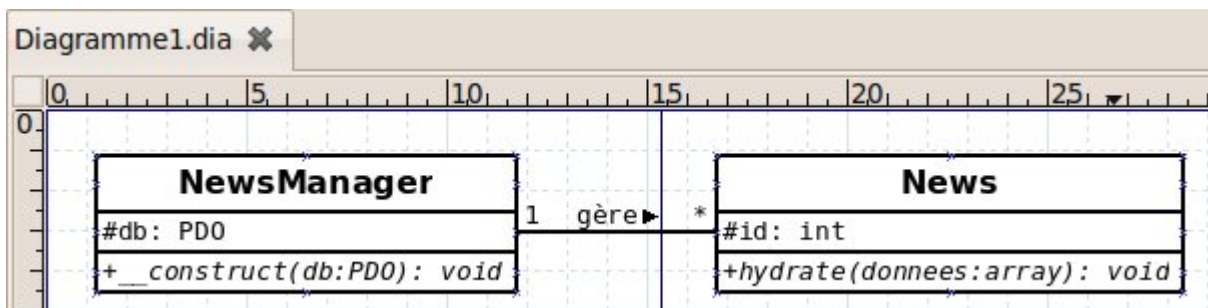
Direction:

Show direction:

Type:

	Side A	Side B
Rôle	<input type="text"/>	<input type="text"/>
Multiplicity	<input type="text"/>	<input type="text"/>
Visibilité	<input type="text" value="Implémentation"/>	<input type="text" value="Implémentation"/>
Afficher la flèche	<input type="text" value="Non"/>	<input type="text" value="Non"/>
Autoroute	<input type="text" value="Oui"/>	
Couleur du texte	<input type="text" value="#000000"/>	
Couleur des lignes	<input type="text" value="#000000"/>	

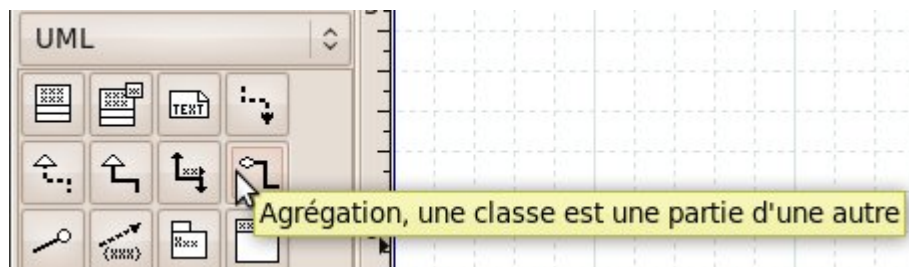
La définition de l'association se place dans le premier champ texte correspondant au nom. Mettez-y par exemple « gère ». Les cardinalités se placent dans les champs textes Multiplicity. La première colonne « Side A » correspond à la classe qui lie l'autre classe. Dans notre exemple, « Side A » correspond à NewsManager et « Side B » à News. Si vous vous êtes bien débrouillés, vous devriez avoir obtenu quelque chose dans ce genre :



L'agrégation

Intéressons-nous maintenant à la flèche représentant l'agrégation. On peut le faire de deux façons différentes. La première est la même que la précédente (j'entends par là que l'icône sur lequel cliquer est le même). Cette fois-ci, il ne faut pas afficher le sens de la liaison et afficher un losange. Pour ce faire, il suffit de sélectionner Aggregation dans la liste déroulante Type et cliquer sur le gros bouton Oui de la ligne Show direction. La seconde solution consiste à utiliser un autre icône qui nous construira la flèche avec le losange directement. En fait c'est juste un raccourci de la première solution. 😊

Cet icône se trouve juste à droite du précédent, à savoir ici :



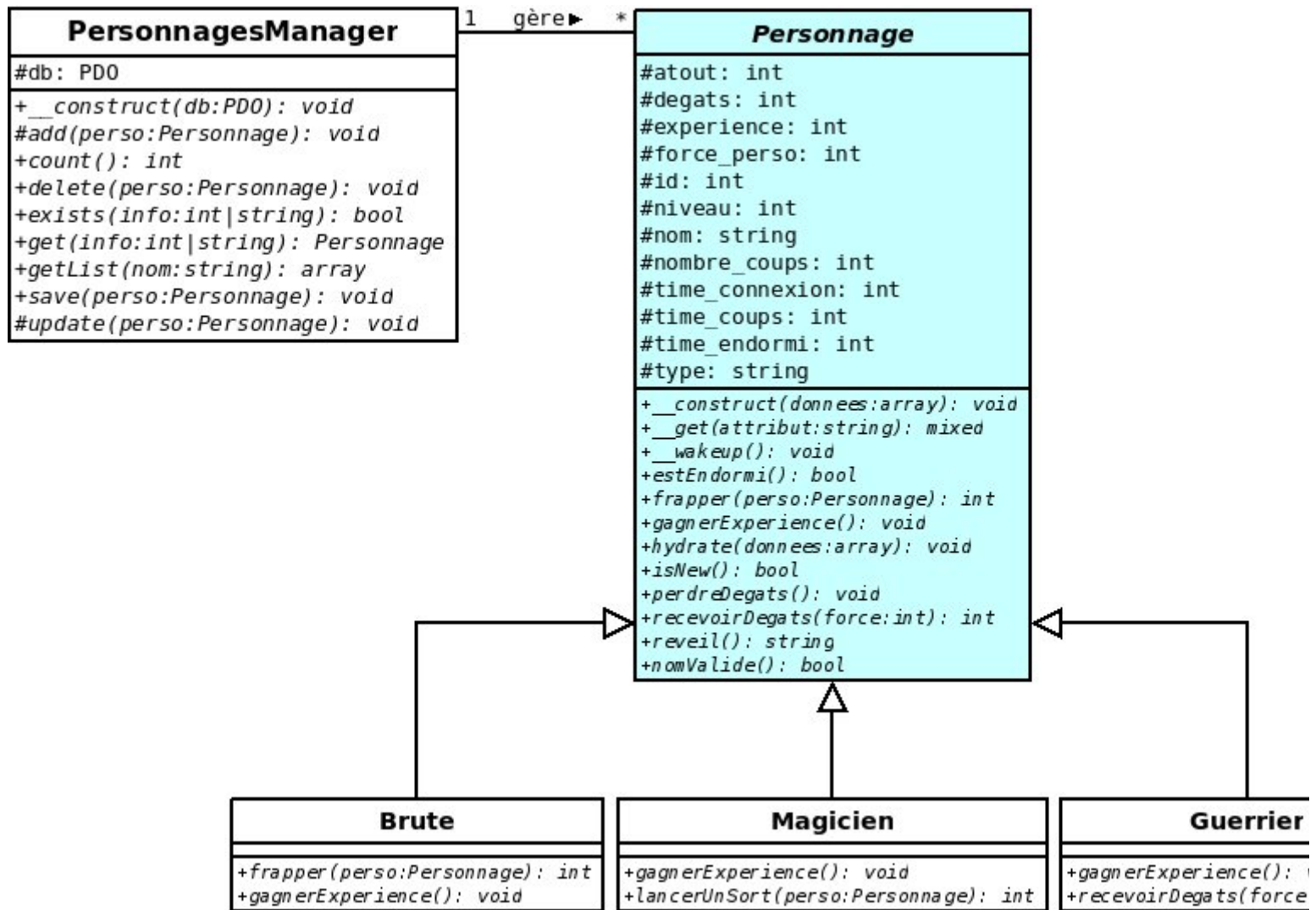
Je pense qu'il est inutile de vous dire comment lier vos deux classes étant donné que le principe reste le même. 😊

La composition

Et enfin, nous terminons par la composition. La composition n'a pas d'icône pour la représenter. Vous pouvez soit cliquer sur l'icône de l'association, soit cliquer sur l'icône de l'agrégation (ce dernier est à préférer dans le sens où une composition se rapproche beaucoup plus d'une agrégation que d'une association). Reliez vos classes comme on a fait jusqu'à maintenant puis double-cliquez sur celle-ci. Nous allons regarder la même liste déroulante à laquelle nous nous sommes intéressés pour l'agrégation : je parle de la liste déroulante Type. À l'intérieur de celle-ci, choisissez l'option Composition puis validez.

Exercice

Vous avez accumulé ici les connaissances pour faire un diagramme complet. Je vais donc vous demander de me reconstituer le diagramme que je vous ai donné au début, à savoir celui-ci :



Exploiter son diagramme

Vous avez maintenant un super gros diagramme avec tout plein d'interactions, mais qu'est-ce que vous allez en faire ? Je vais ici vous expliquer comment exporter votre diagramme dans deux formats différents bien pratiques. L'un sera sous forme d'image, l'autre sous forme de code PHP. 😊

Enregistrer son diagramme

Le chemin menant au dossier dans lequel vous allez enregistrer votre diagramme ne doit pas contenir d'espace ou de caractères spéciaux (lettres accentuées, signes spéciaux comme le ©, etc.). Je vous conseille donc (pour les utilisateurs de Windows), de créer un dossier à la racine de votre disque dur. S'il comporte des caractères spéciaux, vous ne pourrez pas double-cliquer dessus afin de le lancer (comme vous faites avec tout autre document), et s'il contient des espaces ou caractères spéciaux vous ne pourrez pas exporter par la suite votre diagramme sous forme de code PHP (et tout autre langage que propose le logiciel).

