INTRODUCTION À LA POO

Alors ça y est, vous avez décidé de vous lancer dans la POO en PHP ? Sage décision ! Nous allons donc plonger dans ce vaste domaine par une introduction à cette nouvelle façon de penser : qu'est-ce que la POO ? En quoi ça consiste ? En quoi est-ce si différent de la méthode que vous employez pour développer votre site web ? Tant de questions auxquelles je vais répondre.

Cependant, puisque je sais que vous avez hâte de commencer, nous allons entamer sérieusement les choses en créant notre première **classe** dès la fin de ce chapitre. Vous commencerez ainsi vos premiers pas dans la POO en PHP !

Qu'est-ce que la POO ?

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 » qui 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. Ce sont tous des objets. En programmation, les objets sont sensiblement la même chose.

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. En tout cas, le principe est là.

Vous savez désormais qu'on peut avoir des objets dans une application. Mais d'où sortent-ils ? Dans la vie réelle, un objet ne sort pas de nulle part. En effet, chaque objet est défini selon des caractéristiques et un plan bien précis. En POO, ces informations sont contenues dans ce qu'on appelle des **classes**.

Définition d'une classe

Comme je viens de le dire, les classes contiennent la définition des objets que l'on va créer par la suite. Prenons l'exemple le plus simple du monde : les gâteaux et leur moule. Le moule est unique. Il peut produire une quantité infinie de gâteaux. Dans ce cas-là, les gâteaux sont les *objets* et le moule est la *classe* : le moule va définir la forme du gâteau. La classe contient donc 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 ?

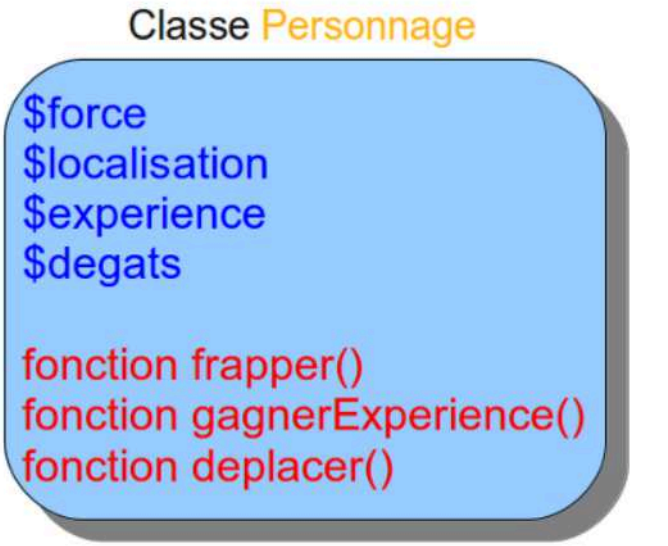
Une classe est une entité regroupant des variables et des fonctions. Chacune de ces fonctions aura accès aux variables de cette entité. Dans le cas du personnage, nous aurons une fonction `frapper()`. Cette fonction devra simplement modifier la variable `$degats` du personnage en fonction de la variable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

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 est un objet.

Exemple : création d'une classe

Nous allons 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 possédera ainsi toutes ces variables et fonctions. Voici donc cette fameuse classe à la figure suivante.



Le schéma de notre 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 et une méthode désigne une fonction.

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. 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**. Pour l'instant, ceci peut sans doute vous paraître abstrait, mais nous y reviendrons.

Bon, je pense que j'ai assez parlé, commençons par créer notre première classe !

Créer une classe

Syntaxe de base

Le but de cette section va être de traduire la figure précédente en code PHP. Avant cela, je vais vous donner la syntaxe de base de toute classe en PHP :

```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. Heureusement, elle est simple.

Ce qu'on vient de faire est donc de créer le moule, le plan qui définira nos objets. On verra dans le prochain chapitre comment utiliser ce plan afin de créer un objet. Pour l'instant, contentons-nous de construire ce plan 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**.

Visibilité d'un attribut ou d'une méthode

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

Le premier, `public`, est le plus simple. Si un attribut ou une méthode est `public`, alors on pourra y avoir accès depuis n'importe où, depuis l'intérieur de l'objet (dans les méthodes qu'on a créées), comme depuis l'extérieur. Je m'explique. Quand on crée 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 n'aura accès aux attributs et méthodes *seulement* 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 pouvez 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 :

```php
<?php
class Personnage
{
  private $_force;        // La force du personnage
  private $_localisation; // Sa localisation
  private $_experience;   // Son expérience
  private $_degats;       // Ses dégâts
}
```

Vous pouvez constater que chaque attribut est précédé d'un underscore (« _ »). Ceci est une notation qu'il est préférable de respecter (il s'agit de la notation PEAR) qui dit que chaque nom d'élément privé (ici il s'agit d'attributs, mais nous verrons plus tard qu'il peut aussi s'agir de méthodes) doit être précédé d'un underscore.

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

```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 scalaire statique. Par conséquent, leur valeur ne peut par exemple pas être issue d'un appel à une fonction (`private $_attribut = strlen('azerty')`) ou d'une variable, superglobale ou non (`private $_attribut = $_SERVER['REQUEST_URI']`). Si votre version de PHP est antérieure à la 5.6, vous ne pouvez spécifier que des valeurs statiques, ce qui rend impossible l'assignation du résultat d'une opération. Par exemple, vous ne pouvez pas faire de `private $_attribut = 1 + 1` ou bien `private $_attribut = 'Hello ' . 'world !'` .

Création de méthodes

Pour la déclaration de méthodes, il suffit de faire précéder le mot-clé `function` à 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 mettrez souvent en `public` (à moins que vous teniez absolument à ce que l'utilisateur ne puisse pas appeler cette méthode, par exemple s'il s'agit d'une fonction qui simplifie certaines tâches sur l'objet mais qui ne doit pas être appelée n'importe comment).

```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 sa localisation).
  {

  }

  public function frapper() // Une méthode qui frappera un personnage (suivant la force qu'il a).
  {

  }

  public function gagnerExperience() // Une méthode augmentant l'attribut $experience du personnage.
  {

  }
}
```

Et voilà. 

De même que l'underscore précédant les noms d'éléments privés, vous pouvez remarquer que le nom des classes commence par une majuscule. Il s'agit aussi du respect de la notation PEAR.

En résumé

- Une classe, c'est un ensemble de variables et de fonctions (attributs et méthodes).
- Un objet, c'est une *instance* de la classe pour pouvoir l'utiliser.
- Tous vos attributs doivent être privés. Pour les méthodes, peu importe leur visibilité. C'est ce qu'on appelle le principe d'encapsulation.
- 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.

☐ J'ai terminé ce chapitre et je passe au suivant

UTILISER LA CLASSE

Une classe, c'est bien beau mais, au même titre qu'un plan de construction pour une maison, si on ne sait pas comment se servir de notre plan pour construire notre maison, cela ne sert à rien ! Nous verrons donc ainsi comment se servir de notre **classe**, notre modèle de base, afin de créer des **objets** et pouvoir s'en servir.

Ce chapitre sera aussi l'occasion d'approfondir un peu le développement de nos classes : actuellement, la classe que l'on a créée contient des méthodes vides, chose un peu inutile vous avouerez. Pas de panique, on va s'occuper de tout ça !

Créer et manipuler un objet

Créer un objet

Nous allons voir comment créer un objet, c'est-à-dire que nous allons utiliser notre classe 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 :

```php
<?php
$perso = new Personnage;
```

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

Appeler les méthodes de l'objet

Pour appeler une méthode d'un objet, il va falloir utiliser un opérateur : il s'agit de l'opérateur -> (une flèche composée d'un tiret suivi d'un chevron fermant). Celui-ci s'utilise de la manière suivante. À gauche de cet opérateur, on place **l'objet** que l'on veut utiliser. Dans l'exemple pris juste au-dessus, cet objet aurait été `$perso`. À droite de l'opérateur, on spécifie le nom de la méthode que l'on veut invoquer.

Et puisqu'un exemple vaut mieux qu'un long discours…

```php
<?php
// Nous créons une classe « Personnage ».
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

  // Nous déclarons une méthode dont le seul but est d'afficher un texte.
  public function parler()
  {
    echo 'Je suis un personnage !';
  }
}

$perso = new Personnage;
$perso->parler();
```

La ligne 18 signifie donc « va chercher l'objet `$perso`, et invoque la méthode `parler()` sur cet objet ». Notez donc bien quelque part l'utilisation de cet opérateur : de manière générale, il sert à accéder à un élément de la classe. Ici, on s'en est servi pour atteindre une méthode, mais nous verrons plus tard qu'il nous permettra aussi d'atteindre un attribut.

Accéder à un élément depuis la classe

Je viens de vous faire créer un objet, et vous êtes maintenant capables d'appeler une méthode. Cependant, le contenu de cette dernière était assez simpliste : elle ne faisait qu'afficher un message. Ici, nous allons voir comment une méthode peut accéder aux attributs de l'objet.

Lorsque vous avez un objet, vous savez que vous pouvez invoquer des méthodes grâce à l'opérateur « `->` ». Je vous ai par ailleurs dit que c'était également grâce à cet opérateur qu'on accédait aux attributs de la classe. Cependant, rappelez-vous : nos attributs sont privés. Par conséquent, ce code lèvera une erreur fatale :

```php
<?php
class Personnage
{
  private $_force;
  private $_experience;
  private $_degats;
}

$perso = new Personnage;
$perso->_experience = $perso->_experience + 1; // Une erreur fatale est levée suite à cette instruction.
```

Ici, on essaye d'accéder à un attribut privé hors de la classe. Ceci est interdit, donc PHP lève une erreur. Dans notre exemple (qui essaye en vain d'augmenter de 1 l'expérience du personnage), il faudra demander à la classe d'augmenter l'expérience. Pour cela, nous allons écrire une méthode `gagnerExperience()` :

```php
<?php
class Personnage
{
  private $_experience;

  public function gagnerExperience()
  {
    // Cette méthode doit ajouter 1 à l'expérience du personnage.
  }
}

$perso = new Personnage;
$perso->afficherExperience();
```

Commentons la ligne 8. Vous avez sans doute reconnu la structure `echo` ayant pour rôle d'afficher une valeur. Ensuite, vous reconnaissez la variable `$this` dont je vous ai parlé : elle représente l'objet que nous sommes en train d'utiliser. Ainsi, dans ce script, les variables `$this` et `$perso` représentent le même objet. L'instruction surlignée veut donc dire : « Affiche-moi cette valeur : dans l'objet utilisé (donc `$perso`), donne-moi la valeur de l'attribut `$_experience`. »

Ainsi, je vais vous demander d'écrire la méthode `gagnerExperience()`. Cette méthode devra ajouter 1 à l'attribut `$_experience`. Je suis sûr que vous pouvez y arriver !

Voici la correction :

```php
<?php
class Personnage
{
  private $_experience = 50;

  public function afficherExperience()
  {
    echo $this->_experience;
  }

  public function gagnerExperience()
  {
    // On ajoute 1 à notre attribut $_experience.
    $this->_experience = $this->_experience + 1;
  }
}

$perso = new Personnage;
$perso->gagnerExperience();   // On gagne de l'expérience.
$perso->afficherExperience(); // On affiche la nouvelle valeur de l'attribut.
```

Implémenter d'autres méthodes

Nous avons créé notre classe Personnage et déjà une méthode, `gagnerExperience()`. J'aimerais qu'on en implémente une autre : la méthode `frapper()`, qui devra infliger des dégâts à un personnage.

Pour partir sur une base commune, nous allons travailler sur cette classe :

```php
<?php
class Personnage
{
  private $_degats = 0; // Les dégâts du personnage.
  private $_experience = 0; // L'expérience du personnage.
  private $_force = 20; // La force du personnage (plus elle est grande, plus l'attaque est puissante).

  public function gagnerExperience()
  {
    // On ajoute 1 à notre attribut $_experience.
    $this->_experience = $this->_experience + 1;
  }
}
```

Commençons par écrire notre méthode `frapper()`. Cette méthode doit accepter un argument : le personnage à frapper. La méthode aura juste pour rôle d'augmenter les dégâts du personnage passé en paramètre.

Pour vous aider à visualiser le contenu de la méthode, imaginez votre code manipulant des objets. Il doit ressembler à ceci :

```php
<?php
// On crée deux personnages
$perso1 = new Personnage;
$perso2 = new Personnage;

// Ensuite, on veut que le personnage n°1 frappe le personnage n°2.
$perso1->frapper($perso2);
```

Pour résumer :

- la méthode `frapper()` demande un argument : le personnage à frapper ;
- cette méthode augmente les dégâts du personnage à frapper en fonction de la force du personnage qui frappe.

On pourrait donc imaginer une classe ressemblant à ceci :

```php
<?php
class Personnage
{
  private $_degats; // Les dégâts du personnage.
  private $_experience; // L'expérience du personnage.
  private $_force; // La force du personnage (plus elle est grande, plus l'attaque est puissante).

  public function frapper($persoAFrapper)
  {
    $persoAFrapper->_degats += $this->_force;
  }

  public function gagnerExperience()
  {
    // On ajoute 1 à notre attribut $_experience.
    $this->_experience = $this->_experience + 1;
  }
}
```

Commentons ensemble le contenu de cette méthode `frapper()`. Celle-ci comporte une instruction composée de deux parties :

1. La première consiste à dire à PHP que l'on veut assigner une nouvelle valeur à l'attribut `$_degats` du personnage à frapper.
2. La seconde partie consiste à donner à PHP la valeur que l'on veut assigner. Ici, nous voyons que cette valeur est atteinte par `$this->_force`.

Maintenant, grosse question : la variable `$this` fait-elle référence au personnage qui frappe ou au personnage frappé ? Pour répondre à cette question, il faut savoir sur quel objet est appelée la méthode. En effet, souvenez-vous que `$this` est une variable représentant l'objet à partir duquel on a appelé la méthode. Dans notre cas, on a appelé la méthode `frapper()` à partir du personnage qui frappe, donc `$this` représente le personnage qui frappe.

L'instruction contenue dans la méthode signifie donc : « Ajoute la valeur de la force du personnage qui frappe à l'attribut `$_degats` du personnage frappé. »

Maintenant, nous pouvons créer une sorte de petite mise en scène qui fait interagir nos personnages. Par exemple, nous pouvons créer un script qui fait combattre les personnages. Le personnage 1 frapperait le

personnage 2 puis gagnerait de l'expérience, puis le personnage 2 frapperait le personnage 1 et gagnerait de l'expérience. Procédez étape par étape :

- créez deux personnages ;
- faites frapper le personnage 1 ;
- faites gagner de l'expérience au personnage 1 ;
- faites frapper le personnage 2 ;
- faites gagner de l'expérience au personnage 2.

Ce script n'est qu'une suite d'appels de méthodes. Chaque puce (sauf la première) correspond à l'appel d'une méthode, ce que vous savez faire. En conclusion, vous êtes aptes à créer ce petit script ! Voici la correction :

```php
<?php
$perso1 = new Personnage; // Un premier personnage
$perso2 = new Personnage; // Un second personnage

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

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

Cependant, un petit problème se pose. Puisque, à la base, les deux personnages ont le même niveau de dégâts, la même expérience et la même force, ils seront à la fin toujours égaux. Pour pallier ce problème, il faudrait pouvoir assigner des valeurs spécifiques aux deux personnages, afin que le combat puisse les différencier. Or, vous ne pouvez pas accéder aux attributs en-dehors de la classe ! Pour savoir comment résoudre ce problème, je vais vous apprendre deux nouveaux mots : **accesseur** et **mutateur**. Mais avant, j'aimerais faire une petite parenthèse.

Exiger des objets en paramètre

Reprenons l'exemple du code auquel nous sommes arrivés et concentrons-nous sur la méthode `frapper()`. Celle-ci accepte un argument : un personnage à frapper. Cependant, qu'est-ce qui vous garantit qu'on passe effectivement un personnage à frapper ? On pourrait très bien passer un argument complètement différent, comme un nombre par exemple :

```php
<?php
$perso = new Personnage;
$perso->frapper(42);
```

Et là, qu'est-ce qui se passe ? Une erreur est générée car, à l'intérieur de la méthode `frapper()`, nous essayons d'appeler une méthode sur le paramètre qui n'est pas un objet. C'est comme si on avait fait ça :

```php
<?php
$persoAFrapper = 42;
$persoAFrapper->_degats += 50; // Le nombre 50 est arbitraire, il est censé représenter une force.
```

Ce qui n'a aucun sens. Il faut donc s'assurer que le paramètre passé est bien un personnage, sinon PHP arrête tout et n'exécute pas la méthode. Pour cela, il suffit d'ajouter un seul mot : le nom de la classe dont le paramètre doit être un objet. Dans notre cas, si le paramètre doit être un objet de type `Personnage`, alors il faudra ajouter le mot-clé `Personnage`, juste avant le nom du paramètre, comme ceci :

```php
<?php
class Personnage
{
  // …

  public function frapper(Personnage $persoAFrapper)
  {
    // …
  }
}
```

Grâce à ça, vous êtes sûrs que la méthode `frapper()` ne sera exécutée *que* si le paramètre passé est de type `Personnage`, sinon PHP interrompt tout le script. Vous pouvez donc appeler les méthodes de l'objet sans crainte qu'un autre type de variable soit passé en paramètre.

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 par le 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.

Les accesseurs et mutateurs

Comme vous le savez, le principe d'encapsulation nous empêche d'accéder directement aux attributs de notre objet puisqu'ils sont privés : seule la classe peut les lire et les modifier. Par conséquent, si vous voulez récupérer un attribut, il va falloir le demander à la classe, de même si vous voulez les modifier.

Accéder à un attribut : l'accesseur

À votre avis, comment peut-on faire pour récupérer la valeur d'un attribut ? La solution est simple : nous allons implémenter des méthodes dont le seul rôle sera de nous donner l'attribut qu'on leur demande ! Ces méthodes ont un nom bien spécial : ce sont des **accesseurs** (ou *getters*). Par convention, ces méthodes portent le même nom que l'attribut dont elles renvoient la valeur. Par exemple, voici la liste des accesseurs de notre classe `Personnage` :

php
```php
<?php
class Personnage
{
  private $_force;
  private $_experience;
  private $_degats;

  public function frapper(Personnage $persoAFrapper)
  {
    $persoAFrapper->_degats += $this->_force;
  }

  public function gagnerExperience()
  {
    // Ceci est un raccourci qui équivaut à écrire « $this->_experience = $this->_experience + 1 »
    // On aurait aussi pu écrire « $this->_experience += 1 »
    $this->_experience++;
  }

  // Ceci est la méthode degats() : elle se charge de renvoyer le contenu de l'attribut $_degats.
  public function degats()
  {
    return $this->_degats;
  }

  // Ceci est la méthode force() : elle se charge de renvoyer le contenu de l'attribut $_force.
  public function force()
  {
    return $this->_force;
  }

  // Ceci est la méthode experience() : elle se charge de renvoyer le contenu de l'attribut $_experience.
  public function experience()
  {
    return $this->_experience;
  }
}
```

Modifier la valeur d'un attribut : les mutateurs

Maintenant, comment cela se passe-t-il si vous voulez modifier un attribut ? Encore une fois, il va falloir que vous demandiez à la classe de le faire pour vous. Je vous rappelle que le principe d'encapsulation est là pour vous empêcher d'assigner un mauvais type de valeur à un attribut : si vous demandez à votre classe de le faire, ce risque est supprimé car la classe « contrôle » la valeur des attributs. Comme vous l'aurez peut-être deviné, ce sera par le biais de méthodes que l'on demandera à notre classe de modifier tel attribut.

La classe doit impérativement contrôler la valeur afin d'assurer son intégrité car, si elle ne le fait pas, on pourra passer n'importe quelle valeur à la classe et le principe d'encapsulation n'est plus respecté ! Ces méthodes ont aussi un nom spécial : il s'agit de **mutateurs** (ou *setters*). Ces méthodes sont de la forme `setNomDeLAttribut()`. Voici la liste des mutateurs (ajoutée à la liste des accesseurs) de notre classe `Personnage` :

```php
<?php
class Personnage
{
  private $_force;
  private $_experience;
  private $_degats;

  public function frapper(Personnage $persoAFrapper)
  {
    $persoAFrapper->_degats += $this->_force;
  }

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

  // Mutateur chargé de modifier l'attribut $_force.
  public function setForce($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;
  }

  // Mutateur chargé de modifier l'attribut $_experience.
  public function setExperience($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;
  }

  // Ceci est la méthode degats() : elle se charge de renvoyer le contenu de l'attribut $_degats.
  public function degats()
  {
    return $this->_degats;
  }

  // Ceci est la méthode force() : elle se charge de renvoyer le contenu de l'attribut $_force.
  public function force()
  {
    return $this->_force;
  }

  // Ceci est la méthode experience() : elle se charge de renvoyer le contenu de l'attribut $_experience.
  public function experience()
  {
    return $this->_experience;
  }
}
```

Voilà ce que j'avais à dire concernant ces accesseurs et mutateurs. Retenez bien ces définitions, vous les trouverez dans la plupart des classes !

Retour sur notre script de combat

Maintenant que nous avons vu ce qu'étaient des accesseurs et des mutateurs, nous pouvons améliorer notre script de combat. Pour commencer, je vais vous demander d'afficher, à la fin du script, la force, l'expérience et le niveau de dégâts de chaque personnage.

Voici la correction :

```php
<?php
$perso1 = new Personnage;  // Un premier personnage
$perso2 = new Personnage;  // Un second personnage

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

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

echo 'Le personnage 1 a ', $perso1->force(), ' de force, contrairement au personnage 2 qui a ', $perso2->force(), ' de force.<br />';
echo 'Le personnage 1 a ', $perso1->experience(), ' d\'expérience, contrairement au personnage 2 qui a ', $perso2->experience(), ' d\'expérience.<br />';
echo 'Le personnage 1 a ', $perso1->degats(), ' de dégâts, contrairement au personnage 2 qui a ', $perso2->degats(), ' de dégâts.<br />';
```

Comme nous l'avions dit, les valeurs finales des deux personnages sont identiques. Pour pallier ce problème, nous allons modifier, juste après la création des personnages, la valeur de la force et de l'expérience des deux personnages. Vous pouvez par exemple favoriser un personnage en lui donnant une plus grande force et une plus grande expérience par rapport au deuxième.

Voici la correction que je vous propose (peu importe les valeurs que vous avez choisies, l'essentiel est que vous ayez appelé les bonnes méthodes) :

```php
<?php
$perso1 = new Personnage; // Un premier personnage
$perso2 = new Personnage; // Un second personnage

$perso1->setForce(10);
$perso1->setExperience(2);

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

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

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

echo 'Le personnage 1 a ', $perso1->force(), ' de force, contrairement au personnage 2 qui a ', $perso2->force(), ' de force.<br />';
echo 'Le personnage 1 a ', $perso1->experience(), ' d\'expérience, contrairement au personnage 2 qui a ', $perso2->experience(), ' d\'expérience.<br />';
echo 'Le personnage 1 a ', $perso1->degats(), ' de dégâts, contrairement au personnage 2 qui a ', $perso2->degats(), ' de dégâts.<br />';
```

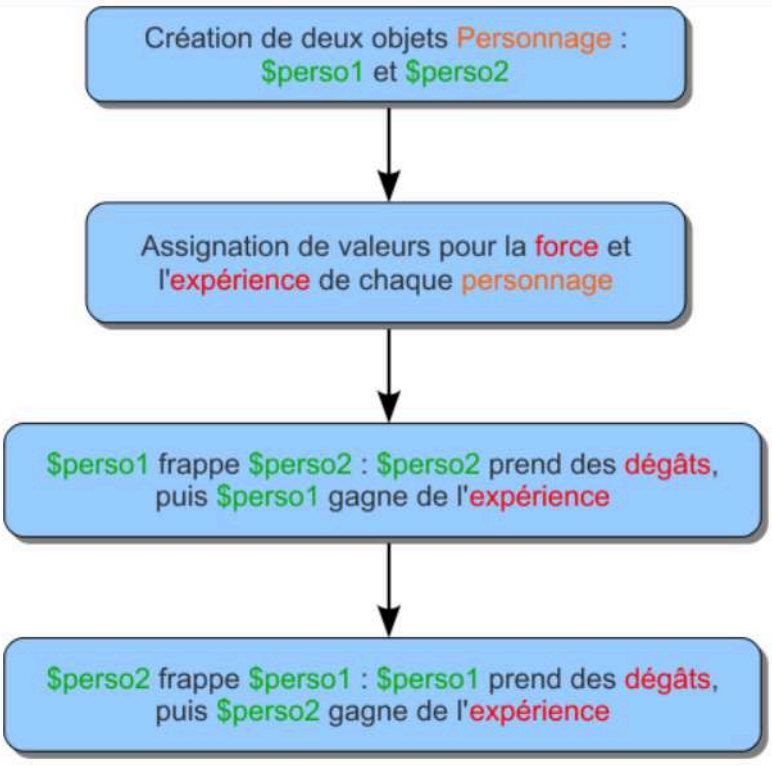
Ce qui affichera :



Résultat affiché par le script

Comme vous le voyez, à la fin, les deux personnages n'ont plus les mêmes caractéristiques !

Pour bien être sûr que vous me suiviez toujours, je vous ai fait un schéma résumant le déroulement du script.



Déroulement du script

Le constructeur

Imaginez un objet pour lequel vous avez besoin d'initialiser les attributs dès sa création, sans connaitre leurs valeurs à l'avance. Par exemple, vous souhaiteriez pouvoir spécifier la force et les dégâts d'un personnage dès sa création. Actuellement, la seule possibilité qui s'offre à vous est de modifier ces attributs manuellement, une fois l'objet créé. Il existe en PHP une méthode, appelée le **constructeur**, qui remplira ce rôle. Ce constructeur n'est autre qu'une méthode écrite dans votre classe.

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

Comme son nom l'indique, le constructeur sert à *construire* l'objet. 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 génèrera aucune erreur. Bien sûr, et comme nous l'avons vu, une classe fonctionne très bien sans constructeur, il n'est en rien obligatoire ! Si vous n'en spécifiez pas, cela revient au même que si vous en aviez écrit un vide (sans instruction à l'intérieur).

```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.
    $this->setForce($force); // Initialisation de la force.
    $this->setDegats($degats); // Initialisation des dégâts.
    $this->_experience = 1; // Initialisation de l'expérience à 1.
  }

  // Mutateur chargé de modifier l'attribut $_force.
  public function setForce($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;
  }

  // Mutateur chargé de modifier l'attribut $_degats.
  public function setDegats($degats)
  {
    if (!is_int($degats)) // S'il ne s'agit pas d'un nombre entier.
    {
      trigger_error('Le niveau de dégâts d\'un personnage doit être un nombre entier', E_USER_WARNING);
      return;
    }

    $this->_degats = $degats;
  }
}
```

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 :

```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 :



Résultat affiché par le script

Notez quelque chose d'important dans la classe : dans le constructeur, les valeurs sont initialisées en appelant les mutateurs correspondant. En effet, si on assignait directement ces valeurs avec les arguments, le principe d'encapsulation ne serait plus respecté et n'importe quel type de valeur pourrait être assigné !

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 classe ! Cependant, sachez qu'il existe certains cas particuliers qui nécessitent le constructeur en privé, mais ce n'est pas pour tout de suite.

L'auto-chargement de classes

Pour une question d'organisation, il vaut mieux créer un fichier par classe. Vous appelez votre fichier comme bon vous semble et placez votre classe dedans. Pour ma part, mes fichiers sont toujours appelés « MaClasse.php ». Ainsi, si je veux pouvoir utiliser la classe `MaClasse`, je n'aurais qu'à inclure ce fichier :

```php
<?php
require 'MaClasse.php'; // J'inclus la classe.

$objet = new MaClasse; // Puis, seulement après, je me sers de ma classe.
```

Maintenant, imaginons que vous ayez plusieurs dizaines de classes… Pas très pratique de les inclure une par une ! Vous vous retrouverez avec des dizaines d'inclusions, certaines pouvant même être inutile si vous ne vous servez pas de toutes vos classes. 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 créerez 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 :

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

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.php` existe).