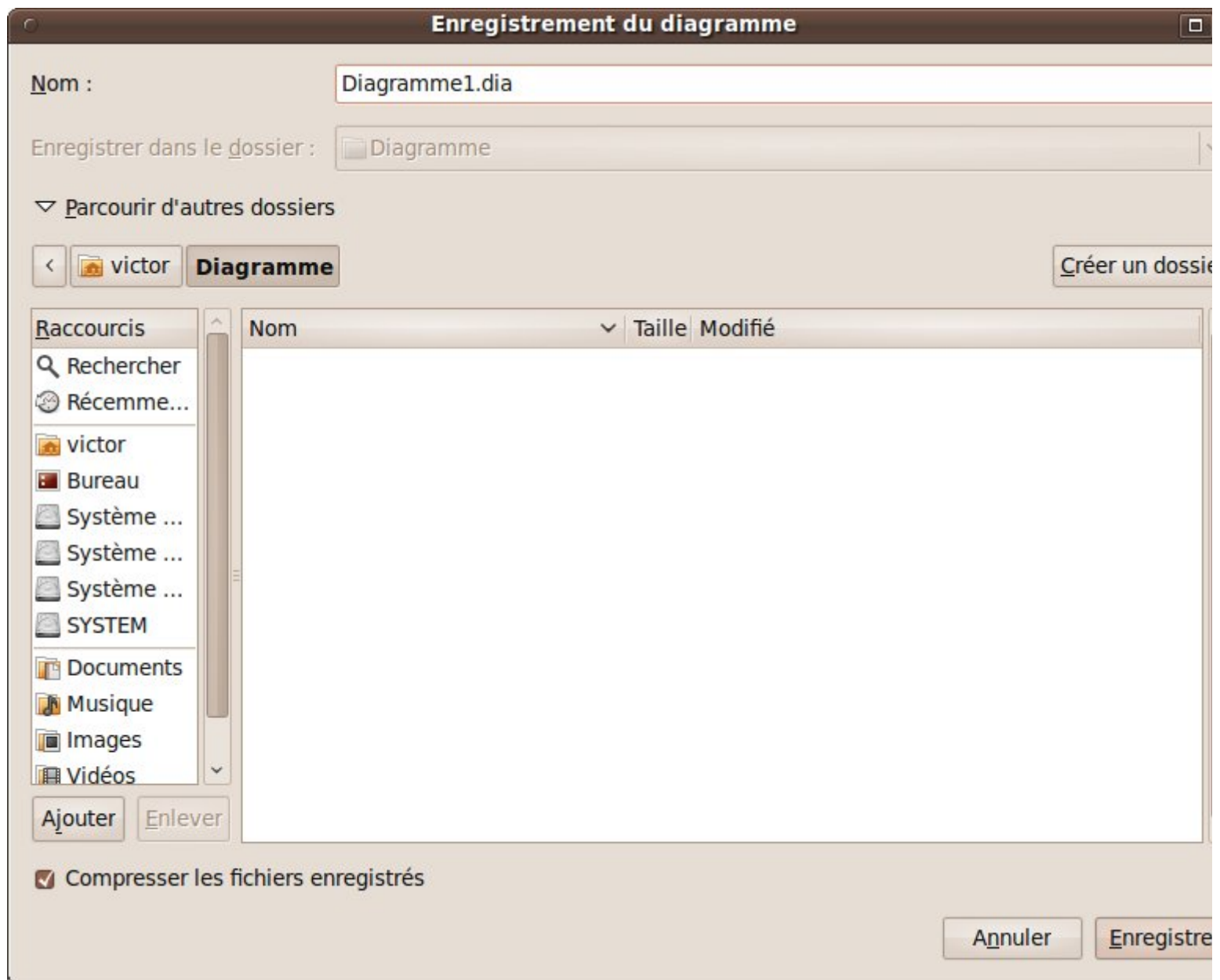
Avant toute exportation, il faut sauvegarder son diagramme, sinon ça ne fonctionnera pas. Pour ce faire, cliquez sur le menu Fichier puis cliquez sur Enregistrer.



Ou cliquez sur cette icône :



Cette fenêtre s'ouvre à vous :



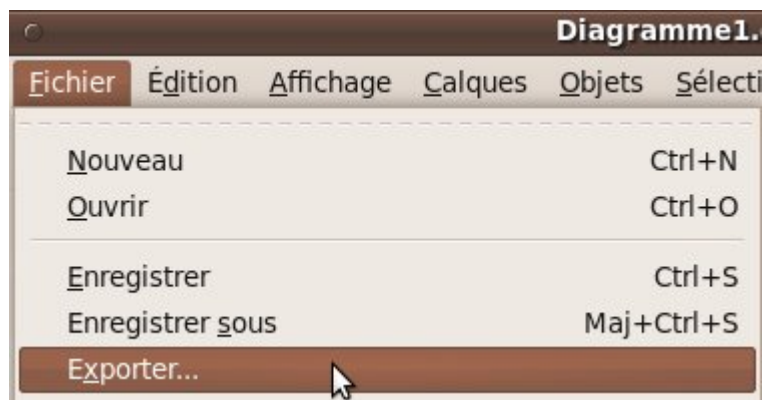
Tout est écrit en français, je ne pense pas avoir besoin d'expliquer en détails. Cette fenêtre contient un explorateur. À gauche sont listés les raccourcis menant aux dossiers les plus courants (le bureau, les documents, ainsi que les disques), tandis qu'à droite sont listés tous les fichiers et dossiers présents dans celui où vous vous trouvez (dans mon cas, le dossier est vierge).

Le champ texte situé tout en haut contient le nom du fichier sous lequel doit être enregistré le diagramme. Une fois que vous avez bien tout paramétré, cliquez sur Enregistrer. Votre diagramme est maintenant enregistré au format .dia.

Exporter son diagramme

Sous forme d'image

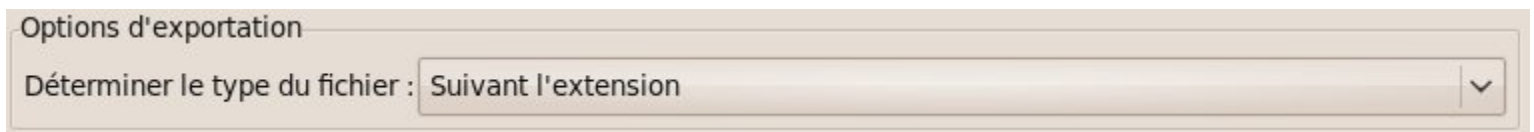
Nous avons enregistré notre diagramme, nous sommes maintenant fin prêts à l'exporter. Pour obtenir une image de ce diagramme, on va cliquer sur Exporter dans le menu Fichier.



Ou cliquez sur cette icône :



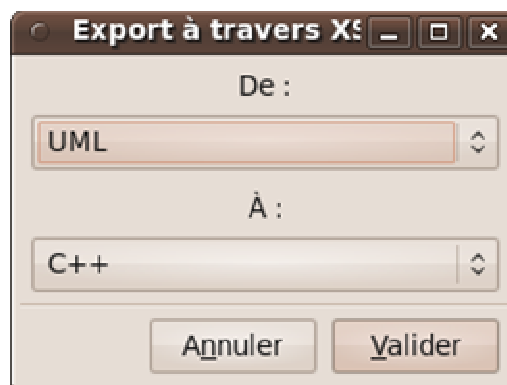
Une fenêtre semblable à celle de l'enregistrement du diagramme apparaît. Cependant, il y a une liste déroulante qui a fait son apparition :



Pour exporter votre diagramme sous forme d'image, vous avez deux possibilités. Soit vous placez l'extension .png à la fin du nom de votre image en haut, soit, dans la liste déroulante, vous sélectionnez l'option Pixbuf[png] (*.png). Avec cette deuxième solution, le champ texte du haut contiendra le nom de votre image avec l'extension .png. Cliquez sur Enregistrer, puis contemplez votre image générée. Pas mal, hein ? 😊

Sous forme de code PHP

Au début, la marche à suivre reste la même, c'est-à-dire qu'il faut aller dans le menu Fichier puis cliquer sur Exporter. Cette fois-ci, nous allons choisir une autre option (logique 🤖). Vous pouvez là aussi réaliser l'opération de deux façons. Soit vous placez à la fin du nom du diagramme l'extension .code, soit vous sélectionnez Filtre de transformation XSL (*.code) dans la liste déroulante. Cliquez sur Enregistrer. Cette nouvelle fenêtre apparaît :



Dans la première liste déroulante, choisissez UML-CLASSES-EXTENDED. Dans la seconde liste, choisissez PHP5 puis cliquez sur Valider. Regardez les nouveaux fichiers générés dans votre dossier. Alors, ça en jette non ? 😊

Si vous n'avez pas l'option UML-CLASSES-EXTENDED c'est que vous n'avez pas installé comme il faut l'extension uml2php5. Relisez donc bien la première partie du tutoriel.

L'UML ne devrait plus avoir énormément de secrets pour vous désormais. Il est très important de savoir ce qu'est l'UML et savoir modéliser, ça pourra vous être utile à de multiples reprises (surtout dans les milieux professionnels). 😊

Les design patterns

Nous allons découvrir dans ce chapitre ce que sont les design patterns (ou motifs de conception). Les design patterns sont donc des façons de concevoir des classes afin de répondre à un problème que nous sommes susceptibles de rencontrer. Ce sont des bonnes pratiques, mais il faut faire attention à ne pas trop en abuser sous prétexte que ça fait classe. 🤔

Laisser une classe créant les objets : le pattern Factory

Le problème

Admettons que vous venez de créer une assez grosse application. Vous avez construit cette application en associant plus ou moins la plupart de vos classes entre elles. À présent, vous voudriez modifier un petit morceau de code afin d'ajouter une fonctionnalité à l'application. Problème : étant donné que la plupart de vos classes sont plus ou moins liées, il va falloir modifier un tas de chose ! Le pattern Factory pourra sûrement vous aider.

Ce motif est très simple à construire. En fait, si vous implémentez ce pattern, vous n'aurez plus de **new** à placer dans la partie globale du script afin d'instancier une classe. En effet, ce ne sera pas à vous de le faire mais à une classe usine. Cette classe aura pour rôle de charger les classes que vous lui passez en argument. Ainsi, quand vous modifierez votre code, vous n'aurez qu'à modifier le masque d'usine pour que la plupart des modifications prennent effet. En gros, vous ne vous soucierez plus de l'instanciation de vos classes, ce sera à l'usine de le faire !

Voici comment se présente une classe implémentant le pattern Factory :

Code : PHP

```
<?php
class DBFactory
{
    public static function load ($sgbdr)
    {
        $classe = 'SGBDR_' . $sgbdr;

        if (file_exists ($chemin = $classe . '.class.php'))
        {
            require $chemin;
            return new $classe;
        }
        else
            throw new RuntimeException ('La classe <strong>' . $classe . '</strong> n\'existe pas');
    }
}
```

Dans votre script, vous pourrez donc faire quelque chose de ce genre :

Code : PHP

```
<?php
try
{
    $mysql = DBFactory::load('MySQL');
}
catch (RuntimeException $e)
{
    echo $e->getMessage();
}

?>
```

Exemple concret

Le but est de créer une classe qui nous distribuera les objets PDO plus facilement. Nous allons partir du principe que vous avez plusieurs SGBDR, ou plusieurs BDD qui utilisent des identifiants différents. Bref, nous allons tout centraliser dans une classe.

Code : PHP

```
<?php
class PDOFactory
{
    public static function getMysqlConnexion()
    {
        $db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');
        $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        return $db;
    }

    public static function getPgsqlConnexion()
    {
        return new PDO('pgsql:host=localhost;dbname=tests', 'root', '');
        $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        return $db;
    }
}

?>
```

Ceci vous simplifiera énormément la tâche. Si vous avez besoin de modifier vos identifiants de connexion, vous n'aurez pas à aller chercher dans tous vos scripts : tout sera placé dans notre factory. 😊

Écouter ses objets : le pattern Observer

Le problème

Dans votre script est présent une classe s'occupant de la gestion d'un module. Lors d'une action précise, vous exécutez une ou plusieurs instructions. Celles-ci n'ont qu'une chose en commun : le fait qu'elles soient appelées car telle action s'est produite. Elles n'ont rien d'autre en commun, elles sont un peu foutues dans la méthode « parce qu'il faut bien les appeler et qu'on sait pas où les mettre ». Il est intéressant dans ce cas-là de séparer les différentes actions effectuées lorsque telle action survient. Pour cela, nous allons regarder du côté du pattern Observer.

Le principe est simple : vous avez une classe observée et une ou plusieurs autre(s) classe(s) qui l'observe(nt). Lorsque telle action survient, vous allez prévenir toutes les classes qui l'observent. Nous allons, pour une raison d'homogénéité, utiliser les interfaces prédéfinies de la SPL. Il s'agit d'une librairie standard qui est fournie d'office avec PHP. Elle contient différentes classes, fonctions, interfaces, etc. Vous vous en êtes déjà servi en utilisant `spl_autoload_register()`. 😊

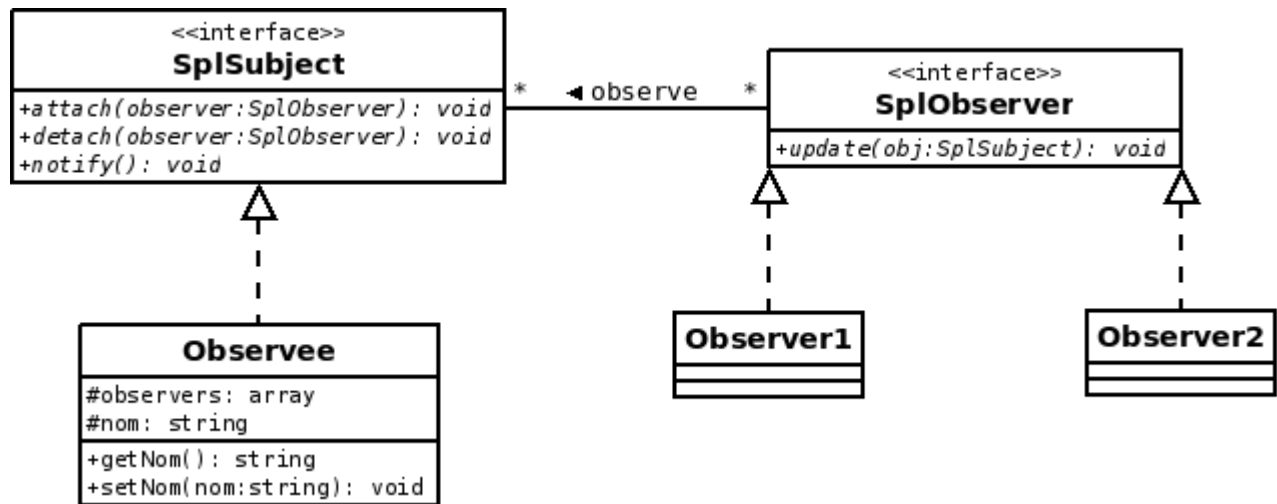
Bref, regardons plutôt ce qui nous intéresse, à savoir deux interfaces : [SplSubject](#) et [SplObserver](#).

La première interface, `SplSubject`, est l'interface implémentée par l'objet observé. Elle contient trois méthodes :

- `attach (SplObserver $observer)` : méthode appelée pour ajouter une classe observatrice à notre classe observée ;
- `detach (SplObserver $observer)` : méthode appelée pour supprimer une classe observatrice ;
- `notify()` : méthode appelée lorsqu'on aura besoin de prévenir toutes les classes observatrices que quelque chose s'est produit.

L'interface `SplObserver` est l'interface implémentée par les différents observateurs. Elle ne contient qu'une seule méthode qui est celle appelée par la classe observée dans la méthode `notify()` : il s'agit de `update ((SplSubject $subject))`.

Voici un diagramme mettant en œuvre ce design pattern :



On va maintenant imaginer le code correspondant au diagramme. Commençons par la classe observée :

Code : PHP - Classe observée

```

<?php
class Observee implements SplSubject
{
    // Ceci est le tableau qui va contenir tous les objets qui nous observent
    protected $observers = array();

    // Dès que cet attribut changera on notifiera les classes observatrices
    protected $nom;

    public function attach (SplObserver $observer)
    {
        $this->observers[] = $observer;
    }

    public function detach (SplObserver $observer)
    {
        if (is_int ($key = array_search ($observer, $this->observers, true)))
            unset ($this->observers[$key]);
    }

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

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

    public function setNom ($nom)
    {
        $this->nom = $nom;
        $this->notify();
    }
}
?>
  
```

Vous pouvez constater la présence du nom des interfaces en guise d'argument. Cela veut dire que cet argument doit implémenter l'interface spécifiée.

Voici les deux classes observatrices :

Code : PHP - Classes observatrices

```
<?php
class Observer1 implements SplObserver
{
    public function update (SplSubject $obj)
    {
        echo __CLASS__, ' a été notifié ! Nouvelle valeur de l\'attribut <strong>nom</strong> : ' . $obj->getNom();
    }
}

class Observer2 implements SplObserver
{
    public function update (SplSubject $obj)
    {
        echo __CLASS__, ' a été notifié ! Nouvelle valeur de l\'attribut <strong>nom</strong> : ' . $obj->getNom();
    }
}
```

Ces deux classes font exactement la même chose, ce n'était qu'à titre d'exemple basique que je vous ai donné ça, histoire que vous voyez la syntaxe de base lors de l'utilisation du pattern Observer.

Pour tester nos classes, vous pouvez utiliser ce bout de code :

Code : PHP

```
<?php
$o = new Observee;
$o->attach(new Observer1); // Ajout d'un observateur
$o->attach(new Observer2); // Ajout d'un autre observateur
$o->setNom('Victor'); // On modifie le nom pour voir si les classes observatrices ont
```

Vous pouvez voir qu'ajouter des classes observatrices de cette façon peut être assez long si on en a 5 ou 6. Il y a une petite technique qui consiste à pouvoir obtenir ce genre de code :

Code : PHP

```
<?php
$o = new Observee;

$o->attach(new Observer1)
->attach(new Observer2)
->attach(new Observer3)
->attach(new Observer4)
->attach(new Observer5);

$o->setNom('Victor'); // On modifie le nom pour voir si les classes observatrices ont
```

Pour effectuer ce genre de manœuvres, la méthode `attach()` doit retourner l'instance qui l'a appelé (en d'autres termes, elle doit retourner `$this`).

Exemple concret

Regardons un exemple concret à présent. Nous allons imaginer que vous ayez, dans votre script, une classe gérant les erreurs générées par PHP. Lorsqu'une erreur est générée, vous aimeriez qu'il se passe deux choses :

- Que l'erreur soit enregistrée en BDD ;

- Que l'erreur vous soit envoyée par mail.

Pour cela, vous pensez donc coder une classe comportant une méthode attrapant l'erreur et effectuant les deux opérations ci-dessus. Grave erreur ! Ceci est surtout à ne pas faire : votre classe est chargée d'intercepter les erreurs, et non de les gérer ! Ce sera à d'autres classes de s'en occuper : ces classes vont observer la classe gérant l'erreur et une fois notifiée, elles vont effectuer l'action pour laquelle elles ont été conçues. Vous voyez un peu la tête qu'aura le script ?