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

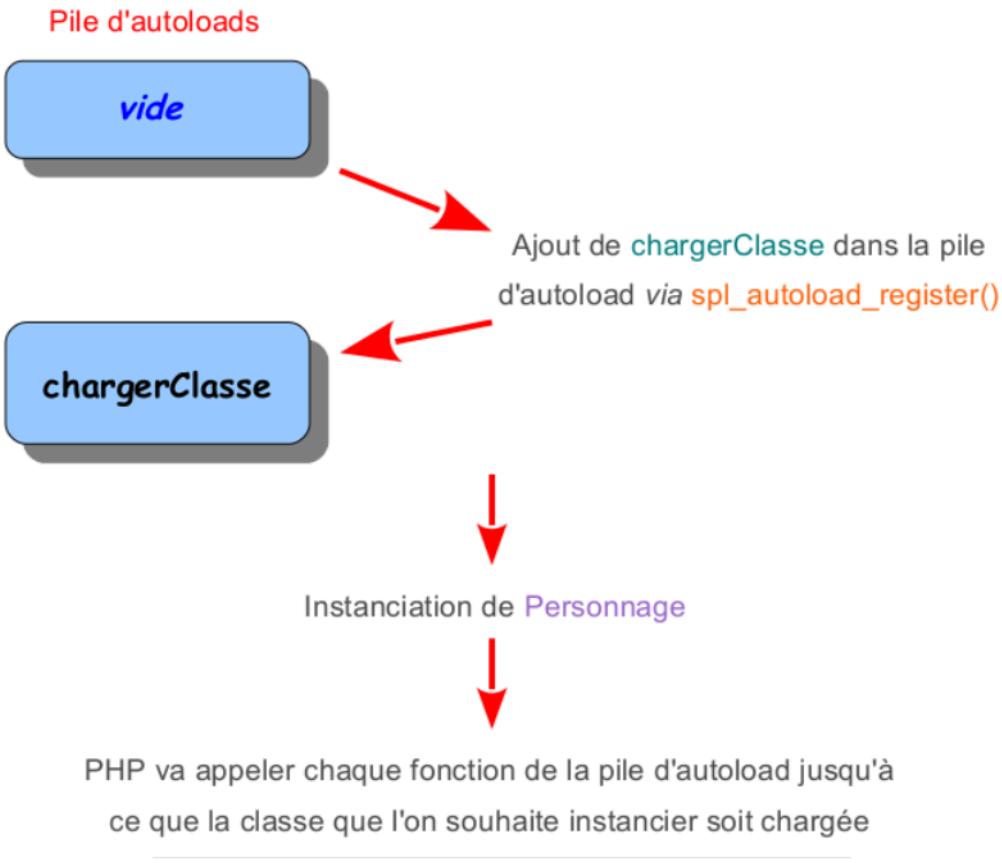
spl_autoload_register('chargerClasse'); // On enregistre la fonction en autoload pour qu'elle soit appelée dès qu'on instanciera une classe non déclarée.

$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, la figure suivante montre ce qui s'est passé.



L'autoload en PHP

Sachez que vous pouvez enregistrer autant de fonctions en autoload que vous le voulez avec `spl_autoload_register`. Si vous en 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.

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 !

En résumé

- Un objet se crée grâce à l'opérateur `new`.
- L'accès à un attribut ou à une méthode d'un objet se fait grâce à l'opérateur « `->` ».
- Pour lire ou modifier un attribut, on utilise des *accesseurs* et des *mutateurs*.
- Le constructeur d'une classe a pour rôle principal d'initialiser l'objet en cours de création, c'est-à-dire d'initialiser la valeur des attributs (soit en assignant directement des valeurs spécifiques, soit en appelant diverses méthodes).
- Les classes peuvent être chargées dynamiquement (c'est-à-dire sans avoir explicitement inclus le fichier la déclarant) grâce à l'auto-chargement de classe (utilisation de `spl_autoload_register`).

☐ J'ai terminé ce chapitre et je passe au suivant

L'OPÉRATEUR DE RÉSOLUTION DE PORTÉE

L'opérateur de résolution de portée (« `::` »), appelé « double deux points » (« *Scope Resolution Operator* » en anglais), 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 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**, sortes d'attributs dont la valeur est constante, c'est-à-dire qu'elle ne change pas. Nous allons d'ailleurs commencer par ces constantes de classe.

Cet opérateur est aussi appelé « Paamayim Nekudotayim ». Mais rassurez-vous, 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 constante à l'aide de la fonction `define`. Les constantes de classe permettent d'éviter tout code *muet*. Voici un code muet :

```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 réaliser 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 nous allons passer à la pratique. Pour déclarer une constante, vous devez faire précéder son nom du mot-clé `const`. Faites bien attention, une constante ne prend pas de « $ » devant son nom ! Voici donc comment créer une constante :

```php
<?php
class Personnage
{
  // Je rappelle : tous les attributs en privé !

  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()
  {

  }
}
```

S'inscrire au cours
Comment ça marche ?

[Théorie] Les bases de la POO

1. Introduction à la POO
2. Utiliser la classe
3. **L'opérateur de résolution de portée**
4. Manipulation de données stockées
5. TP : Mini-jeu de combat
6. L'héritage
7. TP : Des personnages spécialisés
8. Les méthodes magiques

Quiz

Accéder au forum

Créé par

OPENCLASSROOMS

OpenClassrooms, 1ère plateforme e-Education en Europe

Et voilà ! Facile n'est-ce pas ?

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

Et quel est le rapport avec tes « double deux points » ?

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 :

```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)
  {
    // N'oubliez pas qu'il faut assigner la valeur d'un attribut uniquement depuis son setter !
    $this->setForce($forceInitiale);
  }

  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
  {

  }

  public function setForce($force)
  {
    // On vérifie qu'on nous donne bien soit une « FORCE_PETITE », soit une « FORCE_MOYENNE », soit une « FORCE_GRANDE ».
    if (in_array($force, [self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE]))
    {
      $this->_force = $force;
    }
  }
}
```

Et lors de la création de notre personnage :

```php
<?php
// On envoie une « FORCE_MOYENNE » en guise de force initiale.
$perso = new Personnage(Personnage::FORCE_MOYENNE);
```

Notez qu'ici les noms de constantes sont en majuscules : c'est encore et toujours une convention de dénomination.

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

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 une classe et non sur un objet. Par conséquent, je ne veux voir aucun `$this` dans la méthode ! En effet, la méthode n'étant appelée sur aucun objet, il serait illogique que cette variable existe. Souvenez-vous : `$this` est un paramètre implicite. Cela n'est vrai que pour les méthodes appelées sur un objet !

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

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

```php
<?php
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;

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

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

  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
  {

  }

  public function setForce($force)
  {
    // On vérifie qu'on nous donne bien soit une « FORCE_PETITE », soit une « FORCE_MOYENNE », soit une « FORCE_GRANDE ».
    if (in_array($force, [self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE]))
    {
      $this->_force = $force;
    }
  }

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

Et dans le code, vous pourrez faire :

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

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

```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 évitera 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 :

```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)
  {
    $this->setForce($forceInitiale);
  }

  public function deplacer()
  {

  }

  public function frapper()
  {

  }

  public function gagnerExperience()
  {

  }

  public function setForce($force)
  {
    // On vérifie qu'on nous donne bien soit un « FORCE_PETITE », soit une « FORCE_MOYENNE », soit une « FORCE_GRANDE ».
    if (in_array($force, [self::FORCE_PETITE, self::FORCE_MOYENNE, self::FORCE_GRANDE]))
    {
      $this->_force = $force;
    }
  }

  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. D'accord, mais qu'est-ce qu'on peut faire sur une classe ? Et bien tout simplement *modifier les attributs de celle-ci* car, 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 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.

La ligne 47 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 ce n'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.

Nous allons maintenant faire un petit exercice. Je veux que vous me fassiez une classe toute bête qui ne sert à rien. Seulement, à la fin du script, je veux pouvoir afficher le nombre de fois où 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 :

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

  public function __construct()
  {
    // On instancie la variable $compteur qui appartient à la classe (donc utilisation du mot-clé self).
    self::$_compteur++;
  }

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

$test1 = new Compteur;
$test2 = new Compteur;
$test3 = new Compteur;

echo Compteur::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 uniquement dans l'objet créé mais dans la structure complète de la classe ! Je me répète, mais il est essentiel que vous compreniez ça, c'est très important.

En résumé

- L'opérateur « `->` » permet d'accéder à un élément de tel objet, tandis que l'opérateur « `::` » permet d'accéder à un élément de telle classe.
- Au sein d'une méthode, on accède à l'objet grâce à la pseudo-variable `$this`, tandis qu'on accède à la classe grâce au mot-clé `self`.
- Les attributs et méthodes statiques ainsi que les constantes de classe sont des éléments propres à la classe, c'est-à-dire qu'il n'est pas utile de créer un objet pour s'en servir.
- Les constantes de classe sont utiles pour éviter d'avoir un code muet, c'est-à-dire un code qui, sans commentaire, ne nous informe pas vraiment sur son fonctionnement.
- Les attributs et méthodes statiques sont utiles lorsque l'on ne veut pas avoir besoin d'un objet pour s'en servir.

☐ J'ai terminé ce chapitre et je passe au suivant

MANIPULATION DE DONNÉES STOCKÉES

Tout site web dynamique doit être capable de sauvegarder des données afin de les utiliser ultérieurement. Comme vous le savez sans doute, l'un des moyens les plus simples et efficaces est la base de données permettant de sauvegarder des centaines de données et de les récupérer en un temps très court.

Cependant, la gestion des données stockées en BDD peut être assez difficile à cerner en POO. Le débutant ne sait en général pas par où commencer et se pose tout un tas de questions : où créer la connexion avec la BDD ? Où placer les requêtes ? Comment traiter les valeurs retournées ? Nous allons justement voir tout cela ensemble. Accrochez-vous bien, un TP suivra pour voir si vous avez bien tout compris !

Une entité, un objet

Rappels sur la structure d'une BDD

Commençons ce chapitre par observer la structure d'une table en BDD et, plus précisément, la manière dont sont stockées les données. Comme vous le savez sûrement, une table n'est autre qu'un grand tableau où sont organisées les données :

⇐T⇒	id	nom	forcePerso	degats	niveau	experience
☐ ✎ ✖	16	Vyk12	5	55	4	20
☐ ✎ ✖	18	Tchaper	5	5	1	0
☐ ✎ ✖	19	Neo	5	0	1	20

Exemple d'une table de personnages

Nous voyons ici que notre table contient 3 personnages. Ces données sont stockées sous forme d'entrées. C'est sous cette forme que le gestionnaire de la BDD (MySQL, PostgreSQL, etc.) manipule les données. Si l'on regarde du côté de l'application qui les manipule (ici, notre script PHP), on se rend compte que c'est sous une forme similaire que les données sont récupérées. Cette forme utilisée, vous la connaissez bien : vous utilisiez jusqu'à présent des tableaux pour les manipuler. Exemple :

```html+php
<?php
// On admet que $db est un objet PDO
$request = $db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages');

while ($perso = $request->fetch(PDO::FETCH_ASSOC)) // Chaque entrée sera récupérée et placée dans un array.
{
  echo $perso['nom'], ' a ', $perso['forcePerso'], ' de force, ', $perso['degats'], ' de dégâts, ', $perso['experience'], ' d\'expérience et est au niveau ', $perso['niveau'];
}
```

Ça, si vous avez déjà fait un site internet de A à Z, vous avez du l'écrire pas mal de fois !

> Pour les exemples, et dans la suite du cours, j'utilise l'API PDO pour accéder à la base de données. Je vous conseille de lire le cours de M@teo21 sur PDO pour être à l'aise avec !

Maintenant, puisque nous programmons de manière orientée objet, nous voulons travailler seulement avec des objets (c'est le principe même de la POO, rappelons-le) et non plus avec des tableaux. En d'autres termes, il va falloir transformer le tableau que l'on reçoit en objet.

Travailler avec des objets

Avant de travailler avec des objets, encore faut-il savoir de quelle classe ils sont issus. Dans notre cas, nous voulons des objets représentant des personnages. On aura donc besoin d'une classe `Personnage` !

```php
<?php
class Personnage
{

}
```

Vient maintenant une question embêtante dans votre tête pour construire cette classe : « Je commence par où ? »

Une classe est composée de deux parties (éventuellement trois) :

- une partie déclarant les attributs. Ce sont les *caractéristiques* de l'objet ;

- une partie déclarant les méthodes. Ce sont les *fonctionnalités* de chaque objet ;
- éventuellement, une partie déclarant les constantes de classe. Nous nous en occuperons en temps voulu.

Lorsque l'on veut construire une classe, il va donc falloir *systématiquement* se poser les mêmes questions :

- Quelles seront les caractéristiques de mes objets ?
- Quelles seront les fonctionnalités de mes objets ?

Les réponses à ces questions aboutiront à la réalisation du plan de la classe, le plus difficile. 

Commençons donc à réfléchir sur le plan de notre classe en répondant à la première question : « Quelles seront les caractéristiques de mes objets ? » Pour vous mettre sur la piste, regardez de nouveau le tableau de la BDD contenant les entrées, cela vous donnera peut-être la réponse… Et oui, les attributs de notre classe (les caractéristiques) nous sont offerts sur un plateau ! Ils sont listés en haut du tableau : `id`, `nom`, `forcePerso`, `degats`, `niveau` et `experience`.

Écrivons maintenant notre classe :

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;
}
```

Il faut bien sûr implémenter (écrire) les *getters* et *setters* qui nous permettront d'accéder et de modifier les valeurs de notre objet. Pour rappel, un *getter* est une méthode chargée de renvoyer la valeur d'un attribut, tandis qu'un *setter* est une méthode chargée d'assigner une valeur à un attribut **en vérifiant son intégrité** (si vous assignez la valeur sans aucun contrôle, vous perdez tout l'intérêt qu'apporte le principe d'encapsulation).

Pour construire nos setters, il faut donc nous pencher sur les valeurs possibles de chaque attribut :

- les valeurs possibles de l'identifiant sont tous les nombres entiers strictement positifs ;
- les valeurs possibles pour le nom du personnage sont toutes les chaînes de caractères ;
- les valeurs possibles pour la force du personnage sont tous les nombres entiers allant de 1 à 100 ;
- les valeurs possibles pour les dégâts du personnage sont tous les nombres entiers allant de 0 à 100 ;
- les valeurs possibles pour le niveau du personnage sont tous les nombres entiers allant de 1 à 100 ;
- les valeurs possibles pour l'expérience du personnage sont tous les nombres entiers allant de 1 à 100.

On en déduit donc le code de chaque setter. Voici donc ce que donnerait notre classe avec ses getters et setters :

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;

  // Liste des getters

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

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

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

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

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

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

  // Liste des setters

  public function setId($id)
  {
    // On convertit l'argument en nombre entier.
    // Si c'en était déjà un, rien ne changera.
    // Sinon, la conversion donnera le nombre 0 (à quelques exceptions près, mais rien d'important ici).
    $id = (int) $id;

    // On vérifie ensuite si ce nombre est bien strictement positif.
    if ($id > 0)
    {
      // Si c'est le cas, c'est tout bon, on assigne la valeur à l'attribut correspondant.
      $this->_id = $id;
    }
  }

  public function setNom($nom)
  {
    // On vérifie qu'il s'agit bien d'une chaîne de caractères.
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }

  public function setForcePerso($forcePerso)
  {
    $forcePerso = (int) $forcePerso;

    if ($forcePerso >= 1 && $forcePerso <= 100)
    {
      $this->_forcePerso = $forcePerso;
    }
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setNiveau($niveau)
  {
    $niveau = (int) $niveau;

    if ($niveau >= 1 && $niveau <= 100)
    {
      $this->_niveau = $niveau;
    }
  }

  public function setExperience($experience)
  {
    $experience = (int) $experience;

    if ($experience >= 1 && $experience <= 100)
    {
      $this->_experience = $experience;
    }
  }
}
?>
```

Reprenons notre code de tout à l'heure, celui qui nous permettait de lire la BDD. Modifions-le un peu pour qu'on puisse manipuler des objets et non des tableaux :

```html+php
<?php
// On admet que $db est un objet PDO.
$request = $db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages');

while ($donnees = $request->fetch(PDO::FETCH_ASSOC)) // Chaque entrée sera récupérée et placée dans un array.
{
  // On passe les données (stockées dans un tableau) concernant le personnage au constructeur de la classe.
  // On admet que le constructeur de la classe appelle chaque setter pour assigner les valeurs qu'on lui a données aux attributs correspondants.
  $perso = new Personnage($donnees);

  echo $perso->nom(), ' a ', $perso->forcePerso(), ' de force, ', $perso->degats(), ' de dégâts, ', $perso->experience(), ' d\'expérience et est au niveau ', $perso->niveau();
}
```

Et quelles seront les méthodes de nos classes ?

Les méthodes concernent des fonctionnalités que possède l'objet, des actions qu'il peut effectuer. Voyez-vous des fonctionnalités intéressantes à implémenter ? Pour les opérations basiques que l'on effectue, il n'y en a pas besoin. En effet, nous voulons juste créer des objets et assigner des valeurs aux attributs, donc hormis les getters et setters, aucune autre méthode n'est nécessaire !

D'accord, nous avons des objets maintenant. Mais à quoi cela sert-il, concrètement ?

Il est sûr que dans cet exemple précis, cela ne sert à rien. Mais vous verrez plus tard qu'il est beaucoup plus intuitif de travailler avec des objets et par conséquent beaucoup plus pratique, notamment sur de grosses applications où de nombreux objets circulent un peu dans tous les sens.

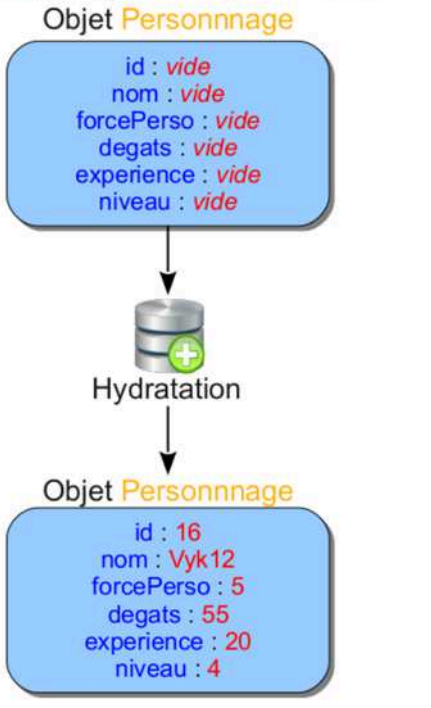
L'hydratation

La théorie de l'hydratation

L'hydratation est un point essentiel dans le domaine de la POO, notamment lorsqu'on utilise des objets représentant des données stockées. Cette notion peut vite devenir compliquée et créer des interrogations pour un développeur débutant si elle est abordée de manière approximative, alors que des explications claires prouvent qu'il n'y a rien de compliqué là-dedans.

Quand on vous parle d'hydratation, c'est qu'on parle d'« objet à hydrater ». Hydrater un objet, c'est tout simplement lui apporter ce dont il a besoin pour fonctionner. En d'autres termes plus précis, hydrater un objet revient à lui fournir des données correspondant à ses attributs pour qu'il assigne les valeurs souhaitées à ces derniers. L'objet aura ainsi des attributs valides et sera en lui-même valide. On dit que l'objet a ainsi été hydraté.

Schématiquement, une hydratation se produit comme ceci :



Hydratation d'un objet

Au début, nous avons un objet `Personnage` dont les attributs sont vides. Comme le schéma le représente, l'hydratation consiste à assigner des valeurs aux attributs. Ainsi l'objet est fonctionnel car il contient des attributs valides : nous avons donc bien hydraté l'objet.

L'hydratation en pratique

Comme on l'a vu, hydrater un objet revient à assigner des valeurs à ses attributs. Qu'avons-nous besoin de faire pour réaliser une telle chose ? Il faut ajouter à l'objet l'action de s'hydrater. Et qui dit action dit méthode !

Nous allons donc écrire notre fonction `hydrate()` :

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;

  // Un tableau de données doit être passé à la fonction (d'où le préfixe « array »).
  public function hydrate(array $donnees)
  {

  }

  public function id() { return $this->_id; }
  public function nom() { return $this->_nom; }
  public function forcePerso() { return $this->_forcePerso; }
  public function degats() { return $this->_degats; }
  public function niveau() { return $this->_niveau; }
  public function experience() { return $this->_experience; }

  public function setId($id)
  {
    // L'identifiant du personnage sera, quoi qu'il arrive, un nombre entier.
    $this->_id = (int) $id;
  }

  public function setNom($nom)
  {
    // On vérifie qu'il s'agit bien d'une chaîne de caractères.
    // Dont la longueur est inférieure à 30 caractères.
    if (is_string($nom) && strlen($nom) <= 30)
    {
      $this->_nom = $nom;
    }
  }

  public function setForcePerso($forcePerso)
  {
    $forcePerso = (int) $forcePerso;

    // On vérifie que la force passée est comprise entre 0 et 100.
    if ($forcePerso >= 0 && $forcePerso <= 100)
    {
      $this->_forcePerso = $forcePerso;
    }
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    // On vérifie que les dégâts passés sont compris entre 0 et 100.
    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setNiveau($niveau)
  {
    $niveau = (int) $niveau;

    // On vérifie que le niveau n'est pas négatif.
    if ($niveau >= 0)
    {
      $this->_niveau = $niveau;
    }
  }

  public function setExperience($exp)
  {
    $exp = (int) $exp;

    // On vérifie que l'expérience est comprise entre 0 et 100.
    if ($exp >= 0 && $exp <= 100)
    {
      $this->_experience = $exp;
    }
  }
}
```

Vient maintenant l'aspect le plus important : que doit-on mettre dans cette méthode ? Souvenez-vous de sa fonction : elle doit hydrater l'objet, c'est-à-dire « assigner les valeurs passées en paramètres aux attributs correspondant ». Cependant, je vous le dis avant que vous fassiez fausse route : il ne faut pas assigner ces valeurs directement, comme ceci :

```php
<?php
// …
public function hydrate(array $donnees)
{
  if (isset($donnees['id']))
  {
    $this->_id = $donnees['id'];
  }

  if (isset($donnees['nom']))
  {
    $this->_nom = $donnees['nom'];
  }

  // …
}
// …
```

La principale raison pour laquelle c'est une mauvaise façon de procéder, est qu'en agissant ainsi vous violez le principe d'encapsulation. En effet, de cette manière, vous ne contrôlez pas l'intégrité des valeurs. Qui vous garantit que le tableau contiendra un nom valide ? Rien du tout! Comment contrôler alors l'intégrité de ces valeurs ? Regardez un peu votre classe, la réponse est sous vos yeux… C'est le rôle des setters de faire ces vérifications ! Il faudra donc les appeler au lieu d'assigner les valeurs directement :

```php
<?php
// …
public function hydrate(array $donnees)
{
  if (isset($donnees['id']))
  {
    $this->setId($donnees['id']);
  }

  if (isset($donnees['nom']))
  {
    $this->setNom($donnees['nom']);
  }

  // …
}
// …
```

Théoriquement, nous avons terminé le travail. Cependant, cela n'est pas très flexible : si vous ajoutez un attribut (et donc son setter correspondant), il faudra modifier votre méthode `hydrate()` pour ajouter la possibilité d'assigner cette valeur. De plus, avec les 6 attributs actuellement créés, il est long de vérifier si la clé existe puis d'appeler la méthode correspondante.

Nous allons donc procéder plus rapidement : nous allons créer une boucle qui va parcourir le tableau passé en paramètre. Si le setter correspondant à la clé existe, alors on appelle ce setter en lui passant la valeur en paramètre. En langage naturel, l'algorithme peut s'écrire de cette façon :

```
PARCOURS du tableau $donnees (avec pour clé $cle et pour valeur $valeur)
  On assigne à $setter la valeur « 'set'.$cle », en mettant la première lettre de $cle en majuscu
  SI la méthode $setter de notre classe existe ALORS
    On invoque $setter($valeur)
  FIN SI
FIN PARCOURS
```

Pas de panique, je vais vous expliquer chaque ligne. Dans votre tête, prenez un exemple de valeur pour `$donnees` :

```php
<?php
$donnees = [
  'id' => 16,
  'nom' => 'Vyk12',
  'forcePerso' => 5,
  'degats' => 55,
  'niveau' => 4,
  'experience' => 20
];
```

Vous voyez bien que chaque clé correspond à un attribut de notre objet, à qui on assigne une valeur précise. Dans notre méthode `hydrate()`, lorsqu'on parcourt notre tableau, on a successivement la clé `id`, puis `nom`, puis `forcePerso`, etc., avec leur valeur correspondante.

En récupérant le nom de l'attribut, il est facile de déterminer le setter correspondant. En effet, chaque setter a pour nom `setNomDeLAttribut`.

Il est important de préciser que la première lettre du nom de l'attribut doit être en majuscule. Par exemple, le setter correspondant à `nom` est `setNom`.

Pour l'instant notre méthode ressemble à ceci :

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    $method = 'set'.ucfirst($key);
    // …
  }
}
// …
```

Il faut maintenant vérifier que la méthode existe. Pour cela, nous allons utiliser la fonction method_exists(). Elle prend en premier paramètre le nom de la classe ou une instance de cette classe, et en deuxième paramètre le nom de la méthode qui nous intéresse. La méthode renvoie `true` si la méthode existe, sinon `false`.

Je vous laisse ajouter cette condition qui permet de voir si le setter correspondant existe.

Voici la correction :

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    $method = 'set'.ucfirst($key);

    if (method_exists($this, $method))
    {
      // …
    }
  }
}
// …
```

Et maintenant, il ne reste plus qu'à appeler le setter à l'intérieur de la condition ! Pour cela, je vais vous apprendre une petite astuce. Il est possible d'appeler une méthode dynamiquement, c'est-à-dire appeler une méthode dont le nom n'est pas connu à l'avance (en d'autres termes, le nom est connu seulement pendant l'exécution et est donc stocké dans une variable). Pour ce faire, rien de plus simple ! Regardez ce code :

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

$a = new A;
$method = 'hello';

$a->$method(); // Affiche : « Hello world ! »
```

Il est bien entendu possible de passer des arguments à la méthode.

Vous êtes maintenant capables de créer la dernière instruction dans notre méthode ! Pour rappel, celle-ci doit invoquer le setter dont le nom est contenu dans la variable `$method`.

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    $method = 'set'.ucfirst($key);

    if (method_exists($this, $method))
    {
      // …
    }
  }
}
// …
```

Et maintenant, il ne reste plus qu'à appeler le setter à l'intérieur de la condition ! Pour cela, je vais vous apprendre une petite astuce. Il est possible d'appeler une méthode dynamiquement, c'est-à-dire appeler une méthode dont le nom n'est pas connu à l'avance (en d'autres termes, le nom est connu seulement pendant l'exécution et est donc stocké dans une variable). Pour ce faire, rien de plus simple ! Regardez ce code :

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

$a = new A;
$method = 'hello';

$a->$method(); // Affiche : « Hello world ! »
```



Il est bien entendu possible de passer des arguments à la méthode.

Vous êtes maintenant capables de créer la dernière instruction dans notre méthode ! Pour rappel, celle-ci doit invoquer le setter dont le nom est contenu dans la variable `$method`.

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    // On récupère le nom du setter correspondant à l'attribut.
    $method = 'set'.ucfirst($key);

    // Si le setter correspondant existe.
    if (method_exists($this, $method))
    {
      // On appelle le setter.
      $this->$method($value);
    }
  }
}
// …
```

Cette fonction est très importante, vous la retrouverez dans de nombreux codes (parfois sous des formes différentes) provenant de plusieurs développeurs. Gardez-là donc dans un coin de votre tête.



Il est courant d'implémenter un constructeur à ces classes demandant un tableau de valeurs pour qu'il appelle ensuite la fonction d'hydratation afin que l'objet soit hydraté dès sa création, comme on l'a vu au début de ce chapitre.

Gérer sa BDD correctement

On vient de voir jusqu'à présent comment gérer les données que les requêtes nous renvoient, mais où placer ces requêtes ? Notre but est de programmer orienté objet, donc nous voulons le moins de code possible en-dehors des classes pour mieux l'organiser. Beaucoup de débutants sont tentés de placer les requêtes dans des méthodes de la classe représentant une entité de la BDD. Par exemple, dans le cas du personnage, je parie que la plupart d'entre vous seraient tentés d'écrire les méthodes `add()` et `update()` dans la classe `Personnage`, ayant respectivement pour rôle d'exécuter une requête pour ajouter et modifier un personnage en BDD.

Arrêtez-vous là tout de suite ! Je vais vous expliquer pourquoi vous faites fausse route, puis vous montrerai la bonne marche à suivre.

Une classe, un rôle

En POO, il y a une phrase très importante qu'il faut que vous ayez constamment en tête : « Une classe, un rôle. » Maintenant, répondez clairement à cette question : « Quel est le rôle d'une classe comme `Personnage` » ?

Un objet instanciant une classe comme `Personnage` a pour rôle de *représenter* une ligne présente en BDD. Le verbe « représenter » est ici très important. En effet, « représenter » est très différent de « gérer ». Une ligne de la BDD ne peut pas s'auto-gérer ! C'est comme si vous demandiez à un ouvrier ayant construit un produit de le commercialiser : l'ouvrier est tout à fait capable de le construire, c'est son *rôle*, mais il ne s'occupe pas du tout de sa gestion, il en est incapable. Il faut donc qu'une deuxième personne intervienne, un commercial, qui va s'occuper de vendre ce produit.

Pour revenir à nos objets d'origine, nous aurons donc besoin de quelque chose qui va s'occuper de les gérer. Ce quelque chose, vous l'aurez peut-être deviné, n'est autre qu'un objet. Un objet gérant des entités issues d'une BDD est généralement appelé un « manager ».

Comme un manager ne fonctionne pas sans support de stockage (dans notre cas, une BDD), on va prendre un exemple concret en créant un gestionnaire pour nos personnages, qui va donc se charger d'en ajouter, d'en modifier, d'en supprimer et d'en récupérer. Puisque notre classe est un gestionnaire de personnages, je vous propose de la nommer `PersonnagesManager`. Cependant, rappelez-vous les questions que l'on doit se poser pour établir le plan de notre classe :

- Quelles seront les caractéristiques de mes objets ?
- Quelles seront les fonctionnalités de mes objets ?

Les caractéristiques d'un manager

Partie délicate, car cela est moins intuitif que de trouver les caractéristiques d'un personnage. Pour trouver la réponse, vous devrez passer par une autre question (ce serait trop simple de toujours répondre à la même question !) : « De quoi a besoin un manager pour fonctionner ? »

Même si la réponse ne vous vient pas spontanément à l'esprit, vous la connaissez : elle a besoin d'une connexion à la BDD pour pouvoir exécuter des requêtes. En utilisant PDO, vous devriez savoir que la connexion à la BDD est représentée par un objet, un *objet d'accès à la BDD* (ou DAO pour *Database Access Object*). Vous l'avez peut-être compris : notre manager aura besoin de cet objet pour fonctionner, donc notre classe aura un attribut stockant cet objet.

Notre classe a-t-elle besoin d'autre chose pour fonctionner ?

Non, c'est tout. Vous pouvez donc commencer à écrire votre classe :

```php
<?php
class PersonnagesManager
{
  private $_db;
}
```

Les fonctionnalités d'un manager

Pour pouvoir gérer au mieux des entités présentes en BDD (ici, nos personnages), il va falloir quelques fonctionnalités de base. Quelles sont-elles ?

Un manager doit pouvoir :

- enregistrer une nouvelle entité ;
- modifier une entité ;
- supprimer une entité ;
- sélectionner une entité.

C'est le fameux CRUD (*Create, Read, Update, Delete*). N'oublions pas aussi d'ajouter un setter pour notre manager afin de pouvoir modifier l'attribut `$_db` (pas besoin d'accesseur). La création d'un constructeur sera aussi indispensable si nous voulons assigner à cet attribut un objet PDO dès l'instanciation du manager.

Nous allons donc écrire notre premier manager en écrivant les méthodes correspondant à ces fonctionnalités. Écrivez dans un premier temps les méthodes en ne les remplissant que de commentaires décrivant les instructions qui y seront écrites. Cela vous permettra d'organiser clairement le code dans votre tête.

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO.

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

  public function add(Personnage $perso)
  {
    // Préparation de la requête d'insertion.
    // Assignation des valeurs pour le nom, la force, les dégâts, l'expérience et le niveau du personnage.
    // Exécution de la requête.
  }

  public function delete(Personnage $perso)
  {
    // Exécute une requête de type DELETE.
  }

  public function get($id)
  {
    // Exécute une requête de type SELECT avec une clause WHERE, et retourne un objet Personnage.
  }

  public function getList()
  {
    // Retourne la liste de tous les personnages.
  }

  public function update(Personnage $perso)
  {
    // Prépare une requête de type UPDATE.
    // Assignation des valeurs à la requête.
    // Exécution de la requête.
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Une fois cette étape accomplie, vous pouvez remplacer les commentaires par les instructions correspondantes. C'est la partie la plus facile car vous l'avez déjà fait de nombreuses fois en procédural : ce ne sont que des requêtes à taper et des valeurs à retourner.

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

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

  public function add(Personnage $perso)
  {
    $q = $this->_db->prepare('INSERT INTO personnages SET nom = :nom, forcePerso = :forcePerso, degats = :degats, niveau = :niveau, experience = :experience');

    $q->bindValue(':nom', $perso->nom());
    $q->bindValue(':forcePerso', $perso->forcePerso(), PDO::PARAM_INT);
    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':niveau', $perso->niveau(), PDO::PARAM_INT);
    $q->bindValue(':experience', $perso->experience(), PDO::PARAM_INT);

    $q->execute();
  }

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

  public function get($id)
  {
    $id = (int) $id;

    $q = $this->_db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages WHERE id = '.$id);
    $donnees = $q->fetch(PDO::FETCH_ASSOC);

    return new Personnage($donnees);
  }

  public function getList()
  {
    $persos = [];

    $q = $this->_db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages ORDER BY nom');

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      $persos[] = new Personnage($donnees);
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->_db->prepare('UPDATE personnages SET forcePerso = :forcePerso, degats = :degats, niveau = :niveau, experience = :experience WHERE id = :id');

    $q->bindValue(':forcePerso', $perso->forcePerso(), PDO::PARAM_INT);
    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':niveau', $perso->niveau(), PDO::PARAM_INT);
    $q->bindValue(':experience', $perso->experience(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Essayons tout ça !

Faisons un petit test pour s'assurer que tout cela fonctionne.



Assurez-vous de bien avoir créé une table `personnages` dans votre BDD.

Commençons par créer un objet `Personnage` :

```php
<?php
$perso = new Personnage([
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
]);
```

Maintenant, comment l'ajouter en BDD ? Il faudra d'abord créer une instance de notre manager, *en lui passant une instance de PDO.*

```php
<?php
$perso = new Personnage([
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
]);

$db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');
$manager = new PersonnagesManager($db);
```

Nous n'avons plus qu'une instruction à entrer : celle ordonnant l'ajout du personnage en BDD. Et c'est tout !

```php
<?php
$perso = new Personnage([
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
]);

$db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');
$manager = new PersonnagesManager($db);

$manager->add($perso);
```

Il est maintenant temps de mettre en pratique ce que vous venez d'apprendre. Cependant, avant de continuer, assurez-vous bien de tout avoir compris, sinon vous serez perdus car vous ne comprendrez pas le code du TP !

En résumé

- Du côté du script PHP, chaque enregistrement de la base de données est représenté par un objet possédant une liste d'attributs identique à la liste des colonnes de la table.
- Il doit être possible d'*hydrater* chacun des objets grâce à une méthode `hydrate` qui a pour rôle d'assigner les valeurs passées en paramètre aux attributs correspondant.
- La communication avec la BDD se fait par le biais d'un objet différent de l'objet représentant l'enregistrement (une classe = un rôle). Un tel objet est un *manager*.
- Un *manager* peut stocker les objets en BDD, mais peut tout-à-fait les stocker sur un autre support (fichier XML, fichier texte, etc.).

☐ J'ai terminé ce chapitre et je passe au suivant

TP : MINI-JEU DE COMBAT

Voici un petit TP qui mettra en pratique ce que l'on vient de voir. Celui-ci est étroitement lié avec le précédent chapitre. Ainsi, je vous conseille d'avoir bien compris ce dernier avant d'aborder ce TP, sinon vous n'arriverez sans doute pas au résultat tout seul.

En tout, vous aurez 3 TP de la sorte dont le niveau de difficulté sera croissant. Puisque celui-ci est votre premier, tout sera expliqué dans les moindres détails.

Ce qu'on va faire

Cahier des charges

Ce que nous allons réaliser est très simple. Nous allons 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. Le personnage frappé se verra infliger un certain degré de dégâts.

Un personnage est défini selon 2 caractéristiques :

- Son nom (unique).
- Ses dégâts.

Les dégâts d'un personnage sont compris entre 0 et 100. Au début, il a bien entendu 0 de dégât. Chaque coup qui lui sera porté lui fera prendre 5 points de dégâts. Une fois arrivé à 100 points de dégâts, le personnage est mort (on le supprimera alors de la BDD).

Notions utilisées

Voici une petite liste vous indiquant les points techniques que l'on va mettre en pratique avec ce TP :

- Les attributs et méthodes ;
- l'*instanciation* de la classe ;
- les constantes de classe ;
- et surtout, tout ce qui touche à la **manipulation de données stockées.**

Cela dit, et c'est je pense ce qui sera le plus difficile, ce qui sera mis en valeur ici sera la **conception** d'un mini-projet, c'est-à-dire que l'on travaillera surtout ici votre capacité à programmer orienté objet. Cependant, ne prenez pas trop peur : nous avons effectué une importante partie théorique lors du précédent chapitre, et rien de nouveau n'apparaîtra.

Vous avez donc maintenant une idée de ce qui vous attend. Cependant, ceci étant votre premier TP concernant la POO, il est fort probable que vous ne sachiez pas par où commencer, et c'est normal ! Nous allons donc réaliser le TP ensemble : je vais vous expliquer comment **penser** un mini-projet et vous poser les bonnes questions.

Pré-conception

Avant de nous attaquer au cœur du script, nous allons réfléchir à son organisation. De quoi aura-t-on besoin ? Puisque nous travaillerons avec des personnages, nous aurons besoin de les stocker pour qu'ils puissent durer dans le temps. L'utilisation d'une base de données sera donc indispensable.

Le script étant simple, nous n'aurons qu'une table **personnages** qui aura différents champs. Pour les définir, réfléchissez à ce qui caractérise un personnage. Ainsi nous connaissons déjà 2 champs de cette table que nous avons définis au début : **nom** et **dégâts** . Et bien sûr, n'oublions pas le plus important : l'identifiant du personnage ! Chaque personnage doit posséder un identifiant unique qui permet ainsi de le rechercher plus rapidement (au niveau performances) qu'avec son nom.

Vous pouvez donc ainsi créer votre table tous seuls *via* PhpMyAdmin. Si vous n'êtes pas sûrs de vous, je vous laisse le code SQL créant cette table :

```mysql
CREATE TABLE IF NOT EXISTS `personnages` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `nom` varchar(50) COLLATE utf8_general_ci NOT NULL,
  `degats` tinyint(3) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  UNIQUE KEY `nom` (`nom`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_general_ci;
```

Voir le résultat que vous obtiendrez

Le résultat présenté ici contient les améliorations proposées en fin de chapitre.

Première étape : le personnage

Nous allons commencer le TP. Pour rappel, nous allons réaliser un petit jeu mettant en scène des personnages qui peuvent combattre. Qui dit personnages dit **objets** `Personnage` (je pense que ça, vous l'avez deviné, puisque nous avons travaillé dessus durant les premiers chapitres).

Arrive maintenant un moment délicat dans votre tête : **"Par où je commence ?"**

Souvenez-vous de la première partie du précédent chapitre. Pour construire une classe, vous devez répondre à deux questions qui vont vous permettre d'établir le plan de votre classe :

- Quelles seront les caractéristiques de mes objets ?

- Quelles seront les fonctionnalités de mes objets ?

Voici comment procéder. Nous allons dans un premier temps dresser la liste des caractéristiques du personnage pour ensuite se pencher sur ses fonctionnalités.



Rappel : le traitement des données (c'est-à-dire l'exécution de requêtes permettant d'aller effectuer des opérations en BDD) se fera **dans une autre classe**. Ne vous en préoccupez donc pas maintenant !

Les caractéristiques du personnage

Essayez de vous souvenir quelles sont les caractéristiques d'un personnage (ou plus globalement celles d'une entité représentant un enregistrement présent en BDD). Nous l'avons vu dans la première partie du précédent chapitre.



Les attributs d'un objet représentant un enregistrement présent en BDD correspondent aux noms des champs de cet enregistrement.

Si vous n'avez pas retenu cette information, notez-là quelque part, vous en aurez besoin un très grand nombre de fois.

Maintenant que nous avons déterminé les caractéristiques d'un objet `Personnage`, nous savons quels attributs placer dans notre classe. Voici une première écriture de notre classe `Personnage` :

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;
}
```

Jusque là tout va bien. Tournons-nous maintenant vers les fonctionnalités que doit posséder un personnage.

Les fonctionnalités d'un personnage

Comme nous l'avons dit tout-à-l'heure, pour obtenir les méthodes d'un objet, il faut se demander quelles seront les fonctionnalités de ces entités. Ici, **que pourra faire un personnage** ? Relisez les consignes du début de chapitre et répondez clairement à la question.

Un personnage doit pouvoir :

- **Frapper** un autre personnage ;
- **recevoir** des dégâts.

À chaque fonctionnalité correspond une méthode. Écrivez ces méthodes dans la classe en mettant en commentaire ce qu'elles doivent faire.



Ne vous étonnez pas si vous avez un résultat bien différent du mien. Le contenu des méthodes importe peu car il est issu de mon imagination et il est normal que vous n'ayez pas pensé le script comme moi. Gardez votre version, n'essayez pas de copier/coller mon code.

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
    // Si c'est le cas, on stoppe tout en renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage a bien été frappé.
  }
}
```

Tout ceci n'est que du français, nous sommes encore à l'étape de **réflexion**. Si vous sentez que ça va trop vite, relisez le début du TP et suivez bien les étapes. C'est très important car c'est en général cette étape qui pose problème : la **réflexion** ! Beaucoup de débutants foncent tête baissée sans prendre le temps de concevoir correctement les méthodes au début et se plantent royalement. Prenez donc bien votre temps.

Normalement, des petites choses doivent vous chiffonner.

Pour commencer, la première chose qui doit vous interpeller se situe dans la méthode `frapper()`, lorsqu'il faut vérifier qu'on ne se frappe pas soi-même. Pour cela, il suffit de comparer l'identifiant du personnage qui attaque avec l'identifiant du personnage ciblé. En effet, si l'identifiant est le même, alors il s'agira d'un seul et même personnage.

Ensuite, à certains moments dans le code, il est dit que la méthode « renverra une valeur signifiant que le personnage a bien été frappé » par exemple. Qu'est-ce que cela peut bien signifier ? Ce genre de commentaire laissera place à des **constantes de classe** (si vous l'aviez deviné, bien joué !). Si vous regardez bien le code, vous verrez 3 commentaires de la sorte :

- Méthode `frapper()` : *« [...] renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque »* ⟶ la méthode renverra la valeur de la constante `CEST_MOI` ;
- Méthode `recevoirDegats()` : *« la méthode renverra une valeur signifiant que le personnage a été tué »* ⟶ la méthode renverra la valeur de la constante `PERSONNAGE_TUE` ;
- Méthode `recevoirDegats()` : *« elle renverra une valeur signifiant que le personnage a bien été frappé »* ⟶ la méthode renverra la valeur de la constante `PERSONNAGE_FRAPPE`.

Vous pouvez ajouter ces constantes à votre classe.

html+php
```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;

  const CEST_MOI = 1;
  const PERSONNAGE_TUE = 2;
  const PERSONNAGE_FRAPPE = 3;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
    // Si c'est le cas, on stoppe tout en renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage a bien été frappé.
  }
}
```

Les getters et setters

Actuellement, les attributs de nos objets sont inaccessibles. Il faut créer des *getters* pour pouvoir les lire, et des *setters* pour pouvoir modifier leurs valeurs. Nous allons aller un peu plus rapidement que les précédentes méthodes en écrivant directement le contenu de celles-ci car une ou deux instructions suffisent en général.

N'oubliez pas de vérifier l'intégrité des données dans les *setters* sinon ils perdent tout leur intérêt !

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;

  const CEST_MOI = 1;
  const PERSONNAGE_TUE = 2;
  const PERSONNAGE_FRAPPE = 3;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
    // Si c'est le cas, on stoppe tout en renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage a bien été frappé.
  }

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

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

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

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setId($id)
  {
    $id = (int) $id;

    if ($id > 0)
    {
      $this->_id = $id;
    }
  }

  public function setNom($nom)
  {
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }
}
```

Hydrater ses objets

Deuxième point essentiel que nous allons ré-exploiter dans ce TP : l'hydratation.

Je n'ai pas d'explication supplémentaire à ajouter, tout est dit dans le précédent chapitre. La méthode créée dans ce dernier est d'ailleurs réutilisable ici. Au lieu de regarder la correction en vous replongeant dans le chapitre précédent, essayez plutôt de réécrire la méthode. Pour rappel, celle-ci doit permettre d'assigner aux attributs de l'objet les valeurs correspondantes, passées en paramètre dans un tableau. Si vous avez essayé de réécrire la méthode, voici le résultat que vous auriez du obtenir :

```php
<?php
class Personnage
{
  // ...

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  // ...
}
```

Il ne manque plus qu'à implémenter le constructeur pour qu'on puisse directement hydrater notre objet lors de l'instanciation de la classe. Pour cela, ajoutez un paramètre : `$donnees`. Appelez ensuite directement la méthode `hydrate()`.

```php
<?php
class Personnage
{
  // ...

  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
  }

  // ...
}
```

Codons le tout !

La partie réflexion est terminé, il est maintenant temps d'écrire notre classe `Personnage` ! Voici la correction que je vous propose :

```php
<?php
class Personnage
{
  private $_degats,
          $_id,
          $_nom;

  const CEST_MOI = 1; // Constante renvoyée par la méthode `frapper` si on se frappe soi-même.
  const PERSONNAGE_TUE = 2; // Constante renvoyée par la méthode `frapper` si on a tué le personnage en le frappant.
  const PERSONNAGE_FRAPPE = 3; // Constante renvoyée par la méthode `frapper` si on a bien frappé le personnage.


  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
  }

  public function frapper(Personnage $perso)
  {
    if ($perso->id() == $this->_id)
    {
      return self::CEST_MOI;
    }

    // On indique au personnage qu'il doit recevoir des dégâts.
    // Puis on retourne la valeur renvoyée par la méthode : self::PERSONNAGE_TUE ou self::PERSONNAGE_FRAPPE
    return $perso->recevoirDegats();
  }

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  public function recevoirDegats()
  {
    $this->_degats += 5;

    // Si on a 100 de dégâts ou plus, on dit que le personnage a été tué.
    if ($this->_degats >= 100)
    {
      return self::PERSONNAGE_TUE;
    }

    // Sinon, on se contente de dire que le personnage a bien été frappé.
    return self::PERSONNAGE_FRAPPE;
  }


```

```php
  // GETTERS //


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

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

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

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setId($id)
  {
    $id = (int) $id;

    if ($id > 0)
    {
      $this->_id = $id;
    }
  }

  public function setNom($nom)
  {
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }
}
```

Prenez le temps de bien comprendre ce code et de lire les commentaires. Il est important que vous cerniez son fonctionnement pour savoir ce que vous faites. Cependant, si vous ne comprenez pas toutes les instructions placées dans les méthodes, ne vous affolez pas : si vous avez compris globalement le rôle de chacune d'elles, vous n'aurez pas de handicap pour suivre la prochaine sous-partie.

Seconde étape : stockage en base de données

Attaquons-nous maintenant à la deuxième grosse partie de ce TP, celle consistant à pouvoir stocker nos personnages dans une base de données. Grande question maintenant : **comment faire ?**

Nous avons répondu à cette question dans la troisième partie du précédent chapitre. Au cas où certains seraient toujours tentés de placer les requêtes qui iront chercher les personnages en BDD dans la classe `Personnage`, je vous arrête tout de suite et vous fais un bref rappel avec cette phrase que vous avez déjà rencontrée : **une classe, un rôle.**

J'espère que vous l'avez retenue cette fois-ci !

Vous souvenez-vous de ce que cela signifie ? Quel est le rôle de notre classe `Personnage` ? Où placer nos requêtes ?

La classe `Personnage` a pour rôle de **représenter** un personnage présent en BDD. Elle n'a en aucun cas pour rôle de les **gérer.** Cette gestion sera le rôle d'une autre classe, communément appelée **manager.** Dans notre cas, notre gestionnaire de personnage sera tout simplement nommée `PersonnagesManager`.

Comment va-t-on faire pour construire ces classes ? Quelles questions va-t-on se poser ?

- Quelles seront les caractéristiques d'un manager ?
- Quelles seront les fonctionnalités d'un manager ?

Les caractéristiques d'un manager

Encore une fois, ce point a été abordé dans la troisième partie du précédent chapitre. Allez y faire un tour si vous avez un trou de mémoire ! Voici le code de la classe contenant sa (grande) liste d'attributs.

```php
<?php
class PersonnagesManager
{
  private $_db;
}
```

Les fonctionnalités d'un manager

Dans la troisième partie du précédent chapitre, nous avons vu quelques fonctionnalités de base. Notre manager pouvait :

- Enregistrer un nouveau personnage ;
- modifier un personnage ;

- supprimer un personnage ;
- sélectionner un personnage.

Cependant, ici, nous pouvons ajouter quelques fonctionnalités qui pourront nous être utiles :

- Compter le nombre de personnages ;
- récupérer une liste de plusieurs personnages ;
- savoir si un personnage existe.

Cela nous fait ainsi 7 méthodes à implémenter !



Rappels du précédent chapitre : n'oubliez pas d'ajouter un *setter* pour notre manager afin de pouvoir modifier l'attribut `$_db`. La création d'un constructeur sera aussi indispensable si nous voulons assigner à cet attribut un objet PDO dès l'instanciation du manager.

Comme d'habitude, écrivez le nom des méthodes en ajoutant des commentaires sur ce que doit faire la méthode.

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

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

  public function add(Personnage $perso)
  {
    // Préparation de la requête d'insertion.
    // Assignation des valeurs pour le nom du personnage.
    // Exécution de la requête.

    // Hydratation du personnage passé en paramètre avec assignation de son identifiant et des dégâts initiaux (= 0).
  }

  public function count()
  {
    // Exécute une requête COUNT() et retourne le nombre de résultats retourné.
  }

  public function delete(Personnage $perso)
  {
    // Exécute une requête de type DELETE.
  }

  public function exists($info)
  {
    // Si le paramètre est un entier, c'est qu'on a fourni un identifiant.
      // On exécute alors une requête COUNT() avec une clause WHERE, et on retourne un boolean.

    // Sinon c'est qu'on a passé un nom.
    // Exécution d'une requête COUNT() avec une clause WHERE, et retourne un boolean.
  }

  public function get($info)
  {
    // Si le paramètre est un entier, on veut récupérer le personnage avec son identifiant.
      // Exécute une requête de type SELECT avec une clause WHERE, et retourne un objet Personnage.

    // Sinon, on veut récupérer le personnage avec son nom.
    // Exécute une requête de type SELECT avec une clause WHERE, et retourne un objet Personnage.
  }

  public function getList($nom)
  {
    // Retourne la liste des personnages dont le nom n'est pas $nom.
    // Le résultat sera un tableau d'instances de Personnage.
  }

  public function update(Personnage $perso)
  {
    // Prépare une requête de type UPDATE.
    // Assignation des valeurs à la requête.
    // Exécution de la requête.
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Codons le tout !

Normalement, l'écriture des méthodes devrait être plus facile que dans la précédente partie. En effet, ici, il n'y a que des requêtes à écrire : si vous savez utiliser PDO, vous ne devriez pas avoir de mal. 

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

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

  public function add(Personnage $perso)
  {
    $q = $this->_db->prepare('INSERT INTO personnages SET nom = :nom');
    $q->bindValue(':nom', $perso->nom());
    $q->execute();

    $perso->hydrate([
      'id' => $this->_db->lastInsertId(),
      'degats' => 0,
    ]);
  }

  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 existe.
    {
      return (bool) $this->_db->query('SELECT COUNT(*) FROM personnages WHERE id = '.$info)->fetchColumn();
    }

    // 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([':nom' => $info]);

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

  public function get($info)
  {
    if (is_int($info))
    {
      $q = $this->_db->query('SELECT id, nom, degats FROM personnages WHERE id = '.$info);
      $donnees = $q->fetch(PDO::FETCH_ASSOC);

      return new Personnage($donnees);
    }
    else
    {
      $q = $this->_db->prepare('SELECT id, nom, degats FROM personnages WHERE nom = :nom');
      $q->execute([':nom' => $info]);

      return new Personnage($q->fetch(PDO::FETCH_ASSOC));
    }
  }

  public function getList($nom)
  {
    $persos = [];

    $q = $this->_db->prepare('SELECT id, nom, degats FROM personnages WHERE nom <> :nom ORDER BY nom');
    $q->execute([':nom' => $nom]);

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      $persos[] = new Personnage($donnees);
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->_db->prepare('UPDATE personnages SET degats = :degats WHERE id = :id');

    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Troisième étape : utilisation des classes

J'ai le plaisir de vous annoncer que vous avez fait le plus gros du travail ! Maintenant, nous allons juste utiliser nos classes en les instanciant et en invoquant les méthodes souhaitées sur nos objets. Le plus difficile ici est de se mettre d'accord sur le déroulement du jeu.

Celui-ci étant simple, nous n'aurons besoin que d'un seul fichier. Commençons par le début : que doit afficher notre mini-jeu lorsqu'on ouvre la page pour la première fois ? Il doit afficher un petit formulaire nous demandant le nom du personnage qu'on veut créer ou utiliser.

php
```php
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
  </body>
</html>
```

Vient ensuite la partie traitement. Deux cas peuvent se présenter :

- Le joueur a cliqué sur **Créer ce personnage**. Le script devra créer un objet `Personnage` en passant au constructeur un tableau contenant une entrée (le nom du personnage). Il faudra ensuite s'assurer que le personnage ait un nom valide et qu'il n'existe pas déjà. Après ces vérifications, l'enregistrement en BDD pourra se faire.

- Le joueur a cliqué sur **Utiliser ce personnage**. Le script devra vérifier si le personnage existe bien en BDD. Si c'est le cas, on le récupère de la BDD.



Pour savoir si le nom du personnage est valide, il va falloir implémenter une méthode `nomValide()` (je vous laisse réfléchir à quelle classe) qui retournera `true` ou `false` suivant si le nom est valide ou pas. Un nom est valide s'il n'est pas vide.

Cependant, avant de faire cela, il va falloir préparer le terrain.

- Un *autoload* devra être créé (bien que non indispensable puisqu'il n'y a que deux classes).

- Une instance de PDO devra être créée.

- Une instance de notre manager devra être créée.

Puisque notre manager a été créé, pourquoi ne pas afficher en haut de page le nombre de personnages créés ? 

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

spl_autoload_register('chargerClasse');

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

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un personnage.
{
  $perso = new Personnage(['nom' => $_POST['nom']]); // On crée un nouveau 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
  {
    $manager->add($perso);
  }
}

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

```
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
  </body>
</html>
```

Au cas où, je vous donne la méthode `nomValide()` de la classe `Personnage`. J'espère cependant que vous y êtes arrivés, un simple contrôle avec `empty()` et le tour est joué.

html+php

```
<?php
class Personnage
{
  // ...

  public function nomValide()
  {
    return !empty($this->_nom);
  }

  // ...
}
```

Une fois que nous avons un personnage, que se passera-t-il ? Il faut en effet cacher ce formulaire et laisser place à d'autres informations. Je vous propose d'afficher, dans un premier temps, les informations du personnage sélectionné (son nom et ses dégâts), puis, dans un second temps, la liste des autres personnages avec leurs informations. Il devra être possible de cliquer sur le nom du personnage pour le frapper.

php

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

spl_autoload_register('chargerClasse');

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

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un personnage.
{
  $perso = new Personnage(['nom' => $_POST['nom']]); // On crée un nouveau 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
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
```

```php
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?= htmlspecialchars($perso->nom()) ?><br />
        Dégâts : <?= $perso->degats() ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
    echo '<a href="?frapper=', $unPerso->id(), '">',
htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso->degats(), ')<br />';
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
```

Maintenant, quelque chose devrait vous titiller. En effet, si on recharge la page, on atterrira à nouveau sur le formulaire. Nous allons donc devoir utiliser le système de **sessions**. La première chose à faire sera alors de démarrer la session au début du script, juste après la déclaration de l'*autoload*. La session démarrée, nous pouvons aisément sauvegarder notre personnage. Pour cela, il nous faudra enregistrer le personnage en session (admettons dans `$_SESSION['perso']`) tout à la fin du code. Cela nous permettra, au début du script, de récupérer le personnage sauvegardé et de continuer le jeu.



Pour des raisons pratiques, il est préférable d'ajouter un lien de déconnexion pour qu'on puisse utiliser un autre personnage si on le souhaite. Ce lien aura pour effet de conduire à un `session_destroy()`.

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

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

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

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un
```

```php
$manager = new PersonnageManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un personnage.
{
  $perso = new Personnage(['nom' => $_POST['nom']]); // On crée un nouveau 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
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?= htmlspecialchars($perso->nom()) ?><br />
        Dégâts : <?= $perso->degats() ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
    echo '<a href="?frapper=', $unPerso->id(), '">', htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso->degats(), ')<br />';
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

Il reste maintenant une dernière partie à développer : celle qui s'occupera de frapper un personnage. Puisque nous avons déjà écrit tout le code faisant l'interaction entre l'attaquant et la cible, vous verrez que nous n'aurons presque rien à écrire.

Comment doit se passer la phase de traitement ? Avant toute chose, il faut bien vérifier que le joueur est connecté et que la variable `$perso` existe, sinon nous n'irons pas bien loin. Seconde vérification : il faut demander à notre manager si le personnage que l'on veut frapper existe bien. Si ces deux conditions sont vérifiées, alors on peut lancer l'attaque.

Pour lancer l'attaque, il va falloir récupérer le personnage à frapper grâce à notre manager. Ensuite, il suffira d'invoquer la méthode permettant de frapper le personnage. 

Cependant, nous n'allons pas nous arrêter là. N'oubliez pas que cette méthode peut retourner 3 valeurs différentes :

- `Personnage::CEST_MOI`. Le personnage a voulu se frapper lui-même.
- `Personnage::PERSONNAGE_FRAPPE`. Le personnage a bien été frappé.
- `Personnage::PERSONNAGE_TUE`. Le personnage a été tué.

Il va donc falloir afficher un message en fonction de cette valeur retournée. Aussi, seuls 2 de ces cas nécessitent une mise à jour de la BDD : si le personnage a été frappé ou s'il a été tué. En effet, si on a voulu se frapper soi-même, aucun des deux personnages impliqués n'a été modifié.

php

```php
<?php
// On enregistre notre autoload.
function chargerClasse($classname)
{
  require $classname.'.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 chaque fois qu'une requête a échoué.