Rappel : pour intercepter les erreurs, il vous faut utiliser `set_error_handler()`. Pour faire en sorte que la fonction de callback appelée lors de la génération d'une erreur soit une méthode d'une classe, passez un tableau à deux entrées en premier argument. La première entrée est l'objet sur lequel vous allez appeler la méthode, et la seconde est le nom de la méthode.

Vous êtes capables de le faire tout seul. Voici la correction :

Secret ([cliquez pour afficher](#))

Pfiou, ça en fait du code ! Je ne sais pas si vous vous en rendez compte, mais ce qu'on vient de créer là est une excellente manière de coder. Nous venons de séparer notre code comme il se doit et nous pourrions le modifier aisément car les différentes actions ont été séparées avec logique.

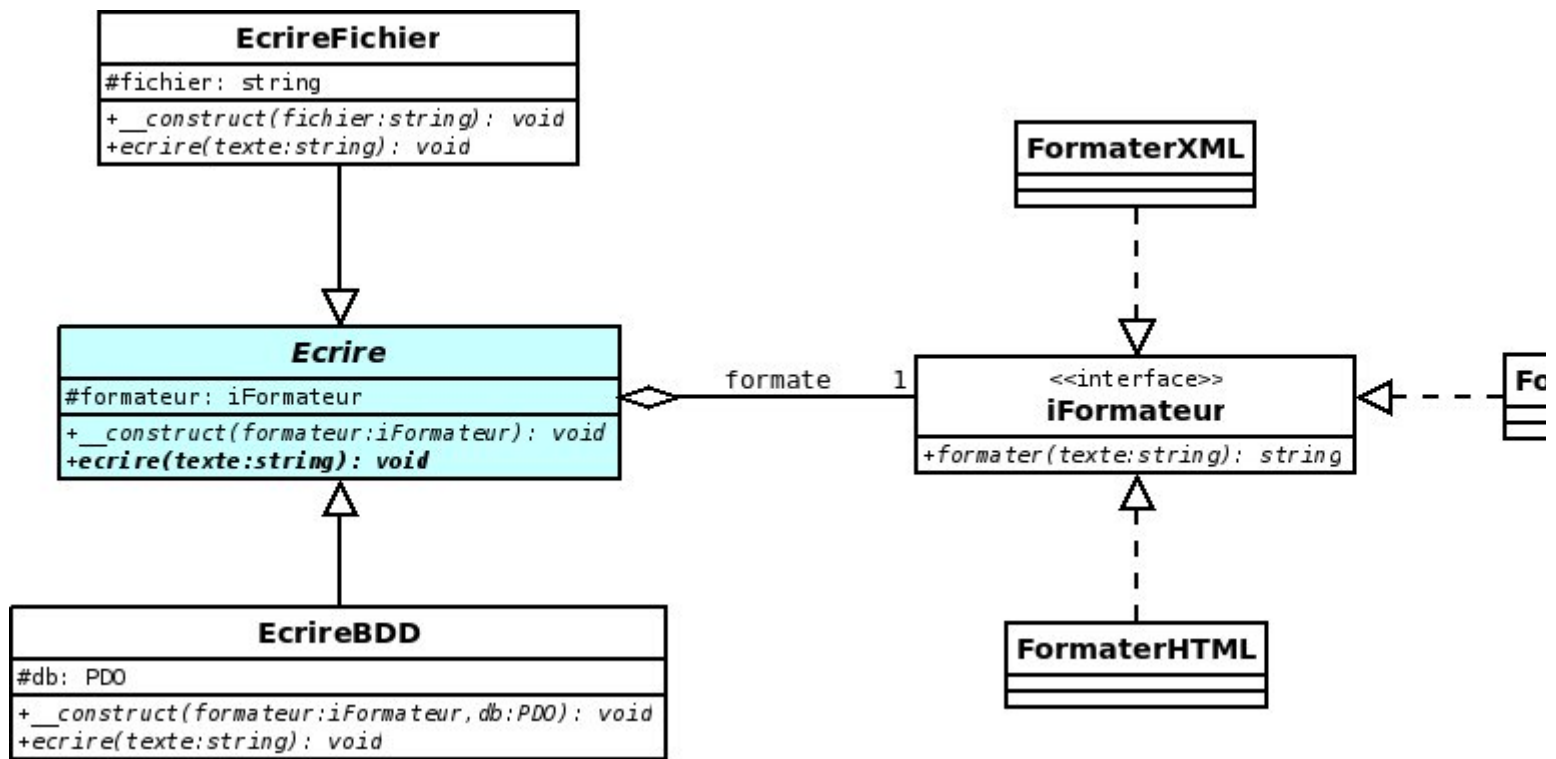
Séparer ses algorithmes : le pattern Strategy

Le problème

Vous avez une classe dédiée à une tâche spécifique. Dans un premier temps, celle-ci effectue une opération suivant un algorithme bien précis. Cependant, avec le temps, cette classe sera amenée à évoluer, et elle suivra plusieurs algorithmes, tout en effectuant la même tâche de base. Par exemple, vous avez une classe `EcrireFichier` qui a pour rôle d'écrire dans un fichier ainsi qu'une classe `EcrireBDD`. Dans un premier temps, ces classes ne contiennent qu'une méthode `ecrire()` qui n'écrit que le texte passé en paramètre dans le fichier ou dans la BDD. Au fil du temps, vous vous rendez compte que c'est dommage qu'elles ne fassent que ça et vous aimeriez bien qu'elles puissent écrire en différents formats (HTML, XML, etc.) : les classes doivent donc formater puis écrire. C'est à ce moment qu'il est intéressant de se tourner vers le pattern Strategy. En effet, sans ce design pattern, vous seriez obligés de créer deux classes différentes pour écrire au format HTML par exemple : `EcrireFichierHTML` et `EcrireBDDHTML`. Pourtant, ces deux classes devront formater le texte de la même façon : nous assisterons à une duplication du code, et c'est la pire chose à faire dans un script ! Imaginez que vous voulez modifier l'algorithme dupliqué une dizaine de fois... Pas très pratique n'est-ce pas ?

Exemple concret

Passons directement à l'exemple concret. Nous allons suivre l'idée que nous avons évoquée à l'instant : l'action d'écrire dans un fichier ou dans une BDD. Il y aura pas mal de classes à créer donc au lieu de vous faire un grand discours, je vais vous montrer le diagramme représentant l'application :



Ça en fait des classes ! Pourtant (je vous assure) le principe est très simple à comprendre. La classe Ecrire est abstraite (ça n'aurait aucun sens de l'instancier : on veut écrire, ok, mais sur quel support ?) et implémente un constructeur qui acceptera un argument : il s'agit du formateur que l'on souhaite utiliser. Nous allons aussi placer une méthode abstraite `ecrire()`, ce qui forcera toutes les classes filles de Ecrire à implémenter cette méthode qui appellera la méthode `formater()` du formateur associé (instance contenue dans l'attribut `$formateur`) afin de récupérer le texte formaté. Allez, au boulot ! 😊

Commençons par l'interface. Rien de bien compliqué, elle ne contient qu'une seule méthode :

Code : PHP - iFormateur.interface.php

```

<?php
interface iFormateur
{
    public function formater ($texte);
}
?>

```

Ensuite vient la classe abstraite Ecrire que voici :

Code : PHP - Ecrire.class.php

```

<?php
abstract class Ecrire
{
    // Attribut contenant l'instance du formateur que l'on veut utiliser
    protected $formateur;

    abstract public function ecrire ($texte);

    // Nous voulons une instance d'une classe implémentant iFormateur en paramètre
    public function __construct (iFormateur $formateur)
    {
        $this->formateur = $formateur;
    }
}
?>

```

Nous allons maintenant créer deux classes héritant de Ecrire : EcrireFichier et EcrireBDD.

Code : PHP - EcrireBDD.class.php

```
<?php
class EcrireBDD extends Ecrire
{
    protected $db;

    public function __construct (iFormateur $formateur, PDO $db)
    {
        parent::__construct($formateur);
        $this->db = $db;
    }

    public function ecrire ($texte)
    {
        $q = $this->db->prepare('INSERT INTO lorem_ipsum SET texte = :texte');
        $q->bindValue(':texte', $this->formateur->formater($texte));
        $q->execute();
    }
}

?>
```

Code : PHP - EcrireFichier.class.php

```
<?php
class EcrireFichier extends Ecrire
{
    // Attribut stockant le chemin du fichier
    protected $fichier;

    public function __construct (iFormateur $formateur, $fichier)
    {
        parent::__construct($formateur);
        $this->fichier = $fichier;
    }

    public function ecrire ($texte)
    {
        $f = fopen ($this->fichier, 'w');
        fwrite ($f, $this->formateur->formater($texte));
        fclose ($f);
    }
}

?>
```

Et enfin, nous avons nos trois formateurs. L'un ne fait rien de particulier (FormaterTexte), et les deux autres formatent le texte en deux langages différents (FormaterHTML et FormaterXML). J'ai décidé d'ajouter le timestamp dans le formatage du texte histoire que le code ne soit pas complètement inutile (surtout pour la classe qui ne fait pas de formatage particulier). 🤨

Code : PHP - FormaterTexte.class.php

```
<?php
class FormateurTexte implements iFormateur
{
    public function formater ($texte)
    {
        return 'Date : ' . time() . "\n" . 'Texte : ' . $texte;
    }
}

?>
```