$manager = new PersonnagesManager($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 personnage.
{
  $perso = new Personnage(['nom' => $_POST['nom']]); // On crée un nouveau 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
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}

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']))
    {
```

```php
      $message = 'Le personnage que vous voulez frapper n\'existe pas !';
    }

    else
    {
      $persoAFrapper = $manager->get((int) $_GET['frapper']);

      $retour = $perso->frapper($persoAFrapper); // On stocke dans $retour les éventuelles erreurs ou messages que renvoie la méthode frapper.

      switch ($retour)
      {
        case Personnage::CEST_MOI :
          $message = 'Mais... pourquoi voulez-vous vous frapper ???';
          break;

        case Personnage::PERSONNAGE_FRAPPE :
          $message = 'Le personnage a bien été frappé !';

          $manager->update($perso);
          $manager->update($persoAFrapper);

          break;

        case Personnage::PERSONNAGE_TUE :
          $message = 'Vous avez tué ce personnage !';

          $manager->update($perso);
          $manager->delete($persoAFrapper);

          break;
      }
    }
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?= htmlspecialchars($perso->nom()) ?><br />
        Dégâts : <?= $perso->degats() ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
  {
    echo '<a href="?frapper=', $unPerso->id(), '">', htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso->degats(), ')<br />';
  }
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

Et voilà, vous avez un jeu opérationnel.

Améliorations possibles

Ce code est très basique, beaucoup d'améliorations sont possibles. En voici quelques unes :

- Un système de **niveau**. Vous pourriez très bien assigner à chaque personnage un niveau de 1 à 100. Le personnage bénéficierait aussi d'une expérience allant de 0 à 100. Lorsque l'expérience atteint 100, le personnage passe au niveau suivant.

  Indice : le **niveau** et l'**expérience** deviendraient des caractéristiques du personnage, donc... Pas besoin de vous le dire, je suis sûr que vous savez ce que ça signifie !

- Un système de **force**. La force du personnage pourrait augmenter en fonction de son niveau, et les dégâts infligés à la victime seront donc plus importants.

  Indice : de même, la **force** du personnage serait aussi une caractéristique du personnage.

- Un système de **limitation**. En effet, un personnage peut en frapper autant qu'il veut dans un laps de temps indéfini. Pourquoi ne pas le limiter à 3 coups par jour ?

  Indice : il faudrait que vous stockiez le **nombre de coups** portés par le personnage, ainsi que la **date du dernier coup porté**. Cela ferait donc deux nouveaux champs en BDD, donc deux nouvelles caractéristiques pour le personnage !

- Un système de **retrait de dégâts**. Chaque jour, si l'utilisateur se connecte, il pourrait voir ses dégâts se soustraire de 10 par exemple.

  Indice : il faudrait stocker la date de dernière connexion. À chaque connexion, vous regarderiez cette date. Si elle est inférieure à 24h, alors vous ne feriez rien. Sinon, vous retireriez 10 de dégâts au personnage puis mettriez à jour cette date de dernière connexion.

Et la liste peut être longue ! Je vous encourage vivement à essayer d'implémenter ces fonctionnalités et à laisser libre court à votre imagination, vous progresserez bien plus !

☐ J'ai terminé ce chapitre et je passe au suivant

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

}

class Magicien extends Personnage // Notre classe Magicien hérite des attributs et méthodes de Personnage.
{

}
```

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 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**.

```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… Très utile ! En plus, tout ce qui est privé je n'y ai pas accès donc…

Nous sommes d'accord, si l'héritage c'était ça, 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 nous avons appris jusqu'à maintenant. Un exemple ?

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

  public function lancerUnSort($perso)
  {
    $perso->recevoirDegats($this->_magie); // On va dire que la magie du magicien représente sa force.
  }
}
```

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, seulement une notice si vous les avez activées disant que l'attribut n'existe pas.

Redéfinir 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 **réécrire** 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.

Nous allons maintenant essayer de redéfinir la méthode `gagnerExperience` afin de modifier l'attribut stockant la magie (admettons **$_magie**) lorsque, justement, on gagne de l'expérience. Problème : si on la réécrit, nous écrasons 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, suivi lui-même du nom de la méthode à appeler.

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

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

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

    if ($this->_magie < 100)
    {
      $this->_magie += 10;
    }
  }
}
```

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

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

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

    echo $retour;
  }
}

$b = new B;
$b->test(); // 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 non 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 redéfinissez une méthode, sa visibilité doit être la même ou plus permissive que dans la classe parente ! Si tel n'est pas le cas, une erreur fatale sera levée. Par exemple, vous ne pouvez redéfinir une méthode publique en disant qu'elle est privée, mais vous pouvez redéfinir une méthode privée en disant qu'elle est privée ou publique.

Héritez à l'infini !

Toute classe en POO peut être héritée si elle ne spécifie pas le contraire, **vraiment toutes.** Vous pouvez ainsi reproduire un arbre réel 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 :

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

}

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

}

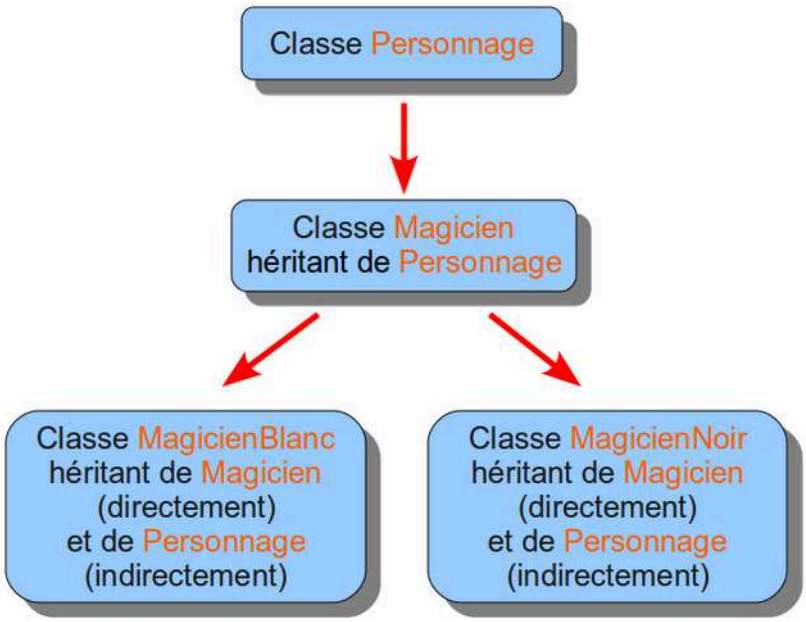
class MagicienBlanc extends Magicien // Classe MagicienBlanc héritant de Magicien.
{

}

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

}
```

Et un petit schéma qui reproduit ce code (figure suivante).



Exemple d'un héritage multiple

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 :

```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 celui-ci.
    echo $this->_attributPrive; // L'attribut est privé, on n'a pas accès celui-ci, donc rien ne s'affichera (mis à part une notice si vous les avez activées).
  }
}

$obj = new ClasseFille;

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

$obj->afficherAttributs(); // Affiche « Hello world ! » suivi de rien du tout ou d'une notice si vous les avez activées.
```

Comme vous pouvez le constater, il n'y a pas d'*underscores* précédant les noms d'éléments protégés. C'est encore une fois la notation PEAR qui nous dit que les noms d'éléments protégés ne sont pas précédés de ce caractère.

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 (utile si vous comptez partager votre 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.**

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 ?

Nous allons donc considérer la classe `Personnage` comme étant une classe **modèle** dont toute classe fille possédera les méthodes et attributs.

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

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

}

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

}

$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, il 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 + `function` + nomDeLaMethode + parenthèses avec ou sans paramètres + **point-virgule**).

```php
<?php
abstract class Personnage
{
  // On va forcer toute classe fille à écrire cette méthode car chaque personnage frappe différemment.
  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 abstraite « frapper » de la classe mère.
  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`.

```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 redéfinir**. Si vous déclarez votre méthode `recevoirDegats` en tant que méthode finale, vous ne pourrez la redéfinir.

```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

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.

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

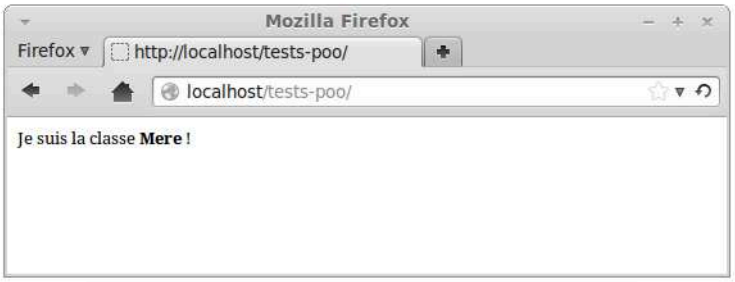
```php
<?php
class Mere
{
  public static function lancerLeTest()
  {
    self::quiEstCe();
  }

  public 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 :



Résultat affiché par le script

Mais qu'est-ce qu'il s'est passé ???

- Appel de la méthode `lancerLeTest` de la classe `Enfant` ;
- la méthode n'a pas été réécrite, on va donc « chercher » la méthode `lancerLeTest` de la classe mère ;
- appel de la méthode `quiEstCe` **de la 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 récrite ?

Tout simplement parce que `self::` fait appel à la méthode statique **de la classe dans laquelle est contenu `self::`**, donc de la classe parente.

Et la résolution statique à la volée dans tout ça ?

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` !

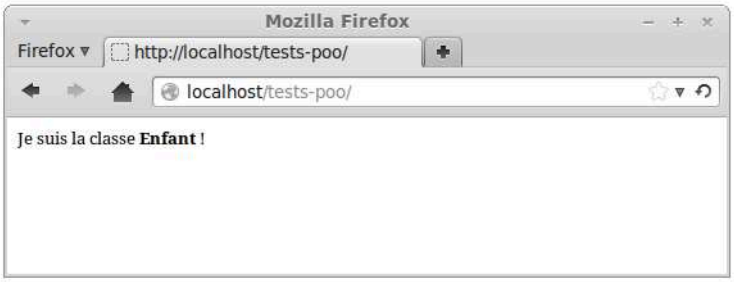
```php
<?php
class Mere
{
  public static function lancerLeTest()
  {
    static::quiEstCe();
  }

  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();
```

Ce qui donnera :



Résultat affiché par le script

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

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

  public function quiEstCe()
  {
    echo 'Je suis la classe « Mere » !';
  }
}

class Enfant extends Mere
{
  public function quiEstCe()
  {
    echo 'Je suis la classe « Enfant » !';
  }
}

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

Cas complexes

Au premier abord, vous n'avez peut-être pas tout compris. Si tel est le cas, ne lisez pas la suite cela risquerait de vous embrouiller davantage. Prenez bien le temps de comprendre ce qui est écrit 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 :

```php
<?php
class A
{

}

class B extends A
{

}

class C extends B
{

}
```

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

```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';
  }
}
```

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

Appelons maintenant cette méthode depuis la classe C dans un contexte statique (nul besoin de créer d'objet, mais ça marche tout aussi bien 😉).

```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();
```

Plaçons un peu de code dans cette méthode, sinon ce n'est pas drôle.

Nous allons 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 » :

```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, créons 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 ! 😉

```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 très compliqué mais fondamental à comprendre.

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

```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 ? Et 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, le code suivant affichera « A ».

```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();
```

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 :

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

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

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

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

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

$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.

En résumé

- Nous pouvons parler d'héritage entre une classe A et une classe B si et seulement si nous pouvons dire « B est un A » (dans ce cas-là, B hérite de A).
- Une classe héritant d'une autre a accès à tous ses attributs et méthodes publics et protégés.
- La visibilité `protected` est équivalente à la visibilité `private` à la différence près qu'un élément protégé est accessible par les classes filles, contrairement aux éléments privés.
- Il est possible d'interdire l'instanciation d'une classe grâce au mot-clé `abstract` (utile pour poser un modèle de base commun à plusieurs classes) et l'héritage d'une classe grâce au mot-clé `final`.
- Il est possible d'obliger ses classes filles à implémenter une méthode grâce au mot-clé `abstract` et de leur interdire la réécriture d'une méthode grâce au mot-clé `final`.
- La résolution statique à la volée permet de savoir quelle classe a été initialement appelée afin d'invoquer des méthodes ou accéder à des attributs de la classe appelée.

☐ J'ai terminé ce chapitre et je passe au suivant

TP : DES PERSONNAGES SPÉCIALISÉS

Avez-vous bien tout retenu du dernier chapitre ? C'est ce que l'on va vérifier maintenant ! En effet, l'héritage est un point très important de la POO qu'il est essentiel de maîtriser, c'est pourquoi je souhaite insister dessus à travers ce TP. Ce dernier est en fait une modification du premier (celui qui mettait en scène notre personnage). Nous allons donc ajouter une possibilité supplémentaire au script : le choix du personnage. Cette modification vous permettra de revoir ce que nous avons abordé depuis le dernier TP, à savoir :

- L'héritage ;
- la portée `protected` ;
- l'abstraction ;
- la résolution statique à la volée.

Ce que nous allons faire

Cahier des charges

Je veux que nous ayons 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 il devra être impossible d'instancier la classe `Personnage`). 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 nouvelle fonctionnalité : celle de lancer un sort qui aura pour effet d'endormir un personnage pendant `$atout * 6` heures (l'attribut `$atout` représente la dose de magie du personnage).
- Un guerrier. Lorsqu'un coup lui est porté, il devra avoir la possibilité de parer le coup en fonction de sa protection (son atout).

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

Comme vous le voyez, chaque personnage possède un **atout**. Cet atout devra être augmenté lorsque le personnage est amené à s'en servir (c'est-à-dire lorsque le magicien lance un sort ou que le guerrier subit des dégâts).

Des nouvelles fonctionnalités pour chaque personnage

Étant donné qu'un magicien peut endormir un personnage, il est nécessaire d'implémenter deux nouvelles fonctionnalités :

- Celle consistant à savoir si un personnage est endormi ou non (nécessaire lorsque ledit personnage voudra en frapper un autre : s'il est endormi, ça ne doit pas être possible).
- Celle consistant à obtenir la date du réveil du personnage sous la forme « XX heures, YY minutes et ZZ secondes », qui s'affichera dans le cadre d'information du personnage s'il est endormi.

La base de données

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 ou guerrier). Nous allons donc créer une colonne **type** et une colonne **atout** (l'attribut qu'il a en plus). Une colonne **timeEndormi** 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) :

```mysql
CREATE TABLE IF NOT EXISTS `personnages_v2` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `nom` varchar(50) COLLATE utf8_general_ci NOT NULL,
  `degats` tinyint(3) unsigned NOT NULL DEFAULT '0',
  `timeEndormi` int(10) unsigned NOT NULL DEFAULT '0',
  `type` enum('magicien','guerrier') COLLATE utf8_general_ci NOT NULL,
  `atout` tinyint(3) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_general_ci;
```

Le coup de pouce du démarrage

Les modifications que je vous demande peuvent vous donner mal à la tête, c'est pourquoi je me propose de vous mettre sur la voie. Procédons classe par classe.

Le personnage

Là où il y a peut-être une difficulté, c'est pour déterminer l'attribut `$type`. En effet, où doit-on assigner cette valeur ? Qui doit le faire ?

Pour des raisons évidentes de risques de bugs, ce ne sera pas à l'utilisateur d'assigner la valeur « guerrier » ou « magicien » à cet attribut, mais à la classe elle-même. Cependant, il serait redondant dans chaque classe fille de **Personnage** de faire un `<?php $this->type = 'magicien'` par exemple, alors on va le faire une bonne fois pour toute dans la classe **Personnage**.

Comment cela est-il possible ?

Je vais vous dévoiler une petite fonctionnalité de PHP. Dans toutes les méthodes d'une classe, il est possible d'accéder à son nom grâce à `::class`. Pour ce qui est à placer avant ce double deux-points, cela dépend de ce que vous voulez obtenir. Généralement, vous aurez le choix entre `self` et `static`. Si vous avez bien compris la partie sur la résolution statique à la volée, vous avez peut-être compris là où je voulais en venir. Essayez ce code :

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

class Fille extends Mere
{

}

new Fille;
```

À l'écran s'affichera « Fille », car il s'agit de la classe que l'on a appelée. Par conséquent, on souhaite afficher le nom de la classe **Fille**. Si l'on avait mis `self::class`, on aurait affiché le nom de la classe **Mere**.

Cette technique de détermination du nom de classe n'est disponible que depuis la version 5.5 de PHP. Si vous avez une version antérieure, vous pouvez utiliser get_called_class qui retournera exactement le même résultat (cette fonction s'appuie elle aussi sur le late static binding).

Pour des raisons pratiques, le type du personnage doit être en minuscules. Un petit appel à strtolower() sera donc indispensable.

Le magicien

Qui dit nouvelle fonctionnalité dit nouvelle méthode. Votre classe `Magicien` devra donc implémenter une nouvelle méthode permettant de lancer un sort. Celle-ci devra vérifier plusieurs points :

- La cible à ensorceler n'est pas le magicien qui lance le sort.
- Le magicien possède encore de la magie (l'atout n'est pas à 0).

Le guerrier

Ce qu'on cherche à faire ici est **modifier** le comportement du personnage lorsqu'il subit des dégâts. Nous allons donc **modifier** la méthode qui se charge d'ajouter des dégâts au personnage. Cette méthode procédera de la sorte :

- Elle calculera d'abord la valeur de l'atout.
- Elle augmentera les dégâts en prenant soin de prendre en compte l'atout.
- Elle indiquera si le personnage a été frappé ou tué.

Tu nous parles d'un atout depuis tout à l'heure, mais comment est-ce qu'on le détermine ?

L'atout du magicien et du guerrier se déterminent de la même façon :

- Si les dégâts sont compris entre 0 et 25, alors l'atout sera de 4.
- Si les dégâts sont compris entre 25 et 50, alors l'atout sera de 3.
- Si les dégâts sont compris entre 50 et 75, alors l'atout sera de 2.
- Si les dégâts sont compris entre 75 et 90, alors l'atout sera de 1.
- Sinon, il sera de 0.

Comment sont-ils exploités ?

Du côté du guerrier, j'utilise une simple formule : la dose de dégâts reçu ne sera pas de 5, mais de `5 - $atout`. Du côté du magicien, là aussi j'utilise une simple formule : il endort sa victime pendant `($this->atout * 6) * 3600` secondes.

Voir le résultat que vous devez obtenir

Comme au précédent TP, le résultat comporte toutes les améliorations proposées en fin de chapitre.

Correction

Nous allons maintenant corriger le TP. Les codes seront d'abord précédés d'explications pour bien mettre au clair ce qui était demandé.

Commençons d'abord par notre classe `Personnage`. Celle-ci devait implémenter deux nouvelles méthodes (sans compter les *getters* et *setters*) : `estEndormi()` et `reveil()`. Aussi il ne fallait pas oublier de modifier la méthode `frapper()` afin de bien vérifier que le personnage qui frappe n'était pas endormi ! Enfin, il fallait assigner la bonne valeur à l'attribut `$type` dans le constructeur.

Il ne fallait pas mettre de *setter* pour l'attribut `$type`. En effet, le type d'un personnage est constant (un magicien ne se transforme pas en guerrier). Il est défini dans le constructeur et l'utilisateur ne pourra pas changer sa valeur (imaginez le non-sens qu'il y aurait si l'utilisateur mettait « magicien » à l'attribut `$type` d'un objet `Guerrier`).

```php
<?php
abstract class Personnage
{
  protected $atout,
            $degats,
            $id,
            $nom,
            $timeEndormi,
            $type;

  const CEST_MOI = 1; // Constante renvoyée par la méthode `frapper` si on se frappe soit-même.
  const PERSONNAGE_TUE = 2; // Constante renvoyée par la méthode `frapper` si on a tué le personnage en le frappant.
  const PERSONNAGE_FRAPPE = 3; // Constante renvoyée par la méthode `frapper` si on a bien frappé le personnage.
  const PERSONNAGE_ENSORCELE = 4; // Constante renvoyée par la méthode `lancerUnSort` (voir classe Magicien) si on a bien ensorcelé un personnage.
  const PAS_DE_MAGIE = 5; // Constante renvoyée par la méthode `lancerUnSort` (voir classe Magicien) si on veut jeter un sort alors que la magie du magicien est à 0.
  const PERSO_ENDORMI = 6; // Constante renvoyée par la méthode `frapper` si le personnage qui veut frapper est endormi.

  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
    $this->type = strtolower(static::class);
  }

  public function estEndormi()
  {
    return $this->timeEndormi > time();
  }

  public function frapper(Personnage $perso)
  {
    if ($perso->id == $this->id)
    {
      return self::CEST_MOI;
    }

    if ($this->estEndormi())
    {
      return self::PERSO_ENDORMI;
    }

    // On indique au personnage qu'il doit recevoir des dégâts.
    // Puis on retourne la valeur renvoyée par la méthode : self::PERSONNAGE_TUE ou self::PERSONNAGE_FRAPPE.
    return $perso->recevoirDegats();
  }

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  public function nomValide()
  {
    return !empty($this->nom);
  }

  public function recevoirDegats()
  {
    $this->degats += 5;

    // 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;
  }

  public function reveil()
  {
    $secondes = $this->timeEndormi;
    $secondes -= time();
```

```php
    $secondes = $this->timeEndormi;
    $secondes -= time();

    $heures = floor($secondes / 3600);
    $secondes -= $heures * 3600;
    $minutes = floor($secondes / 60);
    $secondes -= $minutes * 60;

    $heures .= $heures <= 1 ? ' heure' : ' heures';
    $minutes .= $minutes <= 1 ? ' minute' : ' minutes';
    $secondes .= $secondes <= 1 ? ' seconde' : ' secondes';

    return $heures . ', ' . $minutes . ' et ' . $secondes;
  }

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

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

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

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

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

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

  public function setAtout($atout)
  {
    $atout = (int) $atout;

    if ($atout >= 0 && $atout <= 100)
    {
      $this->atout = $atout;
    }
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->degats = $degats;
    }
  }

  public function setId($id)
  {
    $id = (int) $id;

    if ($id > 0)
    {
      $this->id = $id;
    }
  }

  public function setNom($nom)
  {
    if (is_string($nom))
    {
      $this->nom = $nom;
    }
  }

  public function setTimeEndormi($time)
  {
    $this->timeEndormi = (int) $time;
  }
}
```

Penchons-nous maintenant vers la classe `Guerrier`. Celle-ci devait modifier la méthode `recevoirDegats()` afin d'ajouter une parade lors d'une attaque.

```php
<?php
class Guerrier extends Personnage
{
  public function recevoirDegats()
  {
    if ($this->degats >= 0 && $this->degats <= 25)
    {
      $this->atout = 4;
    }
    elseif ($this->degats > 25 && $this->degats <= 50)
    {
      $this->atout = 3;
    }
    elseif ($this->degats > 50 && $this->degats <= 75)
    {
      $this->atout = 2;
    }
    elseif ($this->degats > 50 && $this->degats <= 90)
    {
      $this->atout = 1;
    }
    else
    {
      $this->atout = 0;
    }

    $this->degats += 5 - $this->atout;

    // 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;
  }
}
```

Enfin, il faut maintenant s'occuper du magicien. La classe le représentant devait ajouter une nouvelle fonctionnalité : celle de pouvoir lancer un sort.

```php
<?php
class Magicien extends Personnage
{
  public function lancerUnSort(Personnage $perso)
  {
    if ($this->degats >= 0 && $this->degats <= 25)
    {
      $this->atout = 4;
    }
    elseif ($this->degats > 25 && $this->degats <= 50)
    {
      $this->atout = 3;
    }
    elseif ($this->degats > 50 && $this->degats <= 75)
    {
      $this->atout = 2;
    }
    elseif ($this->degats > 50 && $this->degats <= 90)
    {
      $this->atout = 1;
    }
    else
    {
      $this->atout = 0;
    }

    if ($perso->id == $this->id)
    {
      return self::CEST_MOI;
    }

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

    if ($this->estEndormi())
    {
      return self::PERSO_ENDORMI;
    }

    $perso->timeEndormi = time() + ($this->atout * 6) * 3600;

    return self::PERSONNAGE_ENSORCELE;
  }
}
```

Passons maintenant au manager. Nous allons toujours garder un seul manager parce que nous gérons toujours des personnages ayant la même structure. Les modifications sont mineures : il faut juste ajouter les deux nouveaux champs dans les requêtes et instancier la bonne classe lorsqu'on récupère un personnage.