Code : PHP - FormaterHTML.class.php

```
<?php
class FormateurHTML implements iFormateur
{
    public function formater ($texte)
    {
        return '<p>Date : ' . time() . '<br />' . "\n". 'Texte : ' . $texte . '</p>';
    }
}
?>
```

Code : PHP - FormaterXML.class.php

```
<?php
class FormateurXML implements iFormateur
{
    public function formater ($texte)
    {
        return '<?xml version="1.0" encoding="ISO-8859-1"?>' . "\n".
            '<message>' . "\n".
            "\t". '<date>' . time() . '</date>' . "\n".
            "\t". '<texte>' . $texte . '</texte>' . "\n".
            '</message>';
    }
}
?>
```

Et testons enfin notre code :

Code : PHP - index.php

```
<?php
function autoload ($classe)
{
    if (file_exists ($chemin = $classe . '.class.php') OR file_exists ($chemin = $classe . '.php'))
        require $chemin;
}

spl_autoload_register ('autoload');

$ecritureFichier = new EcrireFichier (new FormateurHTML, 'fichier.html');
$ecritureFichier->ecrire('Hello world !');
```

Ce code de base a l'avantage d'être très flexible. Il peut paraître un peu gros pour ce que nous avons à faire, mais si l'application est amenée à obtenir beaucoup de fonctionnalités supplémentaires, nous aurons déjà préparé le terrain ! 😊

Une classe, une instance : le pattern Singleton

Nous allons terminer par un pattern qui est en général le premier qu'on vous présente. Si je ne vous l'ai pas présenté au début c'est parce que je veux que vous fassiez attention avec car il peut être très mal utilisé et se transformer en mauvaise pratique. On considèrera alors le pattern comme un « anti-pattern ». Cependant, il est très connu et par conséquent très important de savoir ce que c'est mais surtout : savoir pourquoi il ne faut pas l'utiliser dans certains contextes.

Le problème

Nous avons une classe qui ne doit être instanciée qu'une seule fois. À première vue, ça vous semble impossible, et c'est normal. Jusqu'à présent, nous pouvions faire de multiples `$obj = new Classe;` jusqu'à l'infini, et nous nous retrouvions avec une infinité d'instances de Classe. Il va donc falloir empêcher ceci.

Pour empêcher de créer une instance de cette façon, c'est très simple : il suffit de mettre le constructeur de la classe en privé ou en protégé !

T'es marrant toi, on ne pourra jamais créer d'instance avec cette technique !

Bien sur que si ! Nous allons créer une instance de notre classe à l'intérieur d'elle-même ! De cette façon nous aurons accès au constructeur. 😊

Oui mais voilà, il ne va falloir créer qu'une seule instance... On va donc créer un attribut statique dans notre classe qui contiendra... l'instance de cette classe ! Nous aurons aussi une méthode statique qui aura pour rôle de renvoyer cette instance. Si on l'appelle pour la première fois, alors on instancie la classe puis on retourne l'objet, sinon on se contente de le retourner. 😊

Il y a aussi un petit détail à régler. Nous voulons vraiment une seule instance, et là il est encore possible d'en avoir plusieurs. En effet, rien n'empêche l'utilisateur de cloner l'instance ! Il faut donc bien penser à interdire l'accès à la méthode `__clone()`. 😊

Ainsi, une classe implémentant le pattern Singleton ressemblerait à ceci :

Code : PHP

```
<?php
class MonSingleton
{
    protected static $instance; // Contiendra l'instance de notre classe

    protected function __construct() { } // Constructeur en privé
    protected function __clone() { } // Méthode de clonage en privé aussi

    public static function getInstance()
    {
        if (!isset (self::$instance)) // Si on n'a pas encore instancié notre classe
            self::$instance = new self; // On s'instancie nous-mêmes :)

        return self::$instance;
    }
}
?>
```

Ceci est le strict minimum. À vous d'implémenter de nouvelles méthodes, comme vous l'auriez fait dans votre classe normale. 😊

Voici donc une utilisation de la classe :

Code : PHP

```
<?php
$obj = MonSingleton::getInstance(); // Premier appel : instance créée
$obj->methode1();
?>
```

Exemple concret

Un exemple concret pour le pattern Singleton ? Non, désolé, on va devoir s'en passer. 😊

Hein ? Quoi ? Tu te moques de moi ? Alors il sert à rien ce design pattern ? 😊

Selon moi, non. Je n'ai encore jamais eu besoin de l'utiliser. Ce pattern doit être utilisé uniquement si plusieurs instantiations de la classe provoqueraient un dysfonctionnement. Si le script peut continuer normalement alors que plusieurs instances sont créées, le pattern Singleton ne doit pas être utilisé.

Donc en gros, ce qu'on a appris là, c'est du vent ?

Non. Il est important de connaître ce design pattern, non pas pour l'utiliser, mais pour ne pas l'utiliser, et surtout savoir pourquoi.

Ce pattern peut vous sembler beau au premier abord, mais je dois vous parler de quelque chose : l'injection de dépendances. On n'en a pas parlé jusqu'à maintenant, car je n'en ai pas eu l'occasion. Ici est le moment idéal. Commençons par un exemple.

Soit le code suivant :

Code : PHP

```
<?php
class NewsManager
{
    public function get($id)
    {
        // On admet que MyPDO étend de PDO et qu'il implémente un singleton
        $q = MyPDO::getInstance()->query('SELECT id, auteur, titre, contenu FROM news');

        return $q->fetch(PDO::FETCH_ASSOC);
    }
}
```

Vous vous apercevez qu'ici, le singleton a introduit une dépendance entre deux classes n'appartenant pas au même module. Deux modules ne doivent jamais être liés de cette façon, ce qui est le cas ici. Deux modules doivent être indépendants les uns des autres. D'ailleurs, en y regardant de plus près, ça ressemble fortement à une variable globale. En effet, un singleton n'est rien d'autre qu'une variable globale déguisée (il y a juste une étape en plus pour accéder à la variable) :

Code : PHP

```
<?php
class NewsManager
{
    public function get($id)
    {
        global $db;
        // Revient EXACTEMENT au même que :
        $db = MyPDO::getInstance();

        // Suite des opérations
    }
}
```

Vous ne voyez pas où est le problème ? Souvenez-vous de l'un des points forts de la POO : le fait de pouvoir redistribuer sa classe ou la réutiliser. Là, on ne peut pas, car notre classe NewsManager dépend de MyPDO. Qu'est-ce qui vous dit que la personne qui utilisera NewsManager aura cette dernière ? Rien du tout, et c'est normal. Nous sommes ici face à une dépendance créée par le singleton. De plus, la classe dépend aussi de PDO : il y avait donc déjà une dépendance au début, et le pattern Singleton en a créé une autre. Il faut donc supprimer ces deux dépendances.

Comment faire alors ?

Ce qu'il faut, c'est passer notre DAO au constructeur, sauf que notre classe ne doit pas être dépendante d'une quelconque bibliothèque. Ainsi, notre objet peut très bien utiliser PDO, MySQLi ou que sais-je encore, la classe se servant de lui doit fonctionner de la même manière. Alors comment procéder ? Il faut imposer un comportement spécifique à notre objet en l'obligeant à implémenter certaines méthodes. Je ne vous fais pas attendre : les interfaces sont là pour ça. On va donc créer une interface iDB contenant (pour faire simple) qu'une seule méthode : **query()**.

Code : PHP

```
<?php
interface iDB
{
    public function query($query);
}
```

Pour que l'exemple soit parlant, nous allons créer deux classes utilisant cette structure, l'une utilisant PDO et l'autre MySQLi. Cependant, un problème se pose : le résultat retourné par la méthode **query()** des classes PDO et MySQLi sont des instances de deux classes différentes, et les méthodes disponibles ne sont par conséquent pas les mêmes. Il faut donc créer d'autres classes pour gérer les résultats qui suivent elles aussi une structure définie par une interface (admettons iResult).

Code : PHP

```
<?php
interface iResult
{
    public function fetchAssoc();
}
```

Nous pouvons donc à présent écrire nos 4 classes : MyPDO, MyMySQLi, MyPDOStatement et MyMySQLiResult.

Code : PHP - MyPDO

```
<?php
class MyPDO extends PDO implements iDB
{
    public function query($query)
    {
        return new MyPDOStatement($parent::query($query));
    }
}
```

Code : PHP - MyPDOStatement

```
<?php
class MyPDOStatement implements iResult
{
    protected $st;

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

    public function fetchAssoc()
    {
        return $this->st->fetch(PDO::FETCH_ASSOC);
    }
}
```

Code : PHP - MyMySQLi

```
<?php
class MyMySQLi extends MySQLi implements iDB
{
    public function query($query)
    {
        return new MyMySQLiResult($parent::query($query));
    }
}
```

Code : PHP - MyMySQLiResult


```
<?php
class MyMySQLiResult implements iResult
{
    protected $st;

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

    public function fetchAssoc()
    {
        return $this->st->fetch_assoc();
    }
}
```

On peut donc maintenant écrire notre classe NewsManager. N'oubliez pas de vérifier que les objets sont bien des instances de classes implémentant les interfaces désirées. 😊

Code : PHP

```
<?php
class NewsManager
{
    protected $dao;

    // On souhaite un objet instanciant une classe qui implémente iDB
    public function __construct(iDB $dao)
    {
        $this->dao = $dao;
    }

    public function get($id)
    {
        $q = $this->dao->query('SELECT id, auteur, titre, contenu FROM news WHERE id = ' . $id);

        // On vérifie que le résultat implémente bien iResult
        if (!$q instanceof iResult)
        {
            throw new Exception('Le résultat d\'une requête doit être un objet implémentant iResult');
        }

        return $q->fetchAssoc();
    }
}
```

Testons maintenant notre code.