```php
<?php
class PersonnagesManager
{
  private $db; // Instance de PDO

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

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

    $q->bindValue(':nom', $perso->nom());
    $q->bindValue(':type', $perso->type());

    $q->execute();

    $perso->hydrate([
      'id' => $this->db->lastInsertId(),
      'degats' => 0,
      'atout' => 0
    ]);
  }

  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 existe.
    {
      return (bool) $this->db->query('SELECT COUNT(*) FROM personnages_v2 WHERE id = '.$info)->fetchColumn();
    }

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

    $q = $this->db->prepare('SELECT COUNT(*) FROM personnages_v2 WHERE nom = :nom');
    $q->execute([':nom' => $info]);

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

  public function get($info)
  {
    if (is_int($info))
    {
      $q = $this->db->query('SELECT id, nom, degats, timeEndormi, type, atout FROM personnages_v2 WHERE id = '.$info);
      $perso = $q->fetch(PDO::FETCH_ASSOC);
    }

    else
    {
      $q = $this->db->prepare('SELECT id, nom, degats, timeEndormi, type, atout FROM personnages_v2 WHERE nom = :nom');
      $q->execute([':nom' => $info]);

      $perso = $q->fetch(PDO::FETCH_ASSOC);
    }

    switch ($perso['type'])
    {
      case 'guerrier': return new Guerrier($perso);
      case 'magicien': return new Magicien($perso);
      default: return null;
    }
  }

  public function getList($nom)
  {
    $persos = [];

    $q = $this->db->prepare('SELECT id, nom, degats, timeEndormi, type, atout FROM personnages_v2 WHERE nom <> :nom ORDER BY nom');
    $q->execute([':nom' => $nom]);

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      switch ($donnees['type'])
      {
        case 'guerrier': $persos[] = new Guerrier($donnees); break;
        case 'magicien': $persos[] = new Magicien($donnees); break;
      }
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->db->prepare('UPDATE personnages_v2 SET degats = :degats, timeEndormi = :timeEndormi, atout = :atout WHERE id = :id');

    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':timeEndormi', $perso->timeEndormi(), PDO::PARAM_INT);
    $q->bindValue(':atout', $perso->atout(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);
```

```
    $q->bindValue(':atout', $perso->atout(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }
}
```

Enfin, finissons par la page d'index qui a légèrement changé. Quelles sont les modifications ?

- Le formulaire doit proposer au joueur de choisir le type du personnage qu'il veut créer.
- Lorsqu'on crée un personnage, le script doit créer une instance de la classe désirée et non de `Personnage`.
- Lorsqu'on veut frapper un personnage, on vérifie que l'attaquant n'est pas endormi.
- Un lien permettant d'ensorceler un personnage doit être ajouté pour les magiciens ainsi que l'antidote qui va avec.

php

```php
<?php
function chargerClasse($classe)
{
  require $classe . '.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 personnage.
{
  switch ($_POST['type'])
  {
    case 'magicien' :
      $perso = new Magicien(['nom' => $_POST['nom']]);
      break;

    case 'guerrier' :
      $perso = new Guerrier(['nom' => $_POST['nom']]);
      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
    {
      $manager->add($perso);
    }
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}

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.';
  }

```

```php
    $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
    {
      $persoAFrapper = $manager->get((int) $_GET['frapper']);
      $retour = $perso->frapper($persoAFrapper); // On stocke dans $retour les éventuelles erreurs ou messages que renvoie la méthode frapper.

      switch ($retour)
      {
        case Personnage::CEST_MOI :
          $message = 'Mais... pourquoi voulez-vous vous frapper ???';
          break;

        case Personnage::PERSONNAGE_FRAPPE :
          $message = 'Le personnage a bien été frappé !';

          $manager->update($perso);
          $manager->update($persoAFrapper);

          break;

        case Personnage::PERSONNAGE_TUE :
          $message = 'Vous avez tué ce personnage !';

          $manager->update($perso);
          $manager->delete($persoAFrapper);

          break;

        case Personnage::PERSO_ENDORMI :
          $message = 'Vous êtes endormi, vous ne pouvez pas frapper de personnage !';
          break;
      }
    }
  }
}

elseif (isset($_GET['ensorceler']))
{
  if (!isset($perso))
  {
    $message = 'Merci de créer un personnage ou de vous identifier.';
  }

  else
  {
    // Il faut bien vérifier que le personnage est un magicien.
    if ($perso->type() != 'magicien')
    {
      $message = 'Seuls les magiciens peuvent ensorceler des personnages !';
    }

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

      else
      {
        $persoAEnsorceler = $manager->get((int) $_GET['ensorceler']);
        $retour = $perso->lancerUnSort($persoAEnsorceler);

        switch ($retour)
        {
          case Personnage::CEST_MOI :
            $message = 'Mais... pourquoi voulez-vous vous ensorceler ???';
            break;

          case Personnage::PERSONNAGE_ENSORCELE :
            $message = 'Le personnage a bien été ensorcelé !';

            $manager->update($perso);
            $manager->update($persoAEnsorceler);

            break;

          case Personnage::PAS_DE_MAGIE :
            $message = 'Vous n\'avez pas de magie !';
            break;

          case Personnage::PERSO_ENDORMI :
            $message = 'Vous êtes endormi, vous ne pouvez pas lancer de sort !';
            break;
        }
      }
    }
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat - Version 2</title>

    <meta charset="utf-8" />
  </head>
  <body>
```

```php
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Type : <?= ucfirst($perso->type()) ?><br />
        Nom : <?= htmlspecialchars($perso->nom()) ?><br />
        Dégâts : <?= $perso->degats() ?><br />
<?php
// On affiche l'atout du personnage suivant son type.
switch ($perso->type())
{
  case 'magicien' :
    echo 'Magie : ';
    break;

  case 'guerrier' :
    echo 'Protection : ';
    break;
}

echo $perso->atout();
?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui attaquer ?</legend>
      <p>
<?php
// On récupère tous les personnages par ordre alphabétique, dont le nom est différent de celui de notre personnage (on va pas se frapper nous-même :p).
$retourPersos = $manager->getList($perso->nom());

if (empty($retourPersos))
{
  echo 'Personne à frapper !';
}

else
{
  if ($perso->estEndormi())
  {
    echo 'Un magicien vous a endormi ! Vous allez vous réveiller dans ', $perso->reveil(), '.';
  }

  else
  {
    foreach ($retourPersos as $unPerso)
    {
      echo '<a href="?frapper=', $unPerso->id(), '">', htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso->degats(), ' | type : ', $unPerso->type(), ')';

      // On ajoute un lien pour lancer un sort si le personnage est un magicien.
      if ($perso->type() == 'magicien')
      {
        echo ' | <a href="?ensorceler=', $unPerso->id(), '">Lancer un sort</a>';
      }

      echo '<br />';
    }
  }
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" /> <input type="submit" value="Utiliser ce personnage" name="utiliser" /><br />
        Type :
        <select name="type">
          <option value="magicien">Magicien</option>
          <option value="guerrier">Guerrier</option>
        </select>
        <input type="submit" value="Créer ce personnage" name="creer" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

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 !

Améliorations possibles

Comme au précédent TP, beaucoup d'améliorations sont possibles, à commencer par celles déjà exposées dans le chapitre dudit TP :

- Un système de **niveau**. Vous pourriez très bien assigner à chaque personnage un niveau de 1 à 100. Le personnage bénéficierait aussi d'une expérience allant de 0 à 100. Lorsque l'expérience atteint 100, le personnage passe au niveau suivant.

  Indice : le **niveau** et l'**expérience** deviendraient des caractéristiques du personnage, donc... Pas besoin de vous le dire, je suis sûr que vous savez ce que ça signifie !

- Un système de **force**. La force du personnage pourrait augmenter en fonction de son niveau, les dégâts infligés à la victime seront donc plus importants.

  Indice : de même, la **force** du personnage serait aussi une caractéristique du personnage.

- Un système de **limitation**. En effet, un personnage peut en frapper autant qu'il veut dans un laps de temps indéfini. Pourquoi ne pas le limiter à 3 coups par jour ?

  Indice : il faudrait que vous stockiez le **nombre de coups** portés par le personnage, ainsi que la **date du dernier coup porté**. Cela ferait donc deux nouveaux champs en BDD, et deux nouvelles caractéristiques pour le personnage !

- Un système de **retrait de dégâts**. Chaque jour, si l'utilisateur se connecte, il pourrait voir ses dégâts se soustraire de 10 par exemple.

  Indice : il faudrait stocker la date de dernière connexion. À chaque connexion, vous regarderiez cette date. Si elle est inférieure à 24h, alors vous ne feriez rien. Sinon, vous retireriez 10 de dégâts au personnage puis mettriez à jour cette date de dernière connexion.

Pour reprendre cet esprit, vous pouvez vous entraîner à créer d'autres personnages, comme une **brute** par exemple. Son atout dépendrait aussi de ses dégâts et viendrait augmenter sa force lors d'une attaque.

La seule différence avec le premier TP est l'apparition de nouveaux personnages, les seules améliorations différentes possibles sont donc justement la création de nouveaux personnages. Ainsi je vous encourage à imaginer tout un tas de différents personnages tous aussi farfelus les uns que les autres : cela vous fera grandement progresser et vous aidera à vous familiariser avec l'héritage !

☐ J'ai terminé ce chapitre et je passe au suivant

LES MÉTHODES MAGIQUES

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

Le principe

Vous devez sans doute vous poser une grande question à la vue du 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 évènement. Si la méthode n'existe pas et que l'évènement est exécuté, aucun effet « spécial » ne sera ajouté, l'évènement s'exécutera normalement. Le but des méthodes magiques est d'intercepter un évènement, dire de faire ceci ou cela et retourner une valeur utile pour l'évènement si besoin il y a.

Bonne nouvelle : vous connaissez déjà une méthode magique !

Si si, cherchez bien au fond de votre tête… Et oui, la méthode `__construct` est magique ! Comme nous l'avons vu plus haut, chaque méthode magique s'exécute au moment où un évènement est lancé. L'évènement 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 :

```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. Au même titre que `__construct` , `__destruct` a lui aussi un petit nom : il s'agit du destructeur.

La surcharge magique des attributs et méthodes

Parlons maintenant des méthode liées à la surcharge magique des attributs et méthodes.

Euh, deux secondes là… C'est quoi la « surcharge magique des attributs et méthodes » ?

Vous avez raison, il serait d'abord préférable d'expliquer ceci. La surcharge magique d'attributs ou méthodes consiste à créer dynamiquement des attributs et méthodes. Cela est possible lorsque l'on tente d'accéder à un élément qui n'existe pas ou auquel on n'a pas accès (s'il est privé et qu'on tente d'y accéder depuis l'extérieur de la classe par exemple). 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 lorsque l'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 simplement faire ce que bon vous semble !

Exemple :

```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 <strong>', $valeur, '</strong> mais c\'est pas possible !<br />';
  }
}

$obj = new MaClasse;

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

À la sortie s'affichera :



Résultat affiché par le script

Tenez, petit exercice, stockez dans un tableau tous les attributs (avec leurs valeurs) que nous avons essayé de modifier ou créer.

Solution :

```php
<?php
class MaClasse
{
  private $attributs = [];
  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 à faire, mais cela permet de pratiquer un peu.

Parlons maintenant de `__get`. Cette méthode est appelée lorsque l'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 inaccessible).

Exemple :

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

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

$obj = new MaClasse;

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

Ce qui va afficher :



Résultat affiché par le script

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 :

```php
<?php
class MaClasse
{
  private $attributs = [];
  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->autreAtribut;
```



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 :

```php
<?php
class MaClasse
{
  private $attributs = [];
  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 :



Résultat affiché par le script

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.

php

```php
<?php
class MaClasse
{
  private $attributs = [];
  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 :



Résultat affiché par le script

Finissons par « __call » et « __callStatic »

Laissons de côté les attributs pour le moment et parlons 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 que l'on a essayé d'appeler et le second est la liste des arguments qui lui ont été passés (sous forme de tableau).

Exemple :

```php
<?php
class MaClasse
{
  public function __call($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode($arguments, '</strong>, <strong>'), '</strong>';
  }
}

$obj = new MaClasse;

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

Résultat :



Résultat affiché par le script

Et si on essaye d'appeler une méthode qui n'existe pas statiquement ? Et 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` !

```php
<?php
class MaClasse
{
  public function __call($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode($arguments, '</strong>, <strong>'), '</strong><br />';
  }

  public static function __callStatic($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée dans un contexte statique alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode ($arguments, '</strong>, <strong>'), '</strong><br />';
  }
}

$obj = new MaClasse;

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

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

Résultat :



Résultat affiché par le script

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. Lisez le jusqu'à la partie **Encore plus fort !**, et je vais mieux vous expliquer à partir de là !

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 doutes 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 quel est 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 passe 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 `['serveur', 'login'];`.

Voici ce que pourrait donner notre classe `Connexion` :

```php
<?php
class Connexion
{
  protected $pdo, $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();
  }

  protected function connexionBDD()
  {
    $this->pdo = new PDO('mysql:host='.$this->serveur.';dbname='.$this->dataBase,
$this->utilisateur, $this->motDePasse);
  }

  public function __sleep()
  {
    // Ici sont à placer des instructions à exécuter juste avant la linéarisation.
    // On retourne ensuite la liste des attributs qu'on veut sauver.
    return ['serveur', 'utilisateur', 'motDePasse', 'dataBase'];
  }
}
```

Ainsi, vous pourrez faire ceci :

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

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

« unserialize » et « __wakeup »

Maintenant, nous allons simplement implémenter la fonction `__wakeup`. Qu'allons-nous mettre dedans ?

Rien de compliqué… Nous allons 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` !

```php
<?php
class Connexion
{
  protected $pdo, $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();
  }

  protected function connexionBDD()
  {
    $this->pdo = new PDO('mysql:host='.$this->serveur.';dbname='.$this->dataBase,
$this->utilisateur, $this->motDePasse);
  }

  public function __sleep()
  {
    return ['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 :

```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 l'objet désérialisé sera une instance de `__PHP_Incomplete_Class_Name`, classe qui ne contient aucune méthode (cela produira donc un objet inutile). 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`, `__invoke` et `__debugInfo`.

« __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 :

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

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

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

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

// Solution 1 : le cast

$texte = (string) $obj;
var_dump($texte); // Affiche : string(13) "Hello world !".

// Solution 2 : directement dans un echo
echo $obj; // 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 (`['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 :

```php
<?php
class Export
{
  protected $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 avec les attributs de l'objet que l'on veut exporter.
    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 ayant les mêmes attributs que l'objet précédent.

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

Le code affichera donc :



Résultat affiché par le script

« __invoke »

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 :

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

Essayez ce code et… BAM ! Une erreur fatale. Pour résoudre ce problème, nous allons 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 :

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

$obj = new MaClasse;

$obj(5); // Affiche « 5 ».
```

« __debugInfo »

> ✗ Cette méthode magique n'est disponible que depuis la version 5.6 de PHP.

Cette méthode magique est invoquée sur notre objet lorsqu'on appelle la fonction var_dump. Pour ceux qui ne connaissent pas son principe, cette fonction permet d'obtenir des informations sur la variable qu'on lui donne. Si on lui donne un objet, var_dump va afficher les détails de tous les attributs de l'objet, qu'ils soient publics, protégés ou privés. La méthode magique `__debugInfo` permet de modifier ce comportement en ne sélectionnant que les attributs à afficher ainsi que ce qu'il faut afficher. Pour ce faire, cette méthode renverra sous forme de tableau associatif la liste des attributs à afficher avec leurs valeurs. Voici un exemple d'utilisation (je n'ai volontairement pas implémenté beaucoup de méthodes pour ne pas encombrer l'exemple) :

```php
<?php
class FileReader
{
    protected $f;

    public function __construct($path)
    {
        $this->f = fopen($path, 'c+');
    }

    public function __debugInfo()
    {
        return ['f' => fstat($this->f)];
    }
}

$f = new FileReader('fichier.txt');
var_dump($f); // Affiche les informations retournées par fstat.
```

On a ici un gestionnaire de fichiers qui nous permet de gérer facilement la lecture et l'écriture d'un fichier (du moins, on imagine que ça le fait). Un `var_dump` sur notre objet ne serait pas très révélateur. Par contre, obtenir les informations sur le fichier actuellement ouvert le serait plus, et c'est précisément ce que l'on fait en écrivant la méthode `__debugInfo` avec l'appel de `fstat`.

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

En résumé

- Les méthodes magiques sont des méthodes qui sont appelées automatiquement lorsqu'un certain évènement est déclenché.
- Toutes les méthodes magiques commencent par deux *underscores*, évitez donc d'appeler vos méthodes suivant ce même modèle.
- Les méthodes magiques dont vous vous servirez le plus souvent sont `__construct`, `__set`, `__get` et `__call`. Les autres sont plus « gadget » et vous les rencontrerez moins souvent.

Le code affichera donc :



Résultat affiché par le script

« __invoke »

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 :

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

Essayez ce code et… BAM ! Une erreur fatale. Pour résoudre ce problème, nous allons 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 :

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

$obj = new MaClasse;

$obj(5); // Affiche « 5 ».
```

« __debugInfo »

> ✗ Cette méthode magique n'est disponible que depuis la version 5.6 de PHP.

Cette méthode magique est invoquée sur notre objet lorsqu'on appelle la fonction var_dump. Pour ceux qui ne connaissent pas son principe, cette fonction permet d'obtenir des informations sur la variable qu'on lui donne. Si on lui donne un objet, var_dump va afficher les détails de tous les attributs de l'objet, qu'ils soient publics, protégés ou privés. La méthode magique `__debugInfo` permet de modifier ce comportement en ne sélectionnant que les attributs à afficher ainsi que ce qu'il faut afficher. Pour ce faire, cette méthode renverra sous forme de tableau associatif la liste des attributs à afficher avec leurs valeurs. Voici un exemple d'utilisation (je n'ai volontairement pas implémenté beaucoup de méthodes pour ne pas encombrer l'exemple) :

```php
<?php
class FileReader
{
    protected $f;

    public function __construct($path)
    {
        $this->f = fopen($path, 'c+');
    }

    public function __debugInfo()
    {
        return ['f' => fstat($this->f)];
    }
}

$f = new FileReader('fichier.txt');
var_dump($f); // Affiche les informations retournées par fstat.
```

On a ici un gestionnaire de fichiers qui nous permet de gérer facilement la lecture et l'écriture d'un fichier (du moins, on imagine que ça le fait). Un `var_dump` sur notre objet ne serait pas très révélateur. Par contre, obtenir les informations sur le fichier actuellement ouvert le serait plus, et c'est précisément ce que l'on fait en écrivant la méthode `__debugInfo` avec l'appel de `fstat`.

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

En résumé

- Les méthodes magiques sont des méthodes qui sont appelées automatiquement lorsqu'un certain évènement est déclenché.
- Toutes les méthodes magiques commencent par deux *underscores*, évitez donc d'appeler vos méthodes suivant ce même modèle.
- Les méthodes magiques dont vous vous servirez le plus souvent sont `__construct`, `__set`, `__get` et `__call`. Les autres sont plus « gadget » et vous les rencontrerez moins souvent.

MANIPULATION DE DONNÉES STOCKÉES

Tout site web dynamique doit être capable de sauvegarder des données afin de les utiliser ultérieurement. Comme vous le savez sans doute, l'un des moyens les plus simples et efficaces est la base de données permettant de sauvegarder des centaines de données et de les récupérer en un temps très court.

Cependant, la gestion des données stockées en BDD peut être assez difficile à cerner en POO. Le débutant ne sait en général pas par où commencer et se pose tout un tas de questions : où créer la connexion avec la BDD ? Où placer les requêtes ? Comment traiter les valeurs retournées ? Nous allons justement voir tout cela ensemble. Accrochez-vous bien, un TP suivra pour voir si vous avez bien tout compris !

Une entité, un objet

Rappels sur la structure d'une BDD

Commençons ce chapitre par observer la structure d'une table en BDD et, plus précisément, la manière dont sont stockées les données. Comme vous le savez sûrement, une table n'est autre qu'un grand tableau où sont organisées les données :

⇐T⇒	id	nom	forcePerso	degats	niveau	experience
☐ ✎ ✖	16	Vyk12	5	55	4	20
☐ ✎ ✖	18	Tchaper	5	5	1	0
☐ ✎ ✖	19	Neo	5	0	1	20

Exemple d'une table de personnages

Nous voyons ici que notre table contient 3 personnages. Ces données sont stockées sous forme d'entrées. C'est sous cette forme que le gestionnaire de la BDD (MySQL, PostgreSQL, etc.) manipule les données. Si l'on regarde du côté de l'application qui les manipule (ici, notre script PHP), on se rend compte que c'est sous une forme similaire que les données sont récupérées. Cette forme utilisée, vous la connaissez bien : vous utilisiez jusqu'à présent des tableaux pour les manipuler. Exemple :

```html+php
<?php
// On admet que $db est un objet PDO
$request = $db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages');

while ($perso = $request->fetch(PDO::FETCH_ASSOC)) // Chaque entrée sera récupérée et placée dans un array.
{
  echo $perso['nom'], ' a ', $perso['forcePerso'], ' de force, ', $perso['degats'], ' de dégâts, ', $perso['experience'], ' d\'expérience et est au niveau ', $perso['niveau'];
}
```

Ça, si vous avez déjà fait un site internet de A à Z, vous avez du l'écrire pas mal de fois !

> Pour les exemples, et dans la suite du cours, j'utilise l'API PDO pour accéder à la base de données. Je vous conseille de lire le cours de M@teo21 sur PDO pour être à l'aise avec !

Maintenant, puisque nous programmons de manière orientée objet, nous voulons travailler seulement avec des objets (c'est le principe même de la POO, rappelons-le) et non plus avec des tableaux. En d'autres termes, il va falloir transformer le tableau que l'on reçoit en objet.

Travailler avec des objets

Avant de travailler avec des objets, encore faut-il savoir de quelle classe ils sont issus. Dans notre cas, nous voulons des objets représentant des personnages. On aura donc besoin d'une classe `Personnage` !

```php
<?php
class Personnage
{

}
```

Vient maintenant une question embêtante dans votre tête pour construire cette classe : « Je commence par où ? »

Une classe est composée de deux parties (éventuellement trois) :

- une partie déclarant les attributs. Ce sont les *caractéristiques* de l'objet ;

- une partie déclarant les méthodes. Ce sont les *fonctionnalités* de chaque objet ;
- éventuellement, une partie déclarant les constantes de classe. Nous nous en occuperons en temps voulu.

Lorsque l'on veut construire une classe, il va donc falloir *systématiquement* se poser les mêmes questions :

- Quelles seront les caractéristiques de mes objets ?
- Quelles seront les fonctionnalités de mes objets ?

Les réponses à ces questions aboutiront à la réalisation du plan de la classe, le plus difficile. 

Commençons donc à réfléchir sur le plan de notre classe en répondant à la première question : « Quelles seront les caractéristiques de mes objets ? » Pour vous mettre sur la piste, regardez de nouveau le tableau de la BDD contenant les entrées, cela vous donnera peut-être la réponse… Et oui, les attributs de notre classe (les caractéristiques) nous sont offerts sur un plateau ! Ils sont listés en haut du tableau : `id`, `nom`, `forcePerso`, `degats`, `niveau` et `experience`.

Écrivons maintenant notre classe :

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;
}
```

Il faut bien sûr implémenter (écrire) les *getters* et *setters* qui nous permettront d'accéder et de modifier les valeurs de notre objet. Pour rappel, un *getter* est une méthode chargée de renvoyer la valeur d'un attribut, tandis qu'un *setter* est une méthode chargée d'assigner une valeur à un attribut **en vérifiant son intégrité** (si vous assignez la valeur sans aucun contrôle, vous perdez tout l'intérêt qu'apporte le principe d'encapsulation).

Pour construire nos setters, il faut donc nous pencher sur les valeurs possibles de chaque attribut :

- les valeurs possibles de l'identifiant sont tous les nombres entiers strictement positifs ;
- les valeurs possibles pour le nom du personnage sont toutes les chaînes de caractères ;
- les valeurs possibles pour la force du personnage sont tous les nombres entiers allant de 1 à 100 ;
- les valeurs possibles pour les dégâts du personnage sont tous les nombres entiers allant de 0 à 100 ;
- les valeurs possibles pour le niveau du personnage sont tous les nombres entiers allant de 1 à 100 ;
- les valeurs possibles pour l'expérience du personnage sont tous les nombres entiers allant de 1 à 100.

On en déduit donc le code de chaque setter. Voici donc ce que donnerait notre classe avec ses getters et setters :

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;

  // Liste des getters

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

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

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

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

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

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

  // Liste des setters

  public function setId($id)
  {
    // On convertit l'argument en nombre entier.
    // Si c'en était déjà un, rien ne changera.
    // Sinon, la conversion donnera le nombre 0 (à quelques exceptions près, mais rien d'important ici).
    $id = (int) $id;

    // On vérifie ensuite si ce nombre est bien strictement positif.
    if ($id > 0)
    {
      // Si c'est le cas, c'est tout bon, on assigne la valeur à l'attribut correspondant.
      $this->_id = $id;
    }
  }

  public function setNom($nom)
  {
    // On vérifie qu'il s'agit bien d'une chaîne de caractères.
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }

  public function setForcePerso($forcePerso)
  {
    $forcePerso = (int) $forcePerso;

    if ($forcePerso >= 1 && $forcePerso <= 100)
    {
      $this->_forcePerso = $forcePerso;
    }
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setNiveau($niveau)
  {
    $niveau = (int) $niveau;

    if ($niveau >= 1 && $niveau <= 100)
    {
      $this->_niveau = $niveau;
    }
  }

  public function setExperience($experience)
  {
    $experience = (int) $experience;

    if ($experience >= 1 && $experience <= 100)
    {
      $this->_experience = $experience;
    }
  }
}
?>
```

Reprenons notre code de tout à l'heure, celui qui nous permettait de lire la BDD. Modifions-le un peu pour qu'on puisse manipuler des objets et non des tableaux :

```html+php
<?php
// On admet que $db est un objet PDO.
$request = $db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages');

while ($donnees = $request->fetch(PDO::FETCH_ASSOC)) // Chaque entrée sera récupérée et placée dans un array.
{
  // On passe les données (stockées dans un tableau) concernant le personnage au constructeur de la classe.
  // On admet que le constructeur de la classe appelle chaque setter pour assigner les valeurs qu'on lui a données aux attributs correspondants.
  $perso = new Personnage($donnees);

  echo $perso->nom(), ' a ', $perso->forcePerso(), ' de force, ', $perso->degats(), ' de dégâts, ', $perso->experience(), ' d\'expérience et est au niveau ', $perso->niveau();
}
```

Et quelles seront les méthodes de nos classes ?

Les méthodes concernent des fonctionnalités que possède l'objet, des actions qu'il peut effectuer. Voyez-vous des fonctionnalités intéressantes à implémenter ? Pour les opérations basiques que l'on effectue, il n'y en a pas besoin. En effet, nous voulons juste créer des objets et assigner des valeurs aux attributs, donc hormis les getters et setters, aucune autre méthode n'est nécessaire !

D'accord, nous avons des objets maintenant. Mais à quoi cela sert-il, concrètement ?

Il est sûr que dans cet exemple précis, cela ne sert à rien. Mais vous verrez plus tard qu'il est beaucoup plus intuitif de travailler avec des objets et par conséquent beaucoup plus pratique, notamment sur de grosses applications où de nombreux objets circulent un peu dans tous les sens.

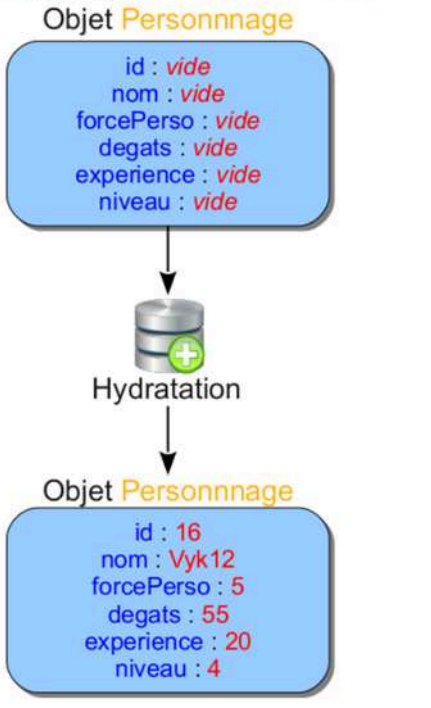
L'hydratation

La théorie de l'hydratation

L'hydratation est un point essentiel dans le domaine de la POO, notamment lorsqu'on utilise des objets représentant des données stockées. Cette notion peut vite devenir compliquée et créer des interrogations pour un développeur débutant si elle est abordée de manière approximative, alors que des explications claires prouvent qu'il n'y a rien de compliqué là-dedans.

Quand on vous parle d'hydratation, c'est qu'on parle d'« objet à hydrater ». Hydrater un objet, c'est tout simplement lui apporter ce dont il a besoin pour fonctionner. En d'autres termes plus précis, hydrater un objet revient à lui fournir des données correspondant à ses attributs pour qu'il assigne les valeurs souhaitées à ces derniers. L'objet aura ainsi des attributs valides et sera en lui-même valide. On dit que l'objet a ainsi été hydraté.

Schématiquement, une hydratation se produit comme ceci :



Hydratation d'un objet

Au début, nous avons un objet `Personnage` dont les attributs sont vides. Comme le schéma le représente, l'hydratation consiste à assigner des valeurs aux attributs. Ainsi l'objet est fonctionnel car il contient des attributs valides : nous avons donc bien hydraté l'objet.

L'hydratation en pratique

Comme on l'a vu, hydrater un objet revient à assigner des valeurs à ses attributs. Qu'avons-nous besoin de faire pour réaliser une telle chose ? Il faut ajouter à l'objet l'action de s'hydrater. Et qui dit action dit méthode !