Code : PHP

```
<?php
$dao = new MyPDO('mysql:host=localhost;dbname=news', 'root', '');
// $dao = new MyMySQLi('localhost', 'root', '', 'news');

$manager = new NewsManager($dao);
print_r($manager->get(2));
```

Je vous laisse commenter et décommenter les deux premières lignes pour vérifier que les deux fonctionnent. Après quelques tests, vous vous rendrez compte que nous avons bel et bien découplé nos classes ! Il n'y a ainsi plus aucune dépendance entre notre classe NewsManager et une quelconque autre classe.

Si vous voulez en savoir plus sur les risques apportés par le singleton et sur l'injection de dépendances, je vous invite à lire [cet article](#).

Le problème dans notre cas, c'est qu'il est difficile de faire de l'injection de dépendances pour qu'une classe supporte toutes les bibliothèques d'accès aux BDD (PDO, MySQLi, etc.) à cause des résultats des requêtes. De son côté, PDO a la classe PDOStatement,

tandis que MySQLi a MySQLi_STMT pour les requêtes préparées et MySQLi_Result pour les résultats de requêtes classiques. Cela est donc difficile de les conformer au même modèle. On va donc, dans le TP qui va venir, utiliser une autre technique pour découpler nos classes.

En résumé

Le principal problème du singleton est de favoriser les dépendances entre deux classes. Il faut donc être très méfiant de ce côté-là, car votre application deviendra difficilement modifiable et on perd alors les avantages de la POO. En bref, je vous recommande d'utiliser le singleton en dernier recours : si vous décidez d'implémenter ce pattern, c'est pour garantir que cette classe ne doit être instanciée qu'une seule fois. Si vous vous rendez compte que deux instances ou plus ne causent pas de problème à l'application, alors n'implémentez pas le singleton. Et par pitié : n'implémentez pas un singleton pour l'utiliser comme une variable globale ! C'est la pire des choses à faire car cela favorise les dépendances entre classes comme on l'a vu.

Ce chapitre est très important. Je ne vous demande pas de retenir tous ces design patterns non plus, mais sachez que ce sont les principaux et qu'ils peuvent vous être utile à plusieurs reprises. Cependant, n'en abusez pas trop : ne les utilisez que quand vous en avez réellement besoin ! 😊

TP : un système de news

Voilà enfin le TP que la plupart de vous attendaient : un système de news orienté objet ! En effet, vous découvrirez de façon concrète comment structurer une petite application en suivant un modèle OO. Qu'attendons-nous ? 😊

Ce qu'on va faire

Commençons par savoir ce qu'on va faire et surtout, de quoi on va avoir besoin.

Ce que nous allons réaliser est très simple, à savoir un système de news basique avec les fonctionnalités suivantes :

- Affichage des 5 premières news à l'accueil du site avec texte réduit à 200 caractères ;
- Possibilité de cliquer sur le titre de la news pour la lire entièrement. L'auteur et la date d'ajout apparaîtront, ainsi que la date de modification si la news a été modifiée ;
- Un espace d'administration qui permettra d'ajouter / modifier / supprimer des news. Cet espace tient sur une page : il y a un formulaire et un tableau en-dessous listant les news avec des liens modifier / supprimer. Quand on clique sur « Modifier », le formulaire se pré-remplit.

Pour réaliser cela, nous allons avoir besoin de créer une table news dont la structure est la suivante :

Code : SQL

```
CREATE TABLE `news` (  
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,  
  `auteur` varchar(30) NOT NULL,  
  `titre` varchar(100) NOT NULL,  
  `contenu` text NOT NULL,  
  `date_ajout` datetime NOT NULL,  
  `date_modif` datetime NOT NULL,  
  PRIMARY KEY (`id`)  
);
```

Concernant l'organisation des classes, nous allons suivre la même structure que pour les personnages, à savoir :

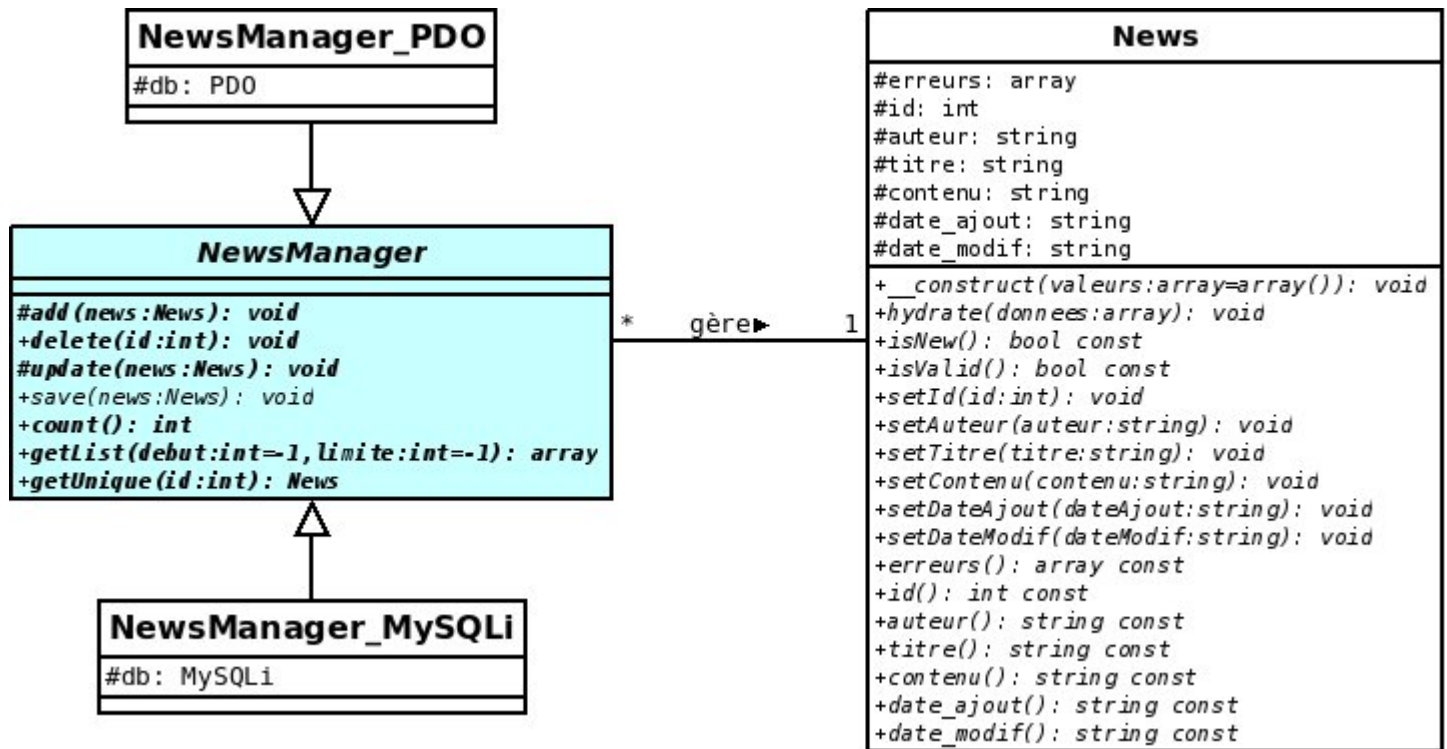
- Une classe News qui contiendra les champs sous forme d'attributs. Son rôle sera de représenter une news ;
- Une classe NewsManager qui gèrera les news. C'est elle qui interagira avec la BDD.

Cependant, je voudrais vous enseigner quelque chose de nouveau. Je voudrais que la classe NewsManager ne soit pas dépendante

de PDO. On a vu dans le chapitre précédent que l'injection de dépendances pouvait être intéressante mais nous compliquait trop la tâche concernant l'adaptation des DAO. Nous allons donc créer une classe par bibliothèque.

Cependant, nos classes doivent suivre le même modèle de base : on aura donc notre classe NewsManager qui sera abstraite, et chaque classe héritera d'elle (NewsManager_PDO, NewsManager_MySQLi, etc.). Ainsi, si nous voulons utiliser une autre bibliothèque pour accéder à la BDD, nous n'aurons qu'à créer une nouvelle classe et nous conserverons l'ancien code, toujours prêt à être utilisé.

Voici un petit diagramme représentant notre module :



Notez la présence de méthodes portant le même nom que les attributs. Ce sont tout simplement des accesseurs.

Je vous ai donné le plan du module, il ne vous reste plus qu'à implémenter les méthodes, c'est le plus facile. Allez, au boulot ! 😊

[Voir le résultat que vous devez obtenir](#)

Correction

Commençons par le cœur du script : les classes. Je vous avais mâché le travail, ça n'a pas dû être bien difficile. 😊

Pour une raison d'organisation, j'ai décidé de placer les 4 classes dans un dossier lib.