Nous allons donc écrire notre fonction `hydrate()` :

```php
<?php
class Personnage
{
  private $_id;
  private $_nom;
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;

  // Un tableau de données doit être passé à la fonction (d'où le préfixe « array »).
  public function hydrate(array $donnees)
  {

  }

  public function id() { return $this->_id; }
  public function nom() { return $this->_nom; }
  public function forcePerso() { return $this->_forcePerso; }
  public function degats() { return $this->_degats; }
  public function niveau() { return $this->_niveau; }
  public function experience() { return $this->_experience; }

  public function setId($id)
  {
    // L'identifiant du personnage sera, quoi qu'il arrive, un nombre entier.
    $this->_id = (int) $id;
  }

  public function setNom($nom)
  {
    // On vérifie qu'il s'agit bien d'une chaîne de caractères.
    // Dont la longueur est inférieure à 30 caractères.
    if (is_string($nom) && strlen($nom) <= 30)
    {
      $this->_nom = $nom;
    }
  }

  public function setForcePerso($forcePerso)
  {
    $forcePerso = (int) $forcePerso;

    // On vérifie que la force passée est comprise entre 0 et 100.
    if ($forcePerso >= 0 && $forcePerso <= 100)
    {
      $this->_forcePerso = $forcePerso;
    }
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    // On vérifie que les dégâts passés sont compris entre 0 et 100.
    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setNiveau($niveau)
  {
    $niveau = (int) $niveau;

    // On vérifie que le niveau n'est pas négatif.
    if ($niveau >= 0)
    {
      $this->_niveau = $niveau;
    }
  }

  public function setExperience($exp)
  {
    $exp = (int) $exp;

    // On vérifie que l'expérience est comprise entre 0 et 100.
    if ($exp >= 0 && $exp <= 100)
    {
      $this->_experience = $exp;
    }
  }
}
```

Vient maintenant l'aspect le plus important : que doit-on mettre dans cette méthode ? Souvenez-vous de sa fonction : elle doit hydrater l'objet, c'est-à-dire « assigner les valeurs passées en paramètres aux attributs correspondant ». Cependant, je vous le dis avant que vous fassiez fausse route : il ne faut pas assigner ces valeurs directement, comme ceci :

```php
<?php
// …
public function hydrate(array $donnees)
{
  if (isset($donnees['id']))
  {
    $this->_id = $donnees['id'];
  }

  if (isset($donnees['nom']))
  {
    $this->_nom = $donnees['nom'];
  }

  // …
}
// …
```

La principale raison pour laquelle c'est une mauvaise façon de procéder, est qu'en agissant ainsi vous violez le principe d'encapsulation. En effet, de cette manière, vous ne contrôlez pas l'intégrité des valeurs. Qui vous garantit que le tableau contiendra un nom valide ? Rien du tout! Comment contrôler alors l'intégrité de ces valeurs ? Regardez un peu votre classe, la réponse est sous vos yeux… C'est le rôle des setters de faire ces vérifications ! Il faudra donc les appeler au lieu d'assigner les valeurs directement :

```php
<?php
// …
public function hydrate(array $donnees)
{
  if (isset($donnees['id']))
  {
    $this->setId($donnees['id']);
  }

  if (isset($donnees['nom']))
  {
    $this->setNom($donnees['nom']);
  }

  // …
}
// …
```

Théoriquement, nous avons terminé le travail. Cependant, cela n'est pas très flexible : si vous ajoutez un attribut (et donc son setter correspondant), il faudra modifier votre méthode `hydrate()` pour ajouter la possibilité d'assigner cette valeur. De plus, avec les 6 attributs actuellement créés, il est long de vérifier si la clé existe puis d'appeler la méthode correspondante.

Nous allons donc procéder plus rapidement : nous allons créer une boucle qui va parcourir le tableau passé en paramètre. Si le setter correspondant à la clé existe, alors on appelle ce setter en lui passant la valeur en paramètre. En langage naturel, l'algorithme peut s'écrire de cette façon :

```
PARCOURS du tableau $donnees (avec pour clé $cle et pour valeur $valeur)
  On assigne à $setter la valeur « 'set'.$cle », en mettant la première lettre de $cle en majuscu
  SI la méthode $setter de notre classe existe ALORS
    On invoque $setter($valeur)
  FIN SI
FIN PARCOURS
```

Pas de panique, je vais vous expliquer chaque ligne. Dans votre tête, prenez un exemple de valeur pour `$donnees` :

```php
<?php
$donnees = [
  'id' => 16,
  'nom' => 'Vyk12',
  'forcePerso' => 5,
  'degats' => 55,
  'niveau' => 4,
  'experience' => 20
];
```

Vous voyez bien que chaque clé correspond à un attribut de notre objet, à qui on assigne une valeur précise. Dans notre méthode `hydrate()`, lorsqu'on parcourt notre tableau, on a successivement la clé `id`, puis `nom`, puis `forcePerso`, etc., avec leur valeur correspondante.

En récupérant le nom de l'attribut, il est facile de déterminer le setter correspondant. En effet, chaque setter a pour nom `setNomDeLAttribut`.

Il est important de préciser que la première lettre du nom de l'attribut doit être en majuscule. Par exemple, le setter correspondant à `nom` est `setNom`.

Pour l'instant notre méthode ressemble à ceci :

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    $method = 'set'.ucfirst($key);
    // …
  }
}
// …
```

Il faut maintenant vérifier que la méthode existe. Pour cela, nous allons utiliser la fonction method_exists(). Elle prend en premier paramètre le nom de la classe ou une instance de cette classe, et en deuxième paramètre le nom de la méthode qui nous intéresse. La méthode renvoie `true` si la méthode existe, sinon `false`.

Je vous laisse ajouter cette condition qui permet de voir si le setter correspondant existe.

Voici la correction :

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    $method = 'set'.ucfirst($key);

    if (method_exists($this, $method))
    {
      // …
    }
  }
}
// …
```

Et maintenant, il ne reste plus qu'à appeler le setter à l'intérieur de la condition ! Pour cela, je vais vous apprendre une petite astuce. Il est possible d'appeler une méthode dynamiquement, c'est-à-dire appeler une méthode dont le nom n'est pas connu à l'avance (en d'autres termes, le nom est connu seulement pendant l'exécution et est donc stocké dans une variable). Pour ce faire, rien de plus simple ! Regardez ce code :

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

$a = new A;
$method = 'hello';

$a->$method(); // Affiche : « Hello world ! »
```

Il est bien entendu possible de passer des arguments à la méthode.

Vous êtes maintenant capables de créer la dernière instruction dans notre méthode ! Pour rappel, celle-ci doit invoquer le setter dont le nom est contenu dans la variable `$method`.

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    $method = 'set'.ucfirst($key);

    if (method_exists($this, $method))
    {
      // …
    }
  }
}
// …
```

Et maintenant, il ne reste plus qu'à appeler le setter à l'intérieur de la condition ! Pour cela, je vais vous apprendre une petite astuce. Il est possible d'appeler une méthode dynamiquement, c'est-à-dire appeler une méthode dont le nom n'est pas connu à l'avance (en d'autres termes, le nom est connu seulement pendant l'exécution et est donc stocké dans une variable). Pour ce faire, rien de plus simple ! Regardez ce code :

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

$a = new A;
$method = 'hello';

$a->$method(); // Affiche : « Hello world ! »
```



Il est bien entendu possible de passer des arguments à la méthode.

Vous êtes maintenant capables de créer la dernière instruction dans notre méthode ! Pour rappel, celle-ci doit invoquer le setter dont le nom est contenu dans la variable `$method`.

```php
<?php
// …
public function hydrate(array $donnees)
{
  foreach ($donnees as $key => $value)
  {
    // On récupère le nom du setter correspondant à l'attribut.
    $method = 'set'.ucfirst($key);

    // Si le setter correspondant existe.
    if (method_exists($this, $method))
    {
      // On appelle le setter.
      $this->$method($value);
    }
  }
}
// …
```

Cette fonction est très importante, vous la retrouverez dans de nombreux codes (parfois sous des formes différentes) provenant de plusieurs développeurs. Gardez-là donc dans un coin de votre tête.



Il est courant d'implémenter un constructeur à ces classes demandant un tableau de valeurs pour qu'il appelle ensuite la fonction d'hydratation afin que l'objet soit hydraté dès sa création, comme on l'a vu au début de ce chapitre.

Gérer sa BDD correctement

On vient de voir jusqu'à présent comment gérer les données que les requêtes nous renvoient, mais où placer ces requêtes ? Notre but est de programmer orienté objet, donc nous voulons le moins de code possible en-dehors des classes pour mieux l'organiser. Beaucoup de débutants sont tentés de placer les requêtes dans des méthodes de la classe représentant une entité de la BDD. Par exemple, dans le cas du personnage, je parie que la plupart d'entre vous seraient tentés d'écrire les méthodes `add()` et `update()` dans la classe `Personnage`, ayant respectivement pour rôle d'exécuter une requête pour ajouter et modifier un personnage en BDD.

Arrêtez-vous là tout de suite ! Je vais vous expliquer pourquoi vous faites fausse route, puis vous montrerai la bonne marche à suivre.

Une classe, un rôle

En POO, il y a une phrase très importante qu'il faut que vous ayez constamment en tête : « Une classe, un rôle. » Maintenant, répondez clairement à cette question : « Quel est le rôle d'une classe comme `Personnage` » ?

Un objet instanciant une classe comme `Personnage` a pour rôle de *représenter* une ligne présente en BDD. Le verbe « représenter » est ici très important. En effet, « représenter » est très différent de « gérer ». Une ligne de la BDD ne peut pas s'auto-gérer ! C'est comme si vous demandiez à un ouvrier ayant construit un produit de le commercialiser : l'ouvrier est tout à fait capable de le construire, c'est son *rôle*, mais il ne s'occupe pas du tout de sa gestion, il en est incapable. Il faut donc qu'une deuxième personne intervienne, un commercial, qui va s'occuper de vendre ce produit.

Pour revenir à nos objets d'origine, nous aurons donc besoin de quelque chose qui va s'occuper de les gérer. Ce quelque chose, vous l'aurez peut-être deviné, n'est autre qu'un objet. Un objet gérant des entités issues d'une BDD est généralement appelé un « manager ».

Comme un manager ne fonctionne pas sans support de stockage (dans notre cas, une BDD), on va prendre un exemple concret en créant un gestionnaire pour nos personnages, qui va donc se charger d'en ajouter, d'en modifier, d'en supprimer et d'en récupérer. Puisque notre classe est un gestionnaire de personnages, je vous propose de la nommer `PersonnagesManager`. Cependant, rappelez-vous les questions que l'on doit se poser pour établir le plan de notre classe :

- Quelles seront les caractéristiques de mes objets ?
- Quelles seront les fonctionnalités de mes objets ?

Les caractéristiques d'un manager

Partie délicate, car cela est moins intuitif que de trouver les caractéristiques d'un personnage. Pour trouver la réponse, vous devrez passer par une autre question (ce serait trop simple de toujours répondre à la même question !) : « De quoi a besoin un manager pour fonctionner ? »

Même si la réponse ne vous vient pas spontanément à l'esprit, vous la connaissez : elle a besoin d'une connexion à la BDD pour pouvoir exécuter des requêtes. En utilisant PDO, vous devriez savoir que la connexion à la BDD est représentée par un objet, un *objet d'accès à la BDD* (ou DAO pour *Database Access Object*). Vous l'avez peut-être compris : notre manager aura besoin de cet objet pour fonctionner, donc notre classe aura un attribut stockant cet objet.

Notre classe a-t-elle besoin d'autre chose pour fonctionner ?

Non, c'est tout. Vous pouvez donc commencer à écrire votre classe :

```php
<?php
class PersonnagesManager
{
  private $_db;
}
```

Les fonctionnalités d'un manager

Pour pouvoir gérer au mieux des entités présentes en BDD (ici, nos personnages), il va falloir quelques fonctionnalités de base. Quelles sont-elles ?

Un manager doit pouvoir :

- enregistrer une nouvelle entité ;
- modifier une entité ;
- supprimer une entité ;
- sélectionner une entité.

C'est le fameux CRUD (*Create, Read, Update, Delete*). N'oublions pas aussi d'ajouter un setter pour notre manager afin de pouvoir modifier l'attribut `$_db` (pas besoin d'accesseur). La création d'un constructeur sera aussi indispensable si nous voulons assigner à cet attribut un objet PDO dès l'instanciation du manager.

Nous allons donc écrire notre premier manager en écrivant les méthodes correspondant à ces fonctionnalités. Écrivez dans un premier temps les méthodes en ne les remplissant que de commentaires décrivant les instructions qui y seront écrites. Cela vous permettra d'organiser clairement le code dans votre tête.

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO.

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

  public function add(Personnage $perso)
  {
    // Préparation de la requête d'insertion.
    // Assignation des valeurs pour le nom, la force, les dégâts, l'expérience et le niveau du personnage.
    // Exécution de la requête.
  }

  public function delete(Personnage $perso)
  {
    // Exécute une requête de type DELETE.
  }

  public function get($id)
  {
    // Exécute une requête de type SELECT avec une clause WHERE, et retourne un objet Personnage.
  }

  public function getList()
  {
    // Retourne la liste de tous les personnages.
  }

  public function update(Personnage $perso)
  {
    // Prépare une requête de type UPDATE.
    // Assignation des valeurs à la requête.
    // Exécution de la requête.
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Une fois cette étape accomplie, vous pouvez remplacer les commentaires par les instructions correspondantes. C'est la partie la plus facile car vous l'avez déjà fait de nombreuses fois en procédural : ce ne sont que des requêtes à taper et des valeurs à retourner.

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

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

  public function add(Personnage $perso)
  {
    $q = $this->_db->prepare('INSERT INTO personnages SET nom = :nom, forcePerso = :forcePerso, degats = :degats, niveau = :niveau, experience = :experience');

    $q->bindValue(':nom', $perso->nom());
    $q->bindValue(':forcePerso', $perso->forcePerso(), PDO::PARAM_INT);
    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':niveau', $perso->niveau(), PDO::PARAM_INT);
    $q->bindValue(':experience', $perso->experience(), PDO::PARAM_INT);

    $q->execute();
  }

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

  public function get($id)
  {
    $id = (int) $id;

    $q = $this->_db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages WHERE id = '.$id);
    $donnees = $q->fetch(PDO::FETCH_ASSOC);

    return new Personnage($donnees);
  }

  public function getList()
  {
    $persos = [];

    $q = $this->_db->query('SELECT id, nom, forcePerso, degats, niveau, experience FROM personnages ORDER BY nom');

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      $persos[] = new Personnage($donnees);
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->_db->prepare('UPDATE personnages SET forcePerso = :forcePerso, degats = :degats, niveau = :niveau, experience = :experience WHERE id = :id');

    $q->bindValue(':forcePerso', $perso->forcePerso(), PDO::PARAM_INT);
    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':niveau', $perso->niveau(), PDO::PARAM_INT);
    $q->bindValue(':experience', $perso->experience(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Essayons tout ça !

Faisons un petit test pour s'assurer que tout cela fonctionne.



Assurez-vous de bien avoir créé une table `personnages` dans votre BDD.

Commençons par créer un objet `Personnage` :

```php
<?php
$perso = new Personnage([
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
]);
```

Maintenant, comment l'ajouter en BDD ? Il faudra d'abord créer une instance de notre manager, *en lui passant une instance de PDO.*

```php
<?php
$perso = new Personnage([
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
]);

$db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');
$manager = new PersonnagesManager($db);
```

Nous n'avons plus qu'une instruction à entrer : celle ordonnant l'ajout du personnage en BDD. Et c'est tout !

```php
<?php
$perso = new Personnage([
  'nom' => 'Victor',
  'forcePerso' => 5,
  'degats' => 0,
  'niveau' => 1,
  'experience' => 0
]);

$db = new PDO('mysql:host=localhost;dbname=tests', 'root', '');
$manager = new PersonnagesManager($db);

$manager->add($perso);
```

Il est maintenant temps de mettre en pratique ce que vous venez d'apprendre. Cependant, avant de continuer, assurez-vous bien de tout avoir compris, sinon vous serez perdus car vous ne comprendrez pas le code du TP !

En résumé

- Du côté du script PHP, chaque enregistrement de la base de données est représenté par un objet possédant une liste d'attributs identique à la liste des colonnes de la table.
- Il doit être possible d'*hydrater* chacun des objets grâce à une méthode `hydrate` qui a pour rôle d'assigner les valeurs passées en paramètre aux attributs correspondant.
- La communication avec la BDD se fait par le biais d'un objet différent de l'objet représentant l'enregistrement (une classe = un rôle). Un tel objet est un *manager*.
- Un *manager* peut stocker les objets en BDD, mais peut tout-à-fait les stocker sur un autre support (fichier XML, fichier texte, etc.).

☐ J'ai terminé ce chapitre et je passe au suivant

TP : MINI-JEU DE COMBAT

Voici un petit TP qui mettra en pratique ce que l'on vient de voir. Celui-ci est étroitement lié avec le précédent chapitre. Ainsi, je vous conseille d'avoir bien compris ce dernier avant d'aborder ce TP, sinon vous n'arriverez sans doute pas au résultat tout seul.

En tout, vous aurez 3 TP de la sorte dont le niveau de difficulté sera croissant. Puisque celui-ci est votre premier, tout sera expliqué dans les moindres détails.

Ce qu'on va faire

Cahier des charges

Ce que nous allons réaliser est très simple. Nous allons 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. Le personnage frappé se verra infliger un certain degré de dégâts.

Un personnage est défini selon 2 caractéristiques :

- Son nom (unique).
- Ses dégâts.

Les dégâts d'un personnage sont compris entre 0 et 100. Au début, il a bien entendu 0 de dégât. Chaque coup qui lui sera porté lui fera prendre 5 points de dégâts. Une fois arrivé à 100 points de dégâts, le personnage est mort (on le supprimera alors de la BDD).

Notions utilisées

Voici une petite liste vous indiquant les points techniques que l'on va mettre en pratique avec ce TP :

- Les attributs et méthodes ;
- l'*instanciation* de la classe ;
- les constantes de classe ;
- et surtout, tout ce qui touche à la **manipulation de données stockées.**

Cela dit, et c'est je pense ce qui sera le plus difficile, ce qui sera mis en valeur ici sera la **conception** d'un mini-projet, c'est-à-dire que l'on travaillera surtout ici votre capacité à programmer orienté objet. Cependant, ne prenez pas trop peur : nous avons effectué une importante partie théorique lors du précédent chapitre, et rien de nouveau n'apparaîtra.

Vous avez donc maintenant une idée de ce qui vous attend. Cependant, ceci étant votre premier TP concernant la POO, il est fort probable que vous ne sachiez pas par où commencer, et c'est normal ! Nous allons donc réaliser le TP ensemble : je vais vous expliquer comment **penser** un mini-projet et vous poser les bonnes questions.

Pré-conception

Avant de nous attaquer au cœur du script, nous allons réfléchir à son organisation. De quoi aura-t-on besoin ? Puisque nous travaillerons avec des personnages, nous aurons besoin de les stocker pour qu'ils puissent durer dans le temps. L'utilisation d'une base de données sera donc indispensable.

Le script étant simple, nous n'aurons qu'une table **personnages** qui aura différents champs. Pour les définir, réfléchissez à ce qui caractérise un personnage. Ainsi nous connaissons déjà 2 champs de cette table que nous avons définis au début : **nom** et **dégâts** . Et bien sûr, n'oublions pas le plus important : l'identifiant du personnage ! Chaque personnage doit posséder un identifiant unique qui permet ainsi de le rechercher plus rapidement (au niveau performances) qu'avec son nom.

Vous pouvez donc ainsi créer votre table tous seuls *via* PhpMyAdmin. Si vous n'êtes pas sûrs de vous, je vous laisse le code SQL créant cette table :

```mysql
CREATE TABLE IF NOT EXISTS `personnages` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `nom` varchar(50) COLLATE utf8_general_ci NOT NULL,
  `degats` tinyint(3) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  UNIQUE KEY `nom` (`nom`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_general_ci;
```

Voir le résultat que vous obtiendrez

Le résultat présenté ici contient les améliorations proposées en fin de chapitre.

Première étape : le personnage

Nous allons commencer le TP. Pour rappel, nous allons réaliser un petit jeu mettant en scène des personnages qui peuvent combattre. Qui dit personnages dit **objets** `Personnage` (je pense que ça, vous l'avez deviné, puisque nous avons travaillé dessus durant les premiers chapitres).

Arrive maintenant un moment délicat dans votre tête : **"Par où je commence ?"**

Souvenez-vous de la première partie du précédent chapitre. Pour construire une classe, vous devez répondre à deux questions qui vont vous permettre d'établir le plan de votre classe :

- Quelles seront les caractéristiques de mes objets ?

- Quelles seront les fonctionnalités de mes objets ?

Voici comment procéder. Nous allons dans un premier temps dresser la liste des caractéristiques du personnage pour ensuite se pencher sur ses fonctionnalités.



Rappel : le traitement des données (c'est-à-dire l'exécution de requêtes permettant d'aller effectuer des opérations en BDD) se fera **dans une autre classe**. Ne vous en préoccupez donc pas maintenant !

Les caractéristiques du personnage

Essayez de vous souvenir quelles sont les caractéristiques d'un personnage (ou plus globalement celles d'une entité représentant un enregistrement présent en BDD). Nous l'avons vu dans la première partie du précédent chapitre.



Les attributs d'un objet représentant un enregistrement présent en BDD correspondent aux noms des champs de cet enregistrement.

Si vous n'avez pas retenu cette information, notez-là quelque part, vous en aurez besoin un très grand nombre de fois.

Maintenant que nous avons déterminé les caractéristiques d'un objet `Personnage`, nous savons quels attributs placer dans notre classe. Voici une première écriture de notre classe `Personnage` :

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;
}
```

Jusque là tout va bien. Tournons-nous maintenant vers les fonctionnalités que doit posséder un personnage.

Les fonctionnalités d'un personnage

Comme nous l'avons dit tout-à-l'heure, pour obtenir les méthodes d'un objet, il faut se demander quelles seront les fonctionnalités de ces entités. Ici, **que pourra faire un personnage** ? Relisez les consignes du début de chapitre et répondez clairement à la question.

Un personnage doit pouvoir :

- **Frapper** un autre personnage ;
- **recevoir** des dégâts.

À chaque fonctionnalité correspond une méthode. Écrivez ces méthodes dans la classe en mettant en commentaire ce qu'elles doivent faire.



Ne vous étonnez pas si vous avez un résultat bien différent du mien. Le contenu des méthodes importe peu car il est issu de mon imagination et il est normal que vous n'ayez pas pensé le script comme moi. Gardez votre version, n'essayez pas de copier/coller mon code.

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
    // Si c'est le cas, on stoppe tout en renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage a bien été frappé.
  }
}
```

Tout ceci n'est que du français, nous sommes encore à l'étape de **réflexion**. Si vous sentez que ça va trop vite, relisez le début du TP et suivez bien les étapes. C'est très important car c'est en général cette étape qui pose problème : la **réflexion** ! Beaucoup de débutants foncent tête baissée sans prendre le temps de concevoir correctement les méthodes au début et se plantent royalement. Prenez donc bien votre temps.

Normalement, des petites choses doivent vous chiffonner.

Pour commencer, la première chose qui doit vous interpeller se situe dans la méthode `frapper()`, lorsqu'il faut vérifier qu'on ne se frappe pas soi-même. Pour cela, il suffit de comparer l'identifiant du personnage qui attaque avec l'identifiant du personnage ciblé. En effet, si l'identifiant est le même, alors il s'agira d'un seul et même personnage.

Ensuite, à certains moments dans le code, il est dit que la méthode « renverra une valeur signifiant que le personnage a bien été frappé » par exemple. Qu'est-ce que cela peut bien signifier ? Ce genre de commentaire laissera place à des **constantes de classe** (si vous l'aviez deviné, bien joué !). Si vous regardez bien le code, vous verrez 3 commentaires de la sorte :

- Méthode `frapper()` : *« [...] renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque »* ⟶ la méthode renverra la valeur de la constante `CEST_MOI` ;
- Méthode `recevoirDegats()` : *« la méthode renverra une valeur signifiant que le personnage a été tué »* ⟶ la méthode renverra la valeur de la constante `PERSONNAGE_TUE` ;
- Méthode `recevoirDegats()` : *« elle renverra une valeur signifiant que le personnage a bien été frappé »* ⟶ la méthode renverra la valeur de la constante `PERSONNAGE_FRAPPE`.

Vous pouvez ajouter ces constantes à votre classe.

html+php
```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;

  const CEST_MOI = 1;
  const PERSONNAGE_TUE = 2;
  const PERSONNAGE_FRAPPE = 3;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
    // Si c'est le cas, on stoppe tout en renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage a bien été frappé.
  }
}
```

Les getters et setters

Actuellement, les attributs de nos objets sont inaccessibles. Il faut créer des *getters* pour pouvoir les lire, et des *setters* pour pouvoir modifier leurs valeurs. Nous allons aller un peu plus rapidement que les précédentes méthodes en écrivant directement le contenu de celles-ci car une ou deux instructions suffisent en général.

N'oubliez pas de vérifier l'intégrité des données dans les *setters* sinon ils perdent tout leur intérêt !

```php
<?php
class Personnage
{
  private $_id,
          $_degats,
          $_nom;

  const CEST_MOI = 1;
  const PERSONNAGE_TUE = 2;
  const PERSONNAGE_FRAPPE = 3;

  public function frapper(Personnage $perso)
  {
    // Avant tout : vérifier qu'on ne se frappe pas soi-même.
    // Si c'est le cas, on stoppe tout en renvoyant une valeur signifiant que le personnage ciblé est le personnage qui attaque.

    // On indique au personnage frappé qu'il doit recevoir des dégâts.
  }

  public function recevoirDegats()
  {
    // On augmente de 5 les dégâts.

    // Si on a 100 de dégâts ou plus, la méthode renverra une valeur signifiant que le personnage a été tué.

    // Sinon, elle renverra une valeur signifiant que le personnage a bien été frappé.
  }

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

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

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

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setId($id)
  {
    $id = (int) $id;

    if ($id > 0)
    {
      $this->_id = $id;
    }
  }

  public function setNom($nom)
  {
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }
}
```

Hydrater ses objets

Deuxième point essentiel que nous allons ré-exploiter dans ce TP : l'hydratation.

Je n'ai pas d'explication supplémentaire à ajouter, tout est dit dans le précédent chapitre. La méthode créée dans ce dernier est d'ailleurs réutilisable ici. Au lieu de regarder la correction en vous replongeant dans le chapitre précédent, essayez plutôt de réécrire la méthode. Pour rappel, celle-ci doit permettre d'assigner aux attributs de l'objet les valeurs correspondantes, passées en paramètre dans un tableau. Si vous avez essayé de réécrire la méthode, voici le résultat que vous auriez du obtenir :

```php
<?php
class Personnage
{
  // ...

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  // ...
}
```

Il ne manque plus qu'à implémenter le constructeur pour qu'on puisse directement hydrater notre objet lors de l'instanciation de la classe. Pour cela, ajoutez un paramètre : `$donnees`. Appelez ensuite directement la méthode `hydrate()`.

```php
<?php
class Personnage
{
  // ...

  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
  }

  // ...
}
```

Codons le tout !

La partie réflexion est terminé, il est maintenant temps d'écrire notre classe `Personnage` ! Voici la correction que je vous propose :

```php
<?php
class Personnage
{
  private $_degats,
          $_id,
          $_nom;

  const CEST_MOI = 1; // Constante renvoyée par la méthode `frapper` si on se frappe soi-même.
  const PERSONNAGE_TUE = 2; // Constante renvoyée par la méthode `frapper` si on a tué le personnage en le frappant.
  const PERSONNAGE_FRAPPE = 3; // Constante renvoyée par la méthode `frapper` si on a bien frappé le personnage.


  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
  }

  public function frapper(Personnage $perso)
  {
    if ($perso->id() == $this->_id)
    {
      return self::CEST_MOI;
    }

    // On indique au personnage qu'il doit recevoir des dégâts.
    // Puis on retourne la valeur renvoyée par la méthode : self::PERSONNAGE_TUE ou self::PERSONNAGE_FRAPPE
    return $perso->recevoirDegats();
  }

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  public function recevoirDegats()
  {
    $this->_degats += 5;

    // Si on a 100 de dégâts ou plus, on dit que le personnage a été tué.
    if ($this->_degats >= 100)
    {
      return self::PERSONNAGE_TUE;
    }

    // Sinon, on se contente de dire que le personnage a bien été frappé.
    return self::PERSONNAGE_FRAPPE;
  }


```

```php
  // GETTERS //


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

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

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

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }

  public function setId($id)
  {
    $id = (int) $id;

    if ($id > 0)
    {
      $this->_id = $id;
    }
  }

  public function setNom($nom)
  {
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }
}
```

Prenez le temps de bien comprendre ce code et de lire les commentaires. Il est important que vous cerniez son fonctionnement pour savoir ce que vous faites. Cependant, si vous ne comprenez pas toutes les instructions placées dans les méthodes, ne vous affolez pas : si vous avez compris globalement le rôle de chacune d'elles, vous n'aurez pas de handicap pour suivre la prochaine sous-partie.

Seconde étape : stockage en base de données

Attaquons-nous maintenant à la deuxième grosse partie de ce TP, celle consistant à pouvoir stocker nos personnages dans une base de données. Grande question maintenant : **comment faire ?**

Nous avons répondu à cette question dans la troisième partie du précédent chapitre. Au cas où certains seraient toujours tentés de placer les requêtes qui iront chercher les personnages en BDD dans la classe `Personnage`, je vous arrête tout de suite et vous fais un bref rappel avec cette phrase que vous avez déjà rencontrée : **une classe, un rôle.**

J'espère que vous l'avez retenue cette fois-ci !

Vous souvenez-vous de ce que cela signifie ? Quel est le rôle de notre classe `Personnage` ? Où placer nos requêtes ?

La classe `Personnage` a pour rôle de **représenter** un personnage présent en BDD. Elle n'a en aucun cas pour rôle de les **gérer.** Cette gestion sera le rôle d'une autre classe, communément appelée **manager.** Dans notre cas, notre gestionnaire de personnage sera tout simplement nommée `PersonnagesManager`.

Comment va-t-on faire pour construire ces classes ? Quelles questions va-t-on se poser ?

- Quelles seront les caractéristiques d'un manager ?
- Quelles seront les fonctionnalités d'un manager ?

Les caractéristiques d'un manager

Encore une fois, ce point a été abordé dans la troisième partie du précédent chapitre. Allez y faire un tour si vous avez un trou de mémoire ! Voici le code de la classe contenant sa (grande) liste d'attributs.

```php
<?php
class PersonnagesManager
{
  private $_db;
}
```

Les fonctionnalités d'un manager

Dans la troisième partie du précédent chapitre, nous avons vu quelques fonctionnalités de base. Notre manager pouvait :

- Enregistrer un nouveau personnage ;
- modifier un personnage ;

- supprimer un personnage ;
- sélectionner un personnage.

Cependant, ici, nous pouvons ajouter quelques fonctionnalités qui pourront nous être utiles :

- Compter le nombre de personnages ;
- récupérer une liste de plusieurs personnages ;
- savoir si un personnage existe.

Cela nous fait ainsi 7 méthodes à implémenter !



Rappels du précédent chapitre : n'oubliez pas d'ajouter un *setter* pour notre manager afin de pouvoir modifier l'attribut `$_db`. La création d'un constructeur sera aussi indispensable si nous voulons assigner à cet attribut un objet PDO dès l'instanciation du manager.

Comme d'habitude, écrivez le nom des méthodes en ajoutant des commentaires sur ce que doit faire la méthode.

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

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

  public function add(Personnage $perso)
  {
    // Préparation de la requête d'insertion.
    // Assignation des valeurs pour le nom du personnage.
    // Exécution de la requête.

    // Hydratation du personnage passé en paramètre avec assignation de son identifiant et des dégâts initiaux (= 0).
  }

  public function count()
  {
    // Exécute une requête COUNT() et retourne le nombre de résultats retourné.
  }

  public function delete(Personnage $perso)
  {
    // Exécute une requête de type DELETE.
  }

  public function exists($info)
  {
    // Si le paramètre est un entier, c'est qu'on a fourni un identifiant.
      // On exécute alors une requête COUNT() avec une clause WHERE, et on retourne un boolean.

    // Sinon c'est qu'on a passé un nom.
    // Exécution d'une requête COUNT() avec une clause WHERE, et retourne un boolean.
  }

  public function get($info)
  {
    // Si le paramètre est un entier, on veut récupérer le personnage avec son identifiant.
      // Exécute une requête de type SELECT avec une clause WHERE, et retourne un objet Personnage.

    // Sinon, on veut récupérer le personnage avec son nom.
    // Exécute une requête de type SELECT avec une clause WHERE, et retourne un objet Personnage.
  }

  public function getList($nom)
  {
    // Retourne la liste des personnages dont le nom n'est pas $nom.
    // Le résultat sera un tableau d'instances de Personnage.
  }

  public function update(Personnage $perso)
  {
    // Prépare une requête de type UPDATE.
    // Assignation des valeurs à la requête.
    // Exécution de la requête.
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Codons le tout !

Normalement, l'écriture des méthodes devrait être plus facile que dans la précédente partie. En effet, ici, il n'y a que des requêtes à écrire : si vous savez utiliser PDO, vous ne devriez pas avoir de mal. 

```php
<?php
class PersonnagesManager
{
  private $_db; // Instance de PDO

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

  public function add(Personnage $perso)
  {
    $q = $this->_db->prepare('INSERT INTO personnages SET nom = :nom');
    $q->bindValue(':nom', $perso->nom());
    $q->execute();

    $perso->hydrate([
      'id' => $this->_db->lastInsertId(),
      'degats' => 0,
    ]);
  }

  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 existe.
    {
      return (bool) $this->_db->query('SELECT COUNT(*) FROM personnages WHERE id = '.$info)->fetchColumn();
    }

    // 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([':nom' => $info]);

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

  public function get($info)
  {
    if (is_int($info))
    {
      $q = $this->_db->query('SELECT id, nom, degats FROM personnages WHERE id = '.$info);
      $donnees = $q->fetch(PDO::FETCH_ASSOC);

      return new Personnage($donnees);
    }
    else
    {
      $q = $this->_db->prepare('SELECT id, nom, degats FROM personnages WHERE nom = :nom');
      $q->execute([':nom' => $info]);

      return new Personnage($q->fetch(PDO::FETCH_ASSOC));
    }
  }

  public function getList($nom)
  {
    $persos = [];

    $q = $this->_db->prepare('SELECT id, nom, degats FROM personnages WHERE nom <> :nom ORDER BY nom');
    $q->execute([':nom' => $nom]);

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      $persos[] = new Personnage($donnees);
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->_db->prepare('UPDATE personnages SET degats = :degats WHERE id = :id');

    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }

  public function setDb(PDO $db)
  {
    $this->_db = $db;
  }
}
```

Troisième étape : utilisation des classes

J'ai le plaisir de vous annoncer que vous avez fait le plus gros du travail ! Maintenant, nous allons juste utiliser nos classes en les instanciant et en invoquant les méthodes souhaitées sur nos objets. Le plus difficile ici est de se mettre d'accord sur le déroulement du jeu.

Celui-ci étant simple, nous n'aurons besoin que d'un seul fichier. Commençons par le début : que doit afficher notre mini-jeu lorsqu'on ouvre la page pour la première fois ? Il doit afficher un petit formulaire nous demandant le nom du personnage qu'on veut créer ou utiliser.

php
```php
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
  </body>
</html>
```

Vient ensuite la partie traitement. Deux cas peuvent se présenter :

- Le joueur a cliqué sur **Créer ce personnage**. Le script devra créer un objet `Personnage` en passant au constructeur un tableau contenant une entrée (le nom du personnage). Il faudra ensuite s'assurer que le personnage ait un nom valide et qu'il n'existe pas déjà. Après ces vérifications, l'enregistrement en BDD pourra se faire.

- Le joueur a cliqué sur **Utiliser ce personnage**. Le script devra vérifier si le personnage existe bien en BDD. Si c'est le cas, on le récupère de la BDD.



Pour savoir si le nom du personnage est valide, il va falloir implémenter une méthode `nomValide()` (je vous laisse réfléchir à quelle classe) qui retournera `true` ou `false` suivant si le nom est valide ou pas. Un nom est valide s'il n'est pas vide.

Cependant, avant de faire cela, il va falloir préparer le terrain.

- Un *autoload* devra être créé (bien que non indispensable puisqu'il n'y a que deux classes).

- Une instance de PDO devra être créée.

- Une instance de notre manager devra être créée.

Puisque notre manager a été créé, pourquoi ne pas afficher en haut de page le nombre de personnages créés ? 

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

spl_autoload_register('chargerClasse');

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

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un personnage.
{
  $perso = new Personnage(['nom' => $_POST['nom']]); // On crée un nouveau 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
  {
    $manager->add($perso);
  }
}

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

```
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
  </body>
</html>
```

Au cas où, je vous donne la méthode `nomValide()` de la classe `Personnage`. J'espère cependant que vous y êtes arrivés, un simple contrôle avec `empty()` et le tour est joué.

html+php

```
<?php
class Personnage
{
  // ...

  public function nomValide()
  {
    return !empty($this->_nom);
  }

  // ...
}
```

Une fois que nous avons un personnage, que se passera-t-il ? Il faut en effet cacher ce formulaire et laisser place à d'autres informations. Je vous propose d'afficher, dans un premier temps, les informations du personnage sélectionné (son nom et ses dégâts), puis, dans un second temps, la liste des autres personnages avec leurs informations. Il devra être possible de cliquer sur le nom du personnage pour le frapper.

php

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

spl_autoload_register('chargerClasse');

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

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un personnage.
{
  $perso = new Personnage(['nom' => $_POST['nom']]); // On crée un nouveau 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
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
```

```php
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?= htmlspecialchars($perso->nom()) ?><br />
        Dégâts : <?= $perso->degats() ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
    echo '<a href="?frapper=', $unPerso->id(), '">',
htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso->degats(), ')<br />';
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
```

Maintenant, quelque chose devrait vous titiller. En effet, si on recharge la page, on atterrira à nouveau sur le formulaire. Nous allons donc devoir utiliser le système de **sessions**. La première chose à faire sera alors de démarrer la session au début du script, juste après la déclaration de l'*autoload*. La session démarrée, nous pouvons aisément sauvegarder notre personnage. Pour cela, il nous faudra enregistrer le personnage en session (admettons dans `$_SESSION['perso']`) tout à la fin du code. Cela nous permettra, au début du script, de récupérer le personnage sauvegardé et de continuer le jeu.



Pour des raisons pratiques, il est préférable d'ajouter un lien de déconnexion pour qu'on puisse utiliser un autre personnage si on le souhaite. Ce lien aura pour effet de conduire à un `session_destroy()`.

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

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

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

$manager = new PersonnagesManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un
```

```php
$manager = new PersonnageManager($db);

if (isset($_POST['creer']) && isset($_POST['nom'])) // Si on a voulu créer un personnage.
{
  $perso = new Personnage(['nom' => $_POST['nom']]); // On crée un nouveau 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
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?= htmlspecialchars($perso->nom()) ?><br />
        Dégâts : <?= $perso->degats() ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
    echo '<a href="?frapper=', $unPerso->id(), '">', htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso->degats(), ')<br />';
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

Il reste maintenant une dernière partie à développer : celle qui s'occupera de frapper un personnage. Puisque nous avons déjà écrit tout le code faisant l'interaction entre l'attaquant et la cible, vous verrez que nous n'aurons presque rien à écrire.

Comment doit se passer la phase de traitement ? Avant toute chose, il faut bien vérifier que le joueur est connecté et que la variable `$perso` existe, sinon nous n'irons pas bien loin. Seconde vérification : il faut demander à notre manager si le personnage que l'on veut frapper existe bien. Si ces deux conditions sont vérifiées, alors on peut lancer l'attaque.

Pour lancer l'attaque, il va falloir récupérer le personnage à frapper grâce à notre manager. Ensuite, il suffira d'invoquer la méthode permettant de frapper le personnage. 

Cependant, nous n'allons pas nous arrêter là. N'oubliez pas que cette méthode peut retourner 3 valeurs différentes :

- `Personnage::CEST_MOI`. Le personnage a voulu se frapper lui-même.
- `Personnage::PERSONNAGE_FRAPPE`. Le personnage a bien été frappé.
- `Personnage::PERSONNAGE_TUE`. Le personnage a été tué.

Il va donc falloir afficher un message en fonction de cette valeur retournée. Aussi, seuls 2 de ces cas nécessitent une mise à jour de la BDD : si le personnage a été frappé ou s'il a été tué. En effet, si on a voulu se frapper soi-même, aucun des deux personnages impliqués n'a été modifié.

php

```php
<?php
// On enregistre notre autoload.
function chargerClasse($classname)
{
  require $classname.'.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 chaque fois qu'une requête a échoué.

$manager = new PersonnagesManager($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 personnage.
{
  $perso = new Personnage(['nom' => $_POST['nom']]); // On crée un nouveau 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
  {
    $manager->add($perso);
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}

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']))
    {
```

```php
      $message = 'Le personnage que vous voulez frapper n\'existe pas !';
    }

    else
    {
      $persoAFrapper = $manager->get((int) $_GET['frapper']);

      $retour = $perso->frapper($persoAFrapper); // On stocke dans $retour les éventuelles erreurs ou messages que renvoie la méthode frapper.

      switch ($retour)
      {
        case Personnage::CEST_MOI :
          $message = 'Mais... pourquoi voulez-vous vous frapper ???';
          break;

        case Personnage::PERSONNAGE_FRAPPE :
          $message = 'Le personnage a bien été frappé !';

          $manager->update($perso);
          $manager->update($persoAFrapper);

          break;

        case Personnage::PERSONNAGE_TUE :
          $message = 'Vous avez tué ce personnage !';

          $manager->update($perso);
          $manager->delete($persoAFrapper);

          break;
      }
    }
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat</title>

    <meta charset="utf-8" />
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche.
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Nom : <?= htmlspecialchars($perso->nom()) ?><br />
        Dégâts : <?= $perso->degats() ?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui frapper ?</legend>
      <p>
<?php
$persos = $manager->getList($perso->nom());

if (empty($persos))
{
  echo 'Personne à frapper !';
}

else
{
  foreach ($persos as $unPerso)
  {
    echo '<a href="?frapper=', $unPerso->id(), '">', htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso->degats(), ')<br />';
  }
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" />
        <input type="submit" value="Créer ce personnage" name="creer" />
        <input type="submit" value="Utiliser ce personnage" name="utiliser" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

Et voilà, vous avez un jeu opérationnel.

Améliorations possibles

Ce code est très basique, beaucoup d'améliorations sont possibles. En voici quelques unes :

- Un système de **niveau**. Vous pourriez très bien assigner à chaque personnage un niveau de 1 à 100. Le personnage bénéficierait aussi d'une expérience allant de 0 à 100. Lorsque l'expérience atteint 100, le personnage passe au niveau suivant.

  Indice : le **niveau** et l'**expérience** deviendraient des caractéristiques du personnage, donc... Pas besoin de vous le dire, je suis sûr que vous savez ce que ça signifie !

- Un système de **force**. La force du personnage pourrait augmenter en fonction de son niveau, et les dégâts infligés à la victime seront donc plus importants.

  Indice : de même, la **force** du personnage serait aussi une caractéristique du personnage.

- Un système de **limitation**. En effet, un personnage peut en frapper autant qu'il veut dans un laps de temps indéfini. Pourquoi ne pas le limiter à 3 coups par jour ?

  Indice : il faudrait que vous stockiez le **nombre de coups** portés par le personnage, ainsi que la **date du dernier coup porté**. Cela ferait donc deux nouveaux champs en BDD, donc deux nouvelles caractéristiques pour le personnage !

- Un système de **retrait de dégâts**. Chaque jour, si l'utilisateur se connecte, il pourrait voir ses dégâts se soustraire de 10 par exemple.

  Indice : il faudrait stocker la date de dernière connexion. À chaque connexion, vous regarderiez cette date. Si elle est inférieure à 24h, alors vous ne feriez rien. Sinon, vous retireriez 10 de dégâts au personnage puis mettriez à jour cette date de dernière connexion.

Et la liste peut être longue ! Je vous encourage vivement à essayer d'implémenter ces fonctionnalités et à laisser libre court à votre imagination, vous progresserez bien plus !

☐ J'ai terminé ce chapitre et je passe au suivant

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 connaitre 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, allons-y !

Notion d'héritage

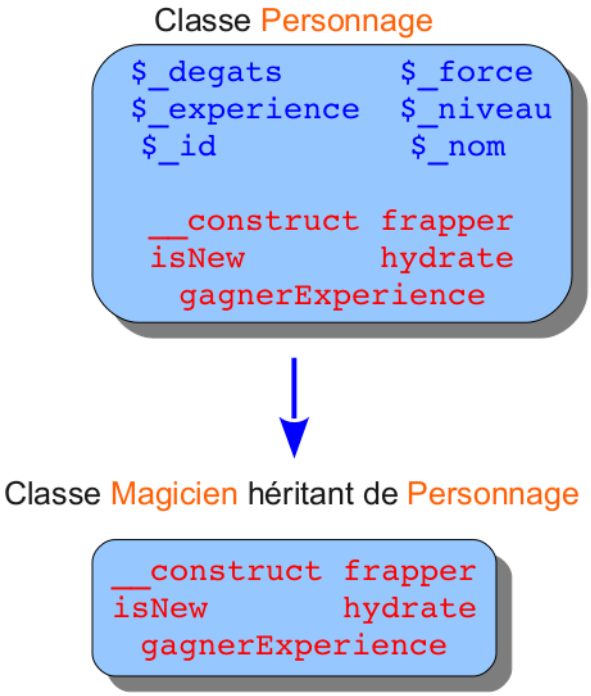
Définition

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 (figure suivante).



Exemple d'une classe Magicien héritant d'une classe Personnage

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 :

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

}

class Magicien extends Personnage // Notre classe Magicien hérite des attributs et méthodes de Personnage.
{

}
```

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 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**.

```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… Très utile ! En plus, tout ce qui est privé je n'y ai pas accès donc…

Nous sommes d'accord, si l'héritage c'était ça, 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 nous avons appris jusqu'à maintenant. Un exemple ?

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

  public function lancerUnSort($perso)
  {
    $perso->recevoirDegats($this->_magie); // On va dire que la magie du magicien représente sa force.
  }
}
```

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, seulement une notice si vous les avez activées disant que l'attribut n'existe pas.

Redéfinir 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 **réécrire** 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.

Nous allons maintenant essayer de redéfinir la méthode `gagnerExperience` afin de modifier l'attribut stockant la magie (admettons **$_magie**) lorsque, justement, on gagne de l'expérience. Problème : si on la réécrit, nous écrasons 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, suivi lui-même du nom de la méthode à appeler.

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

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

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

    if ($this->_magie < 100)
    {
      $this->_magie += 10;
    }
  }
}
```

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

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

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

    echo $retour;
  }
}

$b = new B;
$b->test(); // 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 non 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 redéfinissez une méthode, sa visibilité doit être la même ou plus permissive que dans la classe parente ! Si tel n'est pas le cas, une erreur fatale sera levée. Par exemple, vous ne pouvez redéfinir une méthode publique en disant qu'elle est privée, mais vous pouvez redéfinir une méthode privée en disant qu'elle est privée ou publique.

Héritez à l'infini !

Toute classe en POO peut être héritée si elle ne spécifie pas le contraire, **vraiment toutes.** Vous pouvez ainsi reproduire un arbre réel 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 :

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

}

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

}

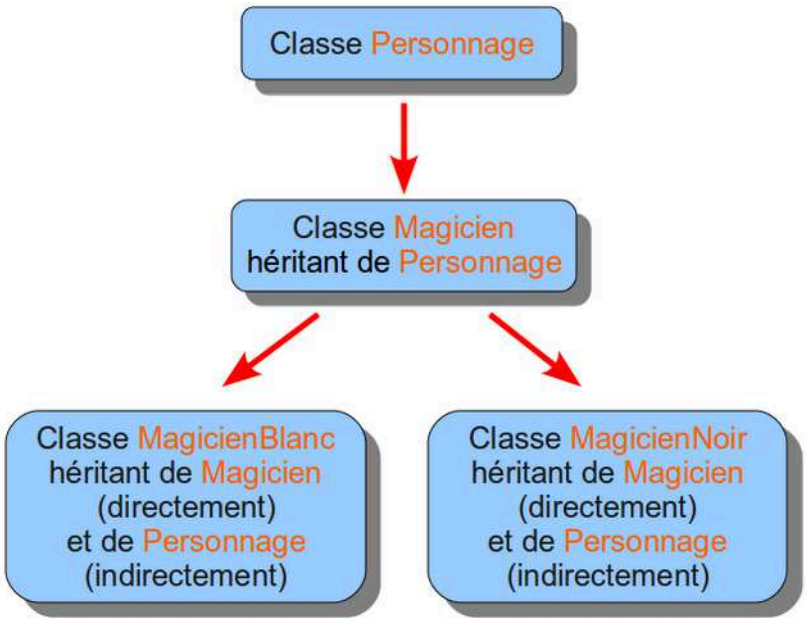
class MagicienBlanc extends Magicien // Classe MagicienBlanc héritant de Magicien.
{

}

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

}
```

Et un petit schéma qui reproduit ce code (figure suivante).



Exemple d'un héritage multiple

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 :

```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 celui-ci.
    echo $this->_attributPrive; // L'attribut est privé, on n'a pas accès celui-ci, donc rien ne s'affichera (mis à part une notice si vous les avez activées).
  }
}

$obj = new ClasseFille;

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

$obj->afficherAttributs(); // Affiche « Hello world ! » suivi de rien du tout ou d'une notice si vous les avez activées.
```

Comme vous pouvez le constater, il n'y a pas d'*underscores* précédant les noms d'éléments protégés. C'est encore une fois la notation PEAR qui nous dit que les noms d'éléments protégés ne sont pas précédés de ce caractère.

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 (utile si vous comptez partager votre 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.**

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 ?

Nous allons donc considérer la classe `Personnage` comme étant une classe **modèle** dont toute classe fille possédera les méthodes et attributs.

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

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

}

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

}

$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, il 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 + `function` + nomDeLaMethode + parenthèses avec ou sans paramètres + **point-virgule**).

```php
<?php
abstract class Personnage
{
  // On va forcer toute classe fille à écrire cette méthode car chaque personnage frappe différemment.
  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 abstraite « frapper » de la classe mère.
  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`.

```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 redéfinir**. Si vous déclarez votre méthode `recevoirDegats` en tant que méthode finale, vous ne pourrez la redéfinir.

```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

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.

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

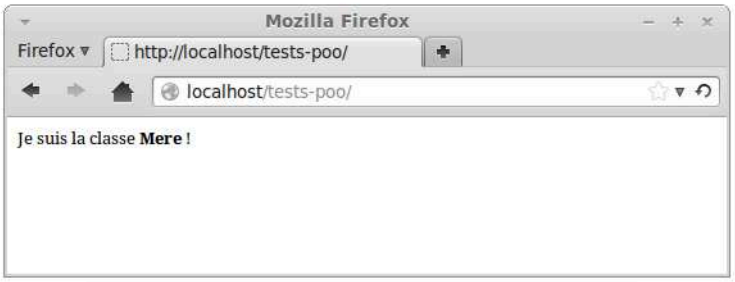
```php
<?php
class Mere
{
  public static function lancerLeTest()
  {
    self::quiEstCe();
  }

  public 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 :



Résultat affiché par le script

Mais qu'est-ce qu'il s'est passé ???

- Appel de la méthode `lancerLeTest` de la classe `Enfant` ;
- la méthode n'a pas été réécrite, on va donc « chercher » la méthode `lancerLeTest` de la classe mère ;
- appel de la méthode `quiEstCe` **de la 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 récrite ?

Tout simplement parce que `self::` fait appel à la méthode statique **de la classe dans laquelle est contenu `self::`**, donc de la classe parente.

Et la résolution statique à la volée dans tout ça ?

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` !

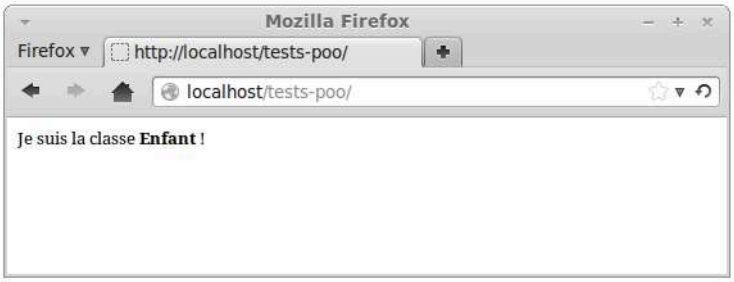
```php
<?php
class Mere
{
  public static function lancerLeTest()
  {
    static::quiEstCe();
  }

  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();
```

Ce qui donnera :



Résultat affiché par le script

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

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

  public function quiEstCe()
  {
    echo 'Je suis la classe « Mere » !';
  }
}

class Enfant extends Mere
{
  public function quiEstCe()
  {
    echo 'Je suis la classe « Enfant » !';
  }
}

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

Cas complexes

Au premier abord, vous n'avez peut-être pas tout compris. Si tel est le cas, ne lisez pas la suite cela risquerait de vous embrouiller davantage. Prenez bien le temps de comprendre ce qui est écrit 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 :

```php
<?php
class A
{

}

class B extends A
{

}

class C extends B
{

}
```

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

```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';
  }
}
```

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

Appelons maintenant cette méthode depuis la classe C dans un contexte statique (nul besoin de créer d'objet, mais ça marche tout aussi bien 😉).

```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();
```

Plaçons un peu de code dans cette méthode, sinon ce n'est pas drôle.

Nous allons 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 » :

```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, créons 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 ! 😉

```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 très compliqué mais fondamental à comprendre.

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

```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 ? Et 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, le code suivant affichera « A ».

```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();
```

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 :

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

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

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

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

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

$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.

En résumé

- Nous pouvons parler d'héritage entre une classe A et une classe B si et seulement si nous pouvons dire « B est un A » (dans ce cas-là, B hérite de A).
- Une classe héritant d'une autre a accès à tous ses attributs et méthodes publics et protégés.
- La visibilité `protected` est équivalente à la visibilité `private` à la différence près qu'un élément protégé est accessible par les classes filles, contrairement aux éléments privés.
- Il est possible d'interdire l'instanciation d'une classe grâce au mot-clé `abstract` (utile pour poser un modèle de base commun à plusieurs classes) et l'héritage d'une classe grâce au mot-clé `final`.
- Il est possible d'obliger ses classes filles à implémenter une méthode grâce au mot-clé `abstract` et de leur interdire la réécriture d'une méthode grâce au mot-clé `final`.
- La résolution statique à la volée permet de savoir quelle classe a été initialement appelée afin d'invoquer des méthodes ou accéder à des attributs de la classe appelée.

☐ J'ai terminé ce chapitre et je passe au suivant

TP : DES PERSONNAGES SPÉCIALISÉS

Avez-vous bien tout retenu du dernier chapitre ? C'est ce que l'on va vérifier maintenant ! En effet, l'héritage est un point très important de la POO qu'il est essentiel de maîtriser, c'est pourquoi je souhaite insister dessus à travers ce TP. Ce dernier est en fait une modification du premier (celui qui mettait en scène notre personnage). Nous allons donc ajouter une possibilité supplémentaire au script : le choix du personnage. Cette modification vous permettra de revoir ce que nous avons abordé depuis le dernier TP, à savoir :

- L'héritage ;
- la portée `protected` ;
- l'abstraction ;
- la résolution statique à la volée.

Ce que nous allons faire

Cahier des charges

Je veux que nous ayons 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 il devra être impossible d'instancier la classe `Personnage`). 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 nouvelle fonctionnalité : celle de lancer un sort qui aura pour effet d'endormir un personnage pendant `$atout * 6` heures (l'attribut `$atout` représente la dose de magie du personnage).
- Un guerrier. Lorsqu'un coup lui est porté, il devra avoir la possibilité de parer le coup en fonction de sa protection (son atout).

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

Comme vous le voyez, chaque personnage possède un **atout**. Cet atout devra être augmenté lorsque le personnage est amené à s'en servir (c'est-à-dire lorsque le magicien lance un sort ou que le guerrier subit des dégâts).

Des nouvelles fonctionnalités pour chaque personnage

Étant donné qu'un magicien peut endormir un personnage, il est nécessaire d'implémenter deux nouvelles fonctionnalités :

- Celle consistant à savoir si un personnage est endormi ou non (nécessaire lorsque ledit personnage voudra en frapper un autre : s'il est endormi, ça ne doit pas être possible).
- Celle consistant à obtenir la date du réveil du personnage sous la forme « XX heures, YY minutes et ZZ secondes », qui s'affichera dans le cadre d'information du personnage s'il est endormi.

La base de données

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 ou guerrier). Nous allons donc créer une colonne **type** et une colonne **atout** (l'attribut qu'il a en plus). Une colonne **timeEndormi** 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) :

```mysql
CREATE TABLE IF NOT EXISTS `personnages_v2` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `nom` varchar(50) COLLATE utf8_general_ci NOT NULL,
  `degats` tinyint(3) unsigned NOT NULL DEFAULT '0',
  `timeEndormi` int(10) unsigned NOT NULL DEFAULT '0',
  `type` enum('magicien','guerrier') COLLATE utf8_general_ci NOT NULL,
  `atout` tinyint(3) unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_general_ci;
```

Le coup de pouce du démarrage

Les modifications que je vous demande peuvent vous donner mal à la tête, c'est pourquoi je me propose de vous mettre sur la voie. Procédons classe par classe.

Le personnage

Là où il y a peut-être une difficulté, c'est pour déterminer l'attribut `$type`. En effet, où doit-on assigner cette valeur ? Qui doit le faire ?

Pour des raisons évidentes de risques de bugs, ce ne sera pas à l'utilisateur d'assigner la valeur « guerrier » ou « magicien » à cet attribut, mais à la classe elle-même. Cependant, il serait redondant dans chaque classe fille de **Personnage** de faire un `<?php $this->type = 'magicien'` par exemple, alors on va le faire une bonne fois pour toute dans la classe **Personnage**.

Comment cela est-il possible ?

Je vais vous dévoiler une petite fonctionnalité de PHP. Dans toutes les méthodes d'une classe, il est possible d'accéder à son nom grâce à `::class`. Pour ce qui est à placer avant ce double deux-points, cela dépend de ce que vous voulez obtenir. Généralement, vous aurez le choix entre `self` et `static`. Si vous avez bien compris la partie sur la résolution statique à la volée, vous avez peut-être compris là où je voulais en venir. Essayez ce code :

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

class Fille extends Mere
{

}

new Fille;
```

À l'écran s'affichera « Fille », car il s'agit de la classe que l'on a appelée. Par conséquent, on souhaite afficher le nom de la classe **Fille**. Si l'on avait mis `self::class`, on aurait affiché le nom de la classe **Mere**.