Code : PHP - News.class.php

```

<?php
/**
 * Classe représentant une news, créée à l'occasion d'un TP du tutoriel « La programmation
 * @author Victor T.
 * @version 2.0
 */
class News
{
    protected $erreurs = array(),
               $id,
               $auteur,
               $titre,
               $contenu,
               $date_ajout,
               $date_modif;

    /**
     * Constantes relatives aux erreurs possibles rencontrées lors de l'exécution de la classe
     */
    const AUTEUR_INVALIDE = 1;
    const TITRE_INVALIDE = 2;
    const CONTENU_INVALIDE = 3;

    /**
     * Constructeur de la classe qui assigne les données spécifiées en paramètre aux attributs
     * @param $valeurs array Les valeurs à assigner
     * @return void
     */
    public function __construct($valeurs = array())
    {
        if (!empty($valeurs)) // Si on a spécifié des valeurs, alors on hydrate l'objet
            $this->hydrate($valeurs);
    }

    /**
     * Méthode assignant les valeurs spécifiées aux attributs correspondant
     * @param $donnees array Les données à assigner
     * @return void
     */
    public function hydrate($donnees)
    {
        foreach ($donnees as $attribut => $valeur)
        {
            $methode = 'set'.str_replace(' ', '', ucwords(str_replace('_', ' ', $attribut)));

            if (is_callable(array($this, $methode)))
            {
                $this->$methode($valeur);
            }
        }
    }

    /**
     * Méthode permettant de savoir si la news est nouvelle
     * @return bool
     */
    public function isNew()
    {
        return empty($this->id);
    }

    /**
     * Méthode permettant de savoir si la news est valide
     * @return bool
     */
    public function isValid()
    {

```

```

<?php
abstract class NewsManager
{
    /**
     * Méthode permettant d'ajouter une news
     * @param $news News La news à ajouter
     * @return void
     */
    abstract protected function add(News $news);

    /**
     * Méthode renvoyant le nombre de news total
     * @return int
     */
    abstract public function count();

    /**
     * Méthode permettant de supprimer une news
     * @param $id int L'identifiant de la news à supprimer
     * @return void
     */
    abstract public function delete($id);

    /**
     * Méthode retournant une liste de news demandée
     * @param $debut int La première news à sélectionner
     * @param $limite int Le nombre de news à sélectionner
     * @return array La liste des news. Chaque entrée est une instance de News.
     */
    abstract public function getList($debut = -1, $limite = -1);

    /**
     * Méthode retournant une news précise
     * @param $id int L'identifiant de la news à récupérer
     * @return News La news demandée
     */
    abstract public function getUnique($id);

    /**
     * Méthode permettant d'enregistrer une news
     * @param $news News la news à enregistrer
     * @see self::add()
     * @see self::modify()
     * @return void
     */
    public function save(News $news)
    {
        if ($news->isValid())
        {
            $news->isNew() ? $this->add($news) : $this->update($news);
        }
        else
        {
            throw new RuntimeException('La news doit être valide pour être enregistrée');
        }
    }

    /**
     * Méthode permettant de modifier une news
     * @param $news news la news à modifier
     * @return void
     */
    abstract protected function update(News $news);
}

```



```

<?php
class NewsManager_PDO extends NewsManager
{
    /**
     * Attribut contenant l'instance représentant la BDD
     * @type PDO
     */
    protected $db;

    /**
     * Constructeur étant chargé d'enregistrer l'instance de PDO dans l'attribut $db
     * @param $db PDO Le DAO
     * @return void
     */
    public function __construct(PDO $db)
    {
        $this->db = $db;
    }

    /**
     * @see NewsManager::add()
     */
    protected function add(News $news)
    {
        $requete = $this->db->prepare('INSERT INTO news SET auteur = :auteur, titre

        $requete->bindValue(':titre', $news->titre());
        $requete->bindValue(':auteur', $news->auteur());
        $requete->bindValue(':contenu', $news->contenu());

        $requete->execute();
    }

    /**
     * @see NewsManager::count()
     */
    public function count()
    {
        return $this->db->query('SELECT COUNT(*) FROM news')->fetchColumn();
    }

    /**
     * @see NewsManager::delete()
     */
    public function delete($id)
    {
        $requete = $this->db->prepare('DELETE FROM news WHERE id = :id');

        $requete->bindValue(':id', $id, PDO::PARAM_INT);

        $requete->execute();
    }

    /**
     * @see NewsManager::getList()
     */
    public function getList($debut = -1, $limite = -1)
    {
        $listeNews = array();

        $sql = 'SELECT id, auteur, titre, contenu, DATE_FORMAT (date_ajout, \'le %d

        if ($debut != -1 || $limite != -1)
            $sql .= ' LIMIT ' . (int) $debut . ', ' . (int) $limite;

        $requete = $this->db->query($sql);

        while ($news = $requete->fetch(PDO::FETCH_ASSOC))

```



```

<?php
class NewsManager_MySQLi extends NewsManager
{
    /**
     * Attribut contenant l'instance représentant la BDD
     * @type MySQLi
     */
    protected $db;

    /**
     * Constructeur étant chargé d'enregistrer l'instance de MySQLi dans l'attribut
     * @param $db MySQLi Le DAO
     * @return void
     */
    public function __construct(MySQLi $db)
    {
        $this->db = $db;
    }

    /**
     * @see NewsManager::add()
     */
    protected function add(News $news)
    {
        $requete = $this->db->prepare('INSERT INTO news SET auteur = ?, titre = ?, c
        $requete->bind_param('sss', $news->auteur(), $news->titre(), $news->contenu
        $requete->execute();
    }

    /**
     * @see NewsManager::count()
     */
    public function count()
    {
        return $this->db->query('SELECT id FROM news')->num_rows;
    }

    /**
     * @see NewsManager::delete()
     */
    public function delete($id)
    {
        $id = (int) $id;

        $requete = $this->db->prepare('DELETE FROM news WHERE id = ?');

        $requete->bind_param('i', $id);

        $requete->execute();
    }

    /**
     * @see NewsManager::getList()
     */
    public function getList($debut = -1, $limite = -1)
    {
        $listeNews = array();

        $sql = 'SELECT id, auteur, titre, contenu, DATE_FORMAT (date_ajout, \'le %d
        if ($debut != -1 || $limite != -1)
            $sql .= ' LIMIT ' . (int) $debut . ', ' . (int) $limite;

        $requete = $this->db->query($sql);

        while ($news = $requete->fetch_assoc())

```

Pour accéder aux instances de PDO et MySQLi, on va s'aider du design pattern factory. Veuillez donc créer une simple classe DBFactory.

Code : PHP - DBFactory.class.php

```
<?php
class DBFactory
{
    public static function getMysqlConnexionWithPDO()
    {
        $db = new PDO('mysql:host=localhost;dbname=news', 'root', '');
        $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        return $db;
    }

    public static function getMysqlConnexionWithMySQLi()
    {
        return new MySQLi('localhost', 'root', '', 'news');
    }
}
```

Nous allons créer deux pages : index.php qui sera accessible au grand public et listera les news, ainsi que admin.php qui nous permettra de gérer les news. Dans ces deux pages, nous aurons besoin d'un autoload. Nous allons donc créer un fichier autoload.inc.php dans le dossier lib qui contiendra notre autoload. Il s'agit d'un simple fichier, voyez par vous-mêmes :

Code : PHP - autoload.inc.php

```
<?php
function autoload ($classname)
{
    if (file_exists ($file = dirname (__FILE__) . '/' . $classname . '.class.php'))
        require $file;
}

spl_autoload_register ('autoload');
```

Maintenant que nous avons créé la partie interne, nous allons nous occuper des pages qui s'afficheront devant vos yeux. Il s'agit bien entendu de la partie la plus facile, le pire est derrière nous. 😊

Commençons par la page d'administration :

Code : PHP - admin.php

```

<?php
    require 'lib/autoload.inc.php';

    $db = DBFactory::getMySQLConnexionWithPDO();
    $manager = new NewsManager_PDO($db);

    if (isset ($_GET['modifier']))
        $news = $manager->getUnique ((int) $_GET['modifier']);

    if (isset ($_GET['supprimer']))
    {
        $manager->delete((int) $_GET['supprimer']);
        $message = 'La news a bien été supprimée !';
    }

    if (isset ($_POST['auteur']))
    {
        $news = new News (
            array (
                'auteur' => $_POST['auteur'],
                'titre' => $_POST['titre'],
                'contenu' => $_POST['contenu']
            )
        );

        if (isset ($_POST['id']))
            $news->setId($_POST['id']);

        if ($news->isValid())
        {
            $manager->save($news);

            $message = $news->isNew() ? 'La news a bien été ajoutée !' : 'La news a bien
        }
        else
            $erreurs = $news->erreurs();
    }
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
    <head>
        <title>Administration</title>
        <meta http-equiv="Content-type" content="text/html; charset=iso-8859-1" />

        <style type="text/css">
            table, td {
                border: 1px solid black;
            }

            table {
                margin:auto;
                text-align: center;
                border-collapse: collapse;
            }

            td {
                padding: 3px;
            }
        </style>
    </head>

    <body>
        <p><a href=".">Accéder à l'accueil du site</a></p>

        <form action="admin.php" method="post">
            <p style="text-align: center">
<?php
    if (isset ($message))

```

Et enfin, la partie visible à tous vos visiteurs :

Code : PHP - index.php

```
<?php
    require 'lib/autoload.inc.php';

    $db = DBFactory::getMySQLConnexionWithPDO();
    $manager = new NewsManager_PDO($db);
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr">
    <head>
        <title>Accueil du site</title>
        <meta http-equiv="Content-type" content="text/html; charset=iso-8859-1" />
    </head>

    <body>
        <p><a href="admin.php">Accéder à l'espace d'administration</a></p>
    <?php
        if (isset ($_GET['id']))
        {
            $news = $manager->getUnique((int) $_GET['id']);

            echo '<p>Par <em>', $news->auteur(), '</em>', ' ', $news->date_ajout(), '</p>', "\n";
            echo '<h2>', $news->titre(), '</h2>', "\n";
            echo '<p>', nl2br($news->contenu()), '</p>', "\n";

            if ($news->date_ajout() != $news->date_modif())
                echo '<p style="text-align: right;"><small><em>Modifiée ', $news->date_modif
        }

        else
        {
            echo '<h2 style="text-align:center">Liste des 5 dernières news</h2>';

            foreach ($manager->getList(0, 5) as $news)
            {
                if (strlen($news->contenu()) <= 200)
                    $contenu = $news->contenu();

                else
                {
                    $debut = substr($news->contenu(), 0, 200);
                    $debut = substr($debut, 0, strrpos($debut, ' ')) . '...';

                    $contenu = $debut;
                }

                echo '<h4><a href="?id=', $news->id(), '>', $news->titre(), '</a></h4>', "\n";
                echo '<p>', nl2br($contenu), '</p>';
            }
        }
    </body>
</html>
```

Vous devriez maintenant mieux comprendre comment structurer une application de façon orienté objet. Pour approfondir ce TP, je vous conseille d'implémenter un module de commentaires. Basez-vous sur le même modèle : une classe représentant un commentaire (Comment), et une autre classe s'occupant de la gestion de ceux-ci (CommentsManager).

Nous allons terminer ce tutoriel sur ce TP. Vous êtes désormais prêts à être lâchés dans la nature et à commencer à créer votre site. Au début, ça ne sera pas facile, mais vous pouvez toujours demander des conseils sur les forums. 😊

Partie 4 : Annexes

Voici ici quelques chapitres vous présentant quelques notions que je n'ai pas pu glisser dans le cours au risque de compliquer celui-ci inutilement. Au début de chaque chapitre seront précisés les pré-requis afin de pouvoir suivre sans difficulté. 😊

L'opérateur instanceof

Je vais ici vous présenter l'opérateur instanceof. Ce sera un court chapitre car cette notion n'est pas bien difficile. Il faut juste posséder quelques pré-requis.

En voici la liste :

- Bien maîtriser les notions de classe, d'objet et d'instance ;
 - Bien maîtriser le concept de l'héritage (si vous ne maîtrisez pas bien la résolution statique à la volée ce n'est pas bien important) ;
 - Savoir ce qu'est une interface et savoir s'en servir.
-

Présentation de l'opérateur

L'opérateur instanceof permet de vérifier si tel objet est une instance de telle classe. C'est un opérateur qui s'utilise dans une condition. Ainsi, on pourra créer des conditions comme « si \$monObjet est une instance de MaClasse, alors... ».

Maintenant nous allons voir comment construire notre condition. À gauche de notre opérateur, on va y placer notre objet. À droite de notre opérateur, nous allons placer, comme vous vous en doutez sûrement, le nom de la classe.

Exemple :

Code : PHP

```
<?php
class A { }
class B { }

$monObjet = new A;

if ($monObjet instanceof A) // Si $monObjet est une instance de A
    echo '$monObjet est une instance de A';
else
    echo '$monObjet n\'est pas une instance de A';

if ($monObjet instanceof B) // Si $monObjet est une instance de B
    echo '$monObjet est une instance de B';
else
    echo '$monObjet n\'est pas une instance de B';

?>
```

Bref, je pense que vous avez compris le principe. 😊

Si votre version de PHP est ultérieure à la version 5.1, alors aucune erreur fatale ne sera générée si vous utilisez l'opérateur instanceof en spécifiant une classe qui n'a pas été déclarée. La condition renverra tout simplement false. 😊

Il y a cependant plusieurs façons de faire, et quelques astuces (c'est d'ailleurs pour toutes les présenter que j'ai créé ce chapitre, car s'il se limitait à ce seul exemple... 🤖).

Parmi ces façons, il y en a une qui consiste à placer le nom de la classe à laquelle on veut vérifier que tel objet est une instance dans une variable sous forme de chaîne de caractères. Exemple :

Code : PHP

```
<?php
class A { }
class B { }

$monObjet = new A;

$classeA = 'A';
$classeB = 'B';

if ($monObjet instanceof $classeA)
    echo '$monObjet est une instance de ', $classeA;
else
    echo '$monObjet n\'est pas une instance de ', $classeA;

if ($monObjet instanceof $classeB)
    echo '$monObjet est une instance de ', $classeB;
else
    echo '$monObjet n\'est pas une instance de ', $classeB;

?>
```

Attention ! Vous ne pouvez spécifier le nom de la classe entre apostrophes ou guillemets directement dans la condition ! Vous devez obligatoirement passer par une variable. Si vous le faites directement, vous obtiendrez une belle erreur d'analyse.

Encore une autre façon d'utiliser cet opérateur est de spécifier un autre objet à la place du nom de la classe. La condition renverra true si les deux objets sont des instances de la même classe. Exemple :

Code : PHP

```
<?php
class A { }
class B { }

$a = new A;
$b = new A;
$c = new B;

if ($a instanceof $b)
    echo '$a et $b sont des instances de la même classe';
else
    echo '$a et $b ne sont pas des instances de la même classe';

if ($a instanceof $c)
    echo '$a et $c sont des instances de la même classe';
else
    echo '$a et $c ne sont pas des instances de la même classe';

?>
```

Et voilà. Vous avez les trois méthodes possibles pour utiliser cet opérateur. Pourtant, il existe encore quelques effets que peut produire instanceof. Continuons donc ce chapitre tranquillement. 😊

instanceof et l'héritage

Voici le retour de l'héritage. En effet, instanceof a un comportement bien particulier avec les classes qui héritent entre elles. Voici donc quels sont ces effets.

Vous vous souvenez sans doute (enfin j'espère 😊) de la première façon d'utiliser l'opérateur. Voici une révélation : la condition renvoie true si la classe spécifiée est une classe parente de la classe instanciée par l'objet spécifié. Exemple :

Code : PHP

```
<?php
class A { }
class B extends A { }
class C extends B { }

$b = new B;

if ($b instanceof A)
    echo '$b est une instance de A ou $b instancie une classe qui est une fille de A'
else
    echo '$b n\'est pas une instance de A et $b instancie une classe qui n\'est pas v

if ($b instanceof C)
    echo '$b est une instance de C ou $b instancie une classe qui est une fille de C'
else
    echo '$b n\'est pas une instance de C et $b instancie une classe qui n\'est pas v
```

Voilà, j'espère que vous avez compris le principe car celui-ci est le même avec les deuxième et troisième méthodes.

Nous allons donc maintenant terminer ce chapitre avec une dernière partie concernant les réactions de l'opérateur avec les interfaces. Ce sera un mix des deux premières parties, donc si vous êtes perdus, relisez bien tout (eh oui, j'espère que vous n'avez pas oublié l'héritage entre interfaces 🤔).

instanceof et les interfaces

Voyons maintenant les effets produits par l'opérateur avec les interfaces.

Hein ? Comment ça ? Je comprends pas... Comment peut-on vérifier qu'un objet soit une instance d'une interface sachant que c'est impossible ? 🤔

Comme vous le dites si bien, c'est impossible de créer une instance d'une interface (au même titre que de créer une instance d'une classe abstraite, ce qu'est à peu près une interface). L'opérateur va donc renvoyer true si tel objet instancie une classe implémentant telle interface.

Voici un exemple :

Code : PHP

```
<?php
interface iA { }
class A implements iA { }
class B { }

$a = new A;
$b = new B;

if ($a instanceof iA)
    echo 'Si iA est une classe, alors $a est une instance de iA ou $a instancie une c
else
    echo 'Si iA est une classe, alors $a n\'est pas une instance de iA et $a n\'insta

if ($b instanceof iA)
    echo 'Si iA est une classe, alors $b est une instance de iA ou $b instancie une c
else
    echo 'Si iA est une classe, alors $b n\'est pas une instance de iA et $b n\'insta
```

Je pense que ce code se passe de commentaires, les valeurs affichées détaillant assez bien je pense. 🤔

Après avoir vu l'utilisation de l'opérateur avec les interfaces, nous allons voir comment il réagit lorsqu'on lui passe en paramètre une interface qui est héritée par une autre interface qui est implémentée par une classe qui est instanciée. Vous voyez à peu près la chose ? Je vais vous le faire en PHP au cas où vous n'avez pas tout suivi. 🤔

Code : PHP

```
<?php
interface iParent { }
interface iFille extends iParent { }
class A implements iFille { }

$a = new A;

if ($a instanceof iParent)
    echo 'Si iParent est une classe, alors $a est une instance de iParent ou $a insta
else
    echo 'Si iParent est une classe, alors $a n\'est pas une instance de iParent et $
```

Vous savez maintenant tous les comportements que peut adopter cet opérateur, et tous les effets qu'il peut produire (tout est écrit dans le précédent code).

Cet opérateur n'est pas énormément utilisé mais il est tout de même intéressant de connaître son existence. 🤔

Ce tutoriel est maintenant terminé. J'espère qu'il vous aura plu et apporté beaucoup de connaissances. 😊

Dès à présent, vous êtes prêts à être lâchés dans la nature. En effet, vous savez tout à propos de la POO en PHP. Cependant, un petit problème se pose : vous ne savez pas programmer un site web entièrement orienté objet. Pour y parvenir, je ne peux que vous recommander de pratiquer en utilisant des frameworks orientés objets. Il en existe un bon nombre, et celui que je vous conseille (c'est d'ailleurs lui que j'utilise, et que le SdZ utilise aussi par la même occasion) est [Symfony](#). Vous avez de la chance, car il existe un excellent tutoriel sur le site du projet : j'ai nommé le [tutoriel Jobeet](#). C'est d'ailleurs le tutoriel de référence. 😊

Vous verrez qu'à force de l'utiliser, vous comprendrez les mécanismes de la POO ainsi que sa puissance. En explorant la partie interne de ce framework, vous comprendrez comment celui-ci fonctionne et serez capables de réaliser vous-mêmes un site web orienté objet ! 😊