Cette technique de détermination du nom de classe n'est disponible que depuis la version 5.5 de PHP. Si vous avez une version antérieure, vous pouvez utiliser get_called_class qui retournera exactement le même résultat (cette fonction s'appuie elle aussi sur le late static binding).

Pour des raisons pratiques, le type du personnage doit être en minuscules. Un petit appel à strtolower() sera donc indispensable.

Le magicien

Qui dit nouvelle fonctionnalité dit nouvelle méthode. Votre classe `Magicien` devra donc implémenter une nouvelle méthode permettant de lancer un sort. Celle-ci devra vérifier plusieurs points :

- La cible à ensorceler n'est pas le magicien qui lance le sort.
- Le magicien possède encore de la magie (l'atout n'est pas à 0).

Le guerrier

Ce qu'on cherche à faire ici est **modifier** le comportement du personnage lorsqu'il subit des dégâts. Nous allons donc **modifier** la méthode qui se charge d'ajouter des dégâts au personnage. Cette méthode procédera de la sorte :

- Elle calculera d'abord la valeur de l'atout.
- Elle augmentera les dégâts en prenant soin de prendre en compte l'atout.
- Elle indiquera si le personnage a été frappé ou tué.

Tu nous parles d'un atout depuis tout à l'heure, mais comment est-ce qu'on le détermine ?

L'atout du magicien et du guerrier se déterminent de la même façon :

- Si les dégâts sont compris entre 0 et 25, alors l'atout sera de 4.
- Si les dégâts sont compris entre 25 et 50, alors l'atout sera de 3.
- Si les dégâts sont compris entre 50 et 75, alors l'atout sera de 2.
- Si les dégâts sont compris entre 75 et 90, alors l'atout sera de 1.
- Sinon, il sera de 0.

Comment sont-ils exploités ?

Du côté du guerrier, j'utilise une simple formule : la dose de dégâts reçu ne sera pas de 5, mais de `5 - $atout`. Du côté du magicien, là aussi j'utilise une simple formule : il endort sa victime pendant `($this->atout * 6) * 3600` secondes.

Voir le résultat que vous devez obtenir

Comme au précédent TP, le résultat comporte toutes les améliorations proposées en fin de chapitre.

Correction

Nous allons maintenant corriger le TP. Les codes seront d'abord précédés d'explications pour bien mettre au clair ce qui était demandé.

Commençons d'abord par notre classe `Personnage`. Celle-ci devait implémenter deux nouvelles méthodes (sans compter les *getters* et *setters*) : `estEndormi()` et `reveil()`. Aussi il ne fallait pas oublier de modifier la méthode `frapper()` afin de bien vérifier que le personnage qui frappe n'était pas endormi ! Enfin, il fallait assigner la bonne valeur à l'attribut `$type` dans le constructeur.

Il ne fallait pas mettre de *setter* pour l'attribut `$type`. En effet, le type d'un personnage est constant (un magicien ne se transforme pas en guerrier). Il est défini dans le constructeur et l'utilisateur ne pourra pas changer sa valeur (imaginez le non-sens qu'il y aurait si l'utilisateur mettait « magicien » à l'attribut `$type` d'un objet `Guerrier`).

```php
<?php
abstract class Personnage
{
  protected $atout,
            $degats,
            $id,
            $nom,
            $timeEndormi,
            $type;

  const CEST_MOI = 1; // Constante renvoyée par la méthode `frapper` si on se frappe soit-même.
  const PERSONNAGE_TUE = 2; // Constante renvoyée par la méthode `frapper` si on a tué le personnage en le frappant.
  const PERSONNAGE_FRAPPE = 3; // Constante renvoyée par la méthode `frapper` si on a bien frappé le personnage.
  const PERSONNAGE_ENSORCELE = 4; // Constante renvoyée par la méthode `lancerUnSort` (voir classe Magicien) si on a bien ensorcelé un personnage.
  const PAS_DE_MAGIE = 5; // Constante renvoyée par la méthode `lancerUnSort` (voir classe Magicien) si on veut jeter un sort alors que la magie du magicien est à 0.
  const PERSO_ENDORMI = 6; // Constante renvoyée par la méthode `frapper` si le personnage qui veut frapper est endormi.

  public function __construct(array $donnees)
  {
    $this->hydrate($donnees);
    $this->type = strtolower(static::class);
  }

  public function estEndormi()
  {
    return $this->timeEndormi > time();
  }

  public function frapper(Personnage $perso)
  {
    if ($perso->id == $this->id)
    {
      return self::CEST_MOI;
    }

    if ($this->estEndormi())
    {
      return self::PERSO_ENDORMI;
    }

    // On indique au personnage qu'il doit recevoir des dégâts.
    // Puis on retourne la valeur renvoyée par la méthode : self::PERSONNAGE_TUE ou self::PERSONNAGE_FRAPPE.
    return $perso->recevoirDegats();
  }

  public function hydrate(array $donnees)
  {
    foreach ($donnees as $key => $value)
    {
      $method = 'set'.ucfirst($key);

      if (method_exists($this, $method))
      {
        $this->$method($value);
      }
    }
  }

  public function nomValide()
  {
    return !empty($this->nom);
  }

  public function recevoirDegats()
  {
    $this->degats += 5;

    // 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;
  }

  public function reveil()
  {
    $secondes = $this->timeEndormi;
    $secondes -= time();
```

```php
    $secondes = $this->timeEndormi;
    $secondes -= time();

    $heures = floor($secondes / 3600);
    $secondes -= $heures * 3600;
    $minutes = floor($secondes / 60);
    $secondes -= $minutes * 60;

    $heures .= $heures <= 1 ? ' heure' : ' heures';
    $minutes .= $minutes <= 1 ? ' minute' : ' minutes';
    $secondes .= $secondes <= 1 ? ' seconde' : ' secondes';

    return $heures . ', ' . $minutes . ' et ' . $secondes;
  }

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

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

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

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

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

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

  public function setAtout($atout)
  {
    $atout = (int) $atout;

    if ($atout >= 0 && $atout <= 100)
    {
      $this->atout = $atout;
    }
  }

  public function setDegats($degats)
  {
    $degats = (int) $degats;

    if ($degats >= 0 && $degats <= 100)
    {
      $this->degats = $degats;
    }
  }

  public function setId($id)
  {
    $id = (int) $id;

    if ($id > 0)
    {
      $this->id = $id;
    }
  }

  public function setNom($nom)
  {
    if (is_string($nom))
    {
      $this->nom = $nom;
    }
  }

  public function setTimeEndormi($time)
  {
    $this->timeEndormi = (int) $time;
  }
}
```

Penchons-nous maintenant vers la classe `Guerrier`. Celle-ci devait modifier la méthode `recevoirDegats()` afin d'ajouter une parade lors d'une attaque.

```php
<?php
class Guerrier extends Personnage
{
  public function recevoirDegats()
  {
    if ($this->degats >= 0 && $this->degats <= 25)
    {
      $this->atout = 4;
    }
    elseif ($this->degats > 25 && $this->degats <= 50)
    {
      $this->atout = 3;
    }
    elseif ($this->degats > 50 && $this->degats <= 75)
    {
      $this->atout = 2;
    }
    elseif ($this->degats > 50 && $this->degats <= 90)
    {
      $this->atout = 1;
    }
    else
    {
      $this->atout = 0;
    }

    $this->degats += 5 - $this->atout;

    // 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;
  }
}
```

Enfin, il faut maintenant s'occuper du magicien. La classe le représentant devait ajouter une nouvelle fonctionnalité : celle de pouvoir lancer un sort.

```php
<?php
class Magicien extends Personnage
{
  public function lancerUnSort(Personnage $perso)
  {
    if ($this->degats >= 0 && $this->degats <= 25)
    {
      $this->atout = 4;
    }
    elseif ($this->degats > 25 && $this->degats <= 50)
    {
      $this->atout = 3;
    }
    elseif ($this->degats > 50 && $this->degats <= 75)
    {
      $this->atout = 2;
    }
    elseif ($this->degats > 50 && $this->degats <= 90)
    {
      $this->atout = 1;
    }
    else
    {
      $this->atout = 0;
    }

    if ($perso->id == $this->id)
    {
      return self::CEST_MOI;
    }

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

    if ($this->estEndormi())
    {
      return self::PERSO_ENDORMI;
    }

    $perso->timeEndormi = time() + ($this->atout * 6) * 3600;

    return self::PERSONNAGE_ENSORCELE;
  }
}
```

Passons maintenant au manager. Nous allons toujours garder un seul manager parce que nous gérons toujours des personnages ayant la même structure. Les modifications sont mineures : il faut juste ajouter les deux nouveaux champs dans les requêtes et instancier la bonne classe lorsqu'on récupère un personnage.

```php
<?php
class PersonnagesManager
{
  private $db; // Instance de PDO

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

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

    $q->bindValue(':nom', $perso->nom());
    $q->bindValue(':type', $perso->type());

    $q->execute();

    $perso->hydrate([
      'id' => $this->db->lastInsertId(),
      'degats' => 0,
      'atout' => 0
    ]);
  }

  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 existe.
    {
      return (bool) $this->db->query('SELECT COUNT(*) FROM personnages_v2 WHERE id = '.$info)->fetchColumn();
    }

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

    $q = $this->db->prepare('SELECT COUNT(*) FROM personnages_v2 WHERE nom = :nom');
    $q->execute([':nom' => $info]);

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

  public function get($info)
  {
    if (is_int($info))
    {
      $q = $this->db->query('SELECT id, nom, degats, timeEndormi, type, atout FROM personnages_v2 WHERE id = '.$info);
      $perso = $q->fetch(PDO::FETCH_ASSOC);
    }

    else
    {
      $q = $this->db->prepare('SELECT id, nom, degats, timeEndormi, type, atout FROM personnages_v2 WHERE nom = :nom');
      $q->execute([':nom' => $info]);

      $perso = $q->fetch(PDO::FETCH_ASSOC);
    }

    switch ($perso['type'])
    {
      case 'guerrier': return new Guerrier($perso);
      case 'magicien': return new Magicien($perso);
      default: return null;
    }
  }

  public function getList($nom)
  {
    $persos = [];

    $q = $this->db->prepare('SELECT id, nom, degats, timeEndormi, type, atout FROM personnages_v2 WHERE nom <> :nom ORDER BY nom');
    $q->execute([':nom' => $nom]);

    while ($donnees = $q->fetch(PDO::FETCH_ASSOC))
    {
      switch ($donnees['type'])
      {
        case 'guerrier': $persos[] = new Guerrier($donnees); break;
        case 'magicien': $persos[] = new Magicien($donnees); break;
      }
    }

    return $persos;
  }

  public function update(Personnage $perso)
  {
    $q = $this->db->prepare('UPDATE personnages_v2 SET degats = :degats, timeEndormi = :timeEndormi, atout = :atout WHERE id = :id');

    $q->bindValue(':degats', $perso->degats(), PDO::PARAM_INT);
    $q->bindValue(':timeEndormi', $perso->timeEndormi(), PDO::PARAM_INT);
    $q->bindValue(':atout', $perso->atout(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);
```

```
    $q->bindValue(':atout', $perso->atout(), PDO::PARAM_INT);
    $q->bindValue(':id', $perso->id(), PDO::PARAM_INT);

    $q->execute();
  }
}
```

Enfin, finissons par la page d'index qui a légèrement changé. Quelles sont les modifications ?

- Le formulaire doit proposer au joueur de choisir le type du personnage qu'il veut créer.
- Lorsqu'on crée un personnage, le script doit créer une instance de la classe désirée et non de `Personnage`.
- Lorsqu'on veut frapper un personnage, on vérifie que l'attaquant n'est pas endormi.
- Un lien permettant d'ensorceler un personnage doit être ajouté pour les magiciens ainsi que l'antidote qui va avec.

php

```php
<?php
function chargerClasse($classe)
{
  require $classe . '.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 personnage.
{
  switch ($_POST['type'])
  {
    case 'magicien' :
      $perso = new Magicien(['nom' => $_POST['nom']]);
      break;

    case 'guerrier' :
      $perso = new Guerrier(['nom' => $_POST['nom']]);
      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
    {
      $manager->add($perso);
    }
  }
}

elseif (isset($_POST['utiliser']) && isset($_POST['nom'])) // Si on a voulu utiliser un personnage.
{
  if ($manager->exists($_POST['nom'])) // Si celui-ci existe.
  {
    $perso = $manager->get($_POST['nom']);
  }
  else
  {
    $message = 'Ce personnage n\'existe pas !'; // S'il n'existe pas, on affichera ce message.
  }
}

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.';
  }

```

```php
    $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
    {
      $persoAFrapper = $manager->get((int) $_GET['frapper']);
      $retour = $perso->frapper($persoAFrapper); // On stocke dans $retour les éventuelles erreurs ou messages que renvoie la méthode frapper.

      switch ($retour)
      {
        case Personnage::CEST_MOI :
          $message = 'Mais... pourquoi voulez-vous vous frapper ???';
          break;

        case Personnage::PERSONNAGE_FRAPPE :
          $message = 'Le personnage a bien été frappé !';

          $manager->update($perso);
          $manager->update($persoAFrapper);

          break;

        case Personnage::PERSONNAGE_TUE :
          $message = 'Vous avez tué ce personnage !';

          $manager->update($perso);
          $manager->delete($persoAFrapper);

          break;

        case Personnage::PERSO_ENDORMI :
          $message = 'Vous êtes endormi, vous ne pouvez pas frapper de personnage !';
          break;
      }
    }
  }
}

elseif (isset($_GET['ensorceler']))
{
  if (!isset($perso))
  {
    $message = 'Merci de créer un personnage ou de vous identifier.';
  }

  else
  {
    // Il faut bien vérifier que le personnage est un magicien.
    if ($perso->type() != 'magicien')
    {
      $message = 'Seuls les magiciens peuvent ensorceler des personnages !';
    }

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

      else
      {
        $persoAEnsorceler = $manager->get((int) $_GET['ensorceler']);
        $retour = $perso->lancerUnSort($persoAEnsorceler);

        switch ($retour)
        {
          case Personnage::CEST_MOI :
            $message = 'Mais... pourquoi voulez-vous vous ensorceler ???';
            break;

          case Personnage::PERSONNAGE_ENSORCELE :
            $message = 'Le personnage a bien été ensorcelé !';

            $manager->update($perso);
            $manager->update($persoAEnsorceler);

            break;

          case Personnage::PAS_DE_MAGIE :
            $message = 'Vous n\'avez pas de magie !';
            break;

          case Personnage::PERSO_ENDORMI :
            $message = 'Vous êtes endormi, vous ne pouvez pas lancer de sort !';
            break;
        }
      }
    }
  }
}
?>
<!DOCTYPE html>
<html>
  <head>
    <title>TP : Mini jeu de combat - Version 2</title>

    <meta charset="utf-8" />
  </head>
  <body>
```

```
  </head>
  <body>
    <p>Nombre de personnages créés : <?= $manager->count() ?></p>
<?php
if (isset($message)) // On a un message à afficher ?
{
  echo '<p>', $message, '</p>'; // Si oui, on l'affiche
}

if (isset($perso)) // Si on utilise un personnage (nouveau ou pas).
{
?>
    <p><a href="?deconnexion=1">Déconnexion</a></p>

    <fieldset>
      <legend>Mes informations</legend>
      <p>
        Type : <?= ucfirst($perso->type()) ?><br />
        Nom : <?= htmlspecialchars($perso->nom()) ?><br />
        Dégâts : <?= $perso->degats() ?><br />
<?php
// On affiche l'atout du personnage suivant son type.
switch ($perso->type())
{
  case 'magicien' :
    echo 'Magie : ';
    break;

  case 'guerrier' :
    echo 'Protection : ';
    break;
}

echo $perso->atout();
?>
      </p>
    </fieldset>

    <fieldset>
      <legend>Qui attaquer ?</legend>
      <p>
<?php
// On récupère tous les personnages par ordre alphabétique, dont le nom est différent de celui de notre personnage (on va pas se frapper nous-même :p).
$retourPersos = $manager->getList($perso->nom());

if (empty($retourPersos))
{
  echo 'Personne à frapper !';
}

else
{
  if ($perso->estEndormi())
  {
    echo 'Un magicien vous a endormi ! Vous allez vous réveiller dans ', $perso->reveil(), '.';
  }

  else
  {
    foreach ($retourPersos as $unPerso)
    {
      echo '<a href="?frapper=', $unPerso->id(), '">', htmlspecialchars($unPerso->nom()), '</a> (dégâts : ', $unPerso->degats(), ' | type : ', $unPerso->type(), ')';

      // On ajoute un lien pour lancer un sort si le personnage est un magicien.
      if ($perso->type() == 'magicien')
      {
        echo ' | <a href="?ensorceler=', $unPerso->id(), '">Lancer un sort</a>';
      }

      echo '<br />';
    }
  }
}
?>
      </p>
    </fieldset>
<?php
}
else
{
?>
    <form action="" method="post">
      <p>
        Nom : <input type="text" name="nom" maxlength="50" /> <input type="submit" value="Utiliser ce personnage" name="utiliser" /><br />
        Type :
        <select name="type">
          <option value="magicien">Magicien</option>
          <option value="guerrier">Guerrier</option>
        </select>
        <input type="submit" value="Créer ce personnage" name="creer" />
      </p>
    </form>
<?php
}
?>
  </body>
</html>
<?php
if (isset($perso)) // Si on a créé un personnage, on le stocke dans une variable session afin d'économiser une requête SQL.
{
  $_SESSION['perso'] = $perso;
}
```

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 !

Améliorations possibles

Comme au précédent TP, beaucoup d'améliorations sont possibles, à commencer par celles déjà exposées dans le chapitre dudit TP :

- Un système de **niveau**. Vous pourriez très bien assigner à chaque personnage un niveau de 1 à 100. Le personnage bénéficierait aussi d'une expérience allant de 0 à 100. Lorsque l'expérience atteint 100, le personnage passe au niveau suivant.

  Indice : le **niveau** et l'**expérience** deviendraient des caractéristiques du personnage, donc... Pas besoin de vous le dire, je suis sûr que vous savez ce que ça signifie !

- Un système de **force**. La force du personnage pourrait augmenter en fonction de son niveau, les dégâts infligés à la victime seront donc plus importants.

  Indice : de même, la **force** du personnage serait aussi une caractéristique du personnage.

- Un système de **limitation**. En effet, un personnage peut en frapper autant qu'il veut dans un laps de temps indéfini. Pourquoi ne pas le limiter à 3 coups par jour ?

  Indice : il faudrait que vous stockiez le **nombre de coups** portés par le personnage, ainsi que la **date du dernier coup porté**. Cela ferait donc deux nouveaux champs en BDD, et deux nouvelles caractéristiques pour le personnage !

- Un système de **retrait de dégâts**. Chaque jour, si l'utilisateur se connecte, il pourrait voir ses dégâts se soustraire de 10 par exemple.

  Indice : il faudrait stocker la date de dernière connexion. À chaque connexion, vous regarderiez cette date. Si elle est inférieure à 24h, alors vous ne feriez rien. Sinon, vous retireriez 10 de dégâts au personnage puis mettriez à jour cette date de dernière connexion.

Pour reprendre cet esprit, vous pouvez vous entraîner à créer d'autres personnages, comme une **brute** par exemple. Son atout dépendrait aussi de ses dégâts et viendrait augmenter sa force lors d'une attaque.

La seule différence avec le premier TP est l'apparition de nouveaux personnages, les seules améliorations différentes possibles sont donc justement la création de nouveaux personnages. Ainsi je vous encourage à imaginer tout un tas de différents personnages tous aussi farfelus les uns que les autres : cela vous fera grandement progresser et vous aidera à vous familiariser avec l'héritage !

☐ J'ai terminé ce chapitre et je passe au suivant

LES MÉTHODES MAGIQUES

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

Le principe

Vous devez sans doute vous poser une grande question à la vue du 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 évènement. Si la méthode n'existe pas et que l'évènement est exécuté, aucun effet « spécial » ne sera ajouté, l'évènement s'exécutera normalement. Le but des méthodes magiques est d'intercepter un évènement, dire de faire ceci ou cela et retourner une valeur utile pour l'évènement si besoin il y a.

Bonne nouvelle : vous connaissez déjà une méthode magique !

Si si, cherchez bien au fond de votre tête… Et oui, la méthode `__construct` est magique ! Comme nous l'avons vu plus haut, chaque méthode magique s'exécute au moment où un évènement est lancé. L'évènement 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 :

```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. Au même titre que `__construct` , `__destruct` a lui aussi un petit nom : il s'agit du destructeur.

La surcharge magique des attributs et méthodes

Parlons maintenant des méthode liées à la surcharge magique des attributs et méthodes.

Euh, deux secondes là… C'est quoi la « surcharge magique des attributs et méthodes » ?

Vous avez raison, il serait d'abord préférable d'expliquer ceci. La surcharge magique d'attributs ou méthodes consiste à créer dynamiquement des attributs et méthodes. Cela est possible lorsque l'on tente d'accéder à un élément qui n'existe pas ou auquel on n'a pas accès (s'il est privé et qu'on tente d'y accéder depuis l'extérieur de la classe par exemple). 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 lorsque l'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 simplement faire ce que bon vous semble !

Exemple :

```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 <strong>', $valeur, '</strong> mais c\'est pas possible !<br />';
  }
}

$obj = new MaClasse;

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

À la sortie s'affichera :



Résultat affiché par le script

Tenez, petit exercice, stockez dans un tableau tous les attributs (avec leurs valeurs) que nous avons essayé de modifier ou créer.

Solution :

```php
<?php
class MaClasse
{
  private $attributs = [];
  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 à faire, mais cela permet de pratiquer un peu.

Parlons maintenant de `__get`. Cette méthode est appelée lorsque l'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 inaccessible).

Exemple :

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

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

$obj = new MaClasse;

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

Ce qui va afficher :



Résultat affiché par le script

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 :

```php
<?php
class MaClasse
{
  private $attributs = [];
  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->autreAtribut;
```



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 :

```php
<?php
class MaClasse
{
  private $attributs = [];
  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 :



Résultat affiché par le script

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.

php

```php
<?php
class MaClasse
{
  private $attributs = [];
  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 :



Résultat affiché par le script

Finissons par « __call » et « __callStatic »

Laissons de côté les attributs pour le moment et parlons 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 que l'on a essayé d'appeler et le second est la liste des arguments qui lui ont été passés (sous forme de tableau).

Exemple :

```php
<?php
class MaClasse
{
  public function __call($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode($arguments, '</strong>, <strong>'), '</strong>';
  }
}

$obj = new MaClasse;

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

Résultat :



Résultat affiché par le script

Et si on essaye d'appeler une méthode qui n'existe pas statiquement ? Et 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` !

```php
<?php
class MaClasse
{
  public function __call($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode($arguments, '</strong>, <strong>'), '</strong><br />';
  }

  public static function __callStatic($nom, $arguments)
  {
    echo 'La méthode <strong>', $nom, '</strong> a été appelée dans un contexte statique alors qu\'elle n\'existe pas ! Ses arguments étaient les suivants : <strong>', implode ($arguments, '</strong>, <strong>'), '</strong><br />';
  }
}

$obj = new MaClasse;

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

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

Résultat :



Résultat affiché par le script

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. Lisez le jusqu'à la partie **Encore plus fort !**, et je vais mieux vous expliquer à partir de là !

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 doutes 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 quel est 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 passe 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 `['serveur', 'login'];`.

Voici ce que pourrait donner notre classe `Connexion` :

```php
<?php
class Connexion
{
  protected $pdo, $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();
  }

  protected function connexionBDD()
  {
    $this->pdo = new PDO('mysql:host='.$this->serveur.';dbname='.$this->dataBase,
$this->utilisateur, $this->motDePasse);
  }

  public function __sleep()
  {
    // Ici sont à placer des instructions à exécuter juste avant la linéarisation.
    // On retourne ensuite la liste des attributs qu'on veut sauver.
    return ['serveur', 'utilisateur', 'motDePasse', 'dataBase'];
  }
}
```

Ainsi, vous pourrez faire ceci :

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

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

« unserialize » et « __wakeup »

Maintenant, nous allons simplement implémenter la fonction `__wakeup`. Qu'allons-nous mettre dedans ?

Rien de compliqué… Nous allons 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` !

```php
<?php
class Connexion
{
  protected $pdo, $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();
  }

  protected function connexionBDD()
  {
    $this->pdo = new PDO('mysql:host='.$this->serveur.';dbname='.$this->dataBase,
$this->utilisateur, $this->motDePasse);
  }

  public function __sleep()
  {
    return ['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 :

```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 l'objet désérialisé sera une instance de `__PHP_Incomplete_Class_Name`, classe qui ne contient aucune méthode (cela produira donc un objet inutile). 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`, `__invoke` et `__debugInfo`.

« __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 :

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

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

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

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

// Solution 1 : le cast

$texte = (string) $obj;
var_dump($texte); // Affiche : string(13) "Hello world !".

// Solution 2 : directement dans un echo
echo $obj; // 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 (`['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 :

```php
<?php
class Export
{
  protected $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 avec les attributs de l'objet que l'on veut exporter.
    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 ayant les mêmes attributs que l'objet précédent.

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

Le code affichera donc :



Résultat affiché par le script

« __invoke »

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 :

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

Essayez ce code et… BAM ! Une erreur fatale. Pour résoudre ce problème, nous allons 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 :

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

$obj = new MaClasse;

$obj(5); // Affiche « 5 ».
```

« __debugInfo »

> ✗ Cette méthode magique n'est disponible que depuis la version 5.6 de PHP.

Cette méthode magique est invoquée sur notre objet lorsqu'on appelle la fonction var_dump. Pour ceux qui ne connaissent pas son principe, cette fonction permet d'obtenir des informations sur la variable qu'on lui donne. Si on lui donne un objet, var_dump va afficher les détails de tous les attributs de l'objet, qu'ils soient publics, protégés ou privés. La méthode magique `__debugInfo` permet de modifier ce comportement en ne sélectionnant que les attributs à afficher ainsi que ce qu'il faut afficher. Pour ce faire, cette méthode renverra sous forme de tableau associatif la liste des attributs à afficher avec leurs valeurs. Voici un exemple d'utilisation (je n'ai volontairement pas implémenté beaucoup de méthodes pour ne pas encombrer l'exemple) :

```php
<?php
class FileReader
{
    protected $f;

    public function __construct($path)
    {
        $this->f = fopen($path, 'c+');
    }

    public function __debugInfo()
    {
        return ['f' => fstat($this->f)];
    }
}

$f = new FileReader('fichier.txt');
var_dump($f); // Affiche les informations retournées par fstat.
```

On a ici un gestionnaire de fichiers qui nous permet de gérer facilement la lecture et l'écriture d'un fichier (du moins, on imagine que ça le fait). Un `var_dump` sur notre objet ne serait pas très révélateur. Par contre, obtenir les informations sur le fichier actuellement ouvert le serait plus, et c'est précisément ce que l'on fait en écrivant la méthode `__debugInfo` avec l'appel de `fstat`.

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

En résumé

- Les méthodes magiques sont des méthodes qui sont appelées automatiquement lorsqu'un certain évènement est déclenché.
- Toutes les méthodes magiques commencent par deux *underscores*, évitez donc d'appeler vos méthodes suivant ce même modèle.
- Les méthodes magiques dont vous vous servirez le plus souvent sont `__construct`, `__set`, `__get` et `__call`. Les autres sont plus « gadget » et vous les rencontrerez moins souvent